Metodología para enfocar los problemas de PD, PDR, BT, AG, PLI y Grafos

Introducción

En este capítulo vamos a comparar las técnicas *Programación Dinámica (PD), Programación Dinámica tipo Reducción (PDR), Algoritmos de Vuelta Atrás (Backtracking, BT*), *Algoritmos Genéticos (AG)* y *Programación Lineal Entera (PLI)*. Muchos problemas se pueden resolver por todas pero alguna es más conveniente que otras. Otros problemas se resuelven, o al menos se resuelven más fácilmente solamente por algunas y no por otras.

Los problemas que se resuelven mediante *Programación Dinámica tipo Reducción (PDR), Algoritmos de Vuelta Atrás (Backtracking, BT*), *Algoritmos Genéticos (AG)* tienen en común que las soluciones de los problemas pueden codificarse como listas de valores enteros (en casos más generales podrían ser de otros tipos pero aquí nos limitamos a los enteros). En el primer y segundo caso esas listas son listas de alternativas. En el tercer caso la lista resultante de decodificar un cromosoma. Cada lista de valores podemos verla, por otra parte, como un conjunto de variables que deben cumplir las restricciones del problema. Si las restricciones son lineales entonces el problema puede ser resuelto también mediante *Programación Lineal Entera.*

Si buscamos una solución del problema todas las técnicas suelen ser adecuadas. Si buscamos más de una solución, todas las soluciones o el número de ellas las técnicas de *BT*, cuando se puede aplicar, es muy adecuada. Si buscamos una solución y es posible aplicar PDR o BT la decisión viene dada por el hecho de que haya problemas compartidos. S los hay es más adecuado usar PDR, que usa memoria. Si no hay problemas compartidos BT es más fácil de aplicar.

Esos elementos comunes entre los cuatro tipos de algoritmos nos permiten plantear soluciones a un problema mediante una de las técnicas inspirándonos en la solución dada por otra.

Aquí vamos a partir de la definición de un problema de optimización mediante una función objetivo y un conjunto de restricciones y desde ahí buscaremos los detalles concretos de cada uno de los algoritmos para resolver el problema. Con los misma metodología abordaremos problemas de búsqueda de una o varias solución en un dominio definido por un conjunto de restricciones o contar el número de las mismas.

Problema de optimización y búsqueda de soluciones valores en un dominio.

Los problemas que pretenden optimizar una función objetivo. Estos problemas pueden escribirse de la forma:

$$\min_{x \in \Omega} f(x)$$

Donde x es un vector de n variables y Ω un dominio dónde las variables deben tomar valores. Los valores de que cumplen $x \in \Omega$ los llamaremos valores válidos. Consideramos que la función objetivo f es una expresión que devuelve valores reales.

Si la función objetivo es irrelevante entonces el problema se reduce a encontrar un valor dentro del dominio. Es decir que cumpla $x \in \Omega$.

Muchos de estos problemas pueden ser enfocados mediante la técnica de Programación Dinámica. Aquí nos vamos a concentrar en la Programación Dinámica de tipo Reducción (es decir sólo existe un subproblema para cada alternativa). Los algoritmos de Vuelta Atrás tienen un enfoque equivalente.

Asumamos que el dominio es definido por un conjunto de restricciones de igualdad y desigualdad de la forma:

$$\begin{aligned} u_i(x) &\leq 0, \quad i = 1, \dots, r \\ v_j(x) &\geq 0, \quad j = 1, \dots, s \\ w_j(x) &= 0, \quad k = 1, \dots, t \end{aligned}$$

Por simplicidad asumimos que sólo hay una de cada una de ellas. El domino queda de la forma:

$$\Omega(x) = \begin{cases} u(x) \leq 0 \\ v(x) \geq 0 \\ w(x) = 0 \end{cases}$$

Búsqueda de las alternativas según el tipo de las soluciones (PD, BT, PLI, AG)

En los problemas de *Programación Dinámica de tipo Reducción* se puede definir la secuencia de problemas:

$$[p_0, p_1, \dots, p_i, \dots, p_n]$$

Donde p_0 es el problema inicial y p_n un caso base o un problema cuyo conjunto de alternativas es vacío. Todos los p_i son problemas generalizados del problema original en una forma que tendremos que determinar. Sea A_i el conjunto de alternativas del problema p_i (para $i: 0..n-1$) y $a_i \in A_i$ la alternativa que nos lleva del problema p_i al p_{i+1} (igualmente para $i: 0..n-1$). A cada problema p_i de la secuencia le podemos asociar la secuencia de alternativas que hemos tomado para llegar a él desde p_0. Sea esta secuencia la_i. Como podemos ver $la_0 = []$.

Sin pérdida de generalidad podemos asumir que A_i es un conjunto de números enteros que queremos caracterizar para cada problema.

Un punto clave es que la secuencia de problemas (siempre que p_n sea un caso base) define una solución para el problema inicial p_0. Por la misma razón la secuencia de alternativas la_n también

define una solución válida de p_0. Es decir existe una función que transforma la lista de alternativas en una solución x. Sea esta función de la forma:

$$x = \varphi(la_n) + x_b$$

Donde x_b es la solución del caso base y $+$ un operador adecuado al tipo de datos. Pensar la forma de esta función nos da muchas pistas para la definición del conjunto de alternativas. Otra cuestión a tener en cuenta es que para dada la_i, la secuencia de alternativas que nos lleva de p_0 a p_i, debemos determinar el conjunto de alternativas A_i que conducen a un problema p_{i+1} válido. Válido en el sentido que pueda tener solución. Siendo por lo tanto $la_{i+1} = la_i + a_i$.

En muchos casos el tipo de valores de la lista de alternativas (la lista la_n) nos puede guiar para escoger el cromosoma adecuado para resolver el problema mediante algoritmos genéticos.

Si las restricciones que definen el problema son lineales entonces el problema puede ser formulado mediante Programación Lineal Entera con las alternativas como variables del problema.

Búsqueda de las propiedades individuales del problema (o del estado si estamos en BT) guiada por las restricciones del problema

Dadas las ideas anteriores y la forma de las restricciones proponemos una forma de generalizar el problema original p a otro generalizado. Cada uno de los p_i será uno de esto problemas generalizados. Pensamos una propiedad individual de los problemas individual del problema generalizado por cada una de las restricciones y de la función objetivo más un índice que define el problema dentro de la secuencia. Tenemos entonces:

$$p_i = \begin{cases} i \\ la_i \\ fa_i \\ va_i \\ ua_i \\ wa_i \end{cases}$$

Los acumuladores anteriores deben ser adecuados para evaluar las restricciones en el caso base, poder filtrar las alternativas que conducen a problemas no válidos y aquellas otras que, en el caso de optimización conduzcan a un problema pero que el mejor ya encontrado. Por otra parte los acumuladores pueden considerarse como propiedades derivadas de la_i. También definimos los incrementos de esos acumuladores y propiedades para un problema p_i que representaremos por Δp_i:

$$\Delta p_i = \begin{cases} \Delta i = i + 1 - i = 1 \\ \Delta la_i = la_{i+1} - la_i = a_i \\ \Delta fa_i = fa_{i+1} - fa_i \\ \Delta va_i = va_{i+1} - va_i \\ \Delta ua_i = ua_{i+1} - ua_i \\ \Delta wa_i = ua_{i+1} - ua_i \end{cases}$$

Normalmente estos incrementos dependen de las propiedades de p_i y la alternativa a_i tomada para pasar a p_{i+1}.

Debe cumplirse:

$$\begin{cases} fa_n = f(\varphi(la_n)) + f_b \\ ua_n = u(\varphi(la_n)) + u_b \\ va_n = v(\varphi(la_n)) + v_b \\ wa_n = w(\varphi(la_n)) + w_b \end{cases}$$

Donde f_b, u_b, v_b, w_b son los valores correspondientes de las funciones $f(x), v(x), u(x), w(x)$ para el caso base. De la misma forma fa_n, ua_n, va_n, wa_n son los valores correspondientes de esas funciones para el problema inicial p_0.

En cada problema en particular habrá que concretar la forma de calcular las propiedades de p_{i+1} ($i+1,\ fa_{i+1}, ua_{i+1}, va_{i+1}, wa_{i+1}$) a partir de $i, fa_i, ua_i,\ va_i,\ wa_i$ asumiendo que se toma la alternativa $a_i \in A_i$. Este cálculo vendrá dado por la forma de las funciones $f(x), v(x), u(x), w(x)$. Para que esto sea posible A_i no puede ser vacío.

Con todos estos elementos podemos definir el conjunto de alternativas A_i para cada problema p_i de la forma:

$$A_i = \{a: 0..m_i |\ ua_{i+1} \leq 0\ \wedge va_{i+1} \geq 0\ \}$$

Donde hemos asumido que las alternativas son número enteros en el intervalo $0..m_i$ siendo m_i valores positivos a determinar. Obsérvese que aquí solo intervienen los acumuladores asociados a las desigualdades pero no a las igualdades.

Cuando usamos la técnica de filtro (ya sea en PD o BT) entonces el conjunto A_i puede ser reducido aún más usando la función de cota $ct(a_i)$ y el mejor valor del objetivo encontrado hasta ese momento f_m. Si el problema original fuera de maximización la función de cota debe ser una cota superior la propiedad objetivo del problema p_i asumiendo que se va a seguir la alternativa a_i hasta el caso base. Este cálculo se hará normalmente siguiendo una estrategia voraz que tome como primer paso la alternativa a_i y posteriormente en cada problema las adecuadas para conseguir la cota superior. Con esos elementos el conjunto A_i queda:

$$A'_i = \{a_i: 0..m_i |\ ua_{i+1} \leq 0\ \wedge va_{i+1} \geq 0\ \wedge\ \ fa_i + ct(a_i)\ >\ f_m\}$$

Si el problema fuera de minimización entonces la función $ct(a_i)$ debe ser una cota inferior y el conjunto A'_i es de la forma:

$$A'_i = \{a_i: 0..m_i |\ ua_{i+1} \leq 0\ \wedge va_{i+1} \geq 0\ \wedge\ \ fa_i + ct(a_i) <\ f_m\}$$

La solución parcial del caso base con la notación anterior es de la forma:

$$\begin{cases} f_b, & existe\ w_b\ tal\ que\ w(\varphi(la_n)) + w_b = 0 \\ \perp, & no\ existe\ w_b\ tal\ que\ w(\varphi(la_n)) + w_b = 0 \end{cases}$$

El decir no hay solución para el caso base (y por lo tanto para el problema inicial siguiendo ese camino) si no podemos encontrar un valor que satisfaga la restricción de igualdad.

El enfoque anterior puede complementarse (o sustituirse) en algunos casos con el planteamiento de un invariante. Si en la secuencia de problemas estamos en la posición i, hemos tomado previamente las alternativas

$$la_i = [a_0, a_1, \dots, a_{i-1}]$$

Las propiedades del problema p_i son:

$$p_i = \begin{cases} i \\ la_i \\ fa_i \\ va_i \\ ua_i \\ wa_i \end{cases}$$

El invariante relaciona la secuencia de alternativas tomadas con las propiedades del problema actual, posiblemente con algunas restricciones adicionales.

Con el esquema anterior podemos concretar el detalle de los métodos a implementar en el caso de PD con reducción y BT siguiendo los pasos siguientes:

- Definir el tipo de las soluciones del problema
- *Alternativas:* El primer paso es pensar cuales pueden ser las alternativas posibles pensando en cómo transformar la secuencia de alternativas en una solución del problema. Es decir pensar en la transformación:

  $$x = \varphi(la_n) + x_b$$

- *Filtrar las alternativas:* Generalizar el problema original con propiedades escogidas a partir de las restricciones y de la función objetivo. A partir de eso definir

  $$A_i = \{a{:}\, 0..m_i |\, ua_{i+1} \leq 0 \ \wedge va_{i+1} \geq 0 \,\}$$

  Para ello es necesario definir un rango para los valores de las alternativas. Es decir los valores m_i.

- *EsCasoBase*: Debemos buscar en un primer momento un caso base lo más sencillo posible. Sencillo desde el punto de vista que la solución del caso base sea lo más simple posible. Posteriormente se podrían añadir otros casos base. Una guía para un caso base sencillo es buscar que la transformación de la lista de alternativas a soluciones pueda ser de la forma:

  $$x = \varphi(la_n) + e$$

  Donde e se el elemento neutro de la correspondiente operación $+$ (0, conjunto vacío, *multiset* vacío, lista vacía, etc.).

  En el caso de que el problema tenga una propiedad compartida que sea una lista ls de objetos que hay que recorrer entonces un caso base que cumple estas características es $i = ls.size()$.

Con BT usamos las mismas ideas para buscar el problema final.

- *Solución del caso base:*

  La solución parcial del caso base, si existe, será de la forma (a, f_b). Donde en muchos casos la alternativa a es irrelevante y en ese caso pondremos _ en su lugar. También para el caso base tenemos que calcular los valores asociados a las propiedades individuales del problema ligadas a las restricciones. Es decir u_b, v, w_b. Con la notación anterior es de la forma:

$$\begin{cases} (a, f_b), & existe\ w_b\ tal\ que\ w(\varphi(la_n)) + w_b = 0 \\ \perp, & no\ existe\ w_b\ tal\ que\ w(\varphi(la_n)) + w_b = 0 \end{cases}$$

  Es decir para que haya solución del caso base debe cumplirse la restricción de igualdad asociada al problema.

- *Subproblema:* Consiste en pasar de las propiedades del problema i a las del $i+1$.

$$p_i = (i, la_i, fa_i, ua_i, va_i, wa_i) \xrightarrow{a_i} p_{i+1} = (i+1, la_{i+1}, fa_{i+1}, ua_{i+1}, va_{i+1}, wa_{i+1})$$

  Donde las propiedades derivadas $(fa_{i+1}, ua_{i+1}, va_{i+1}, wa_{i+1})$ se pueden calcular de forma incremental a partir de (fa_i, ua_i, va_i, wa_i) y el valor de la alternativa a_i.

  De forma equivalente podemos diseñar el método *avanza(a)* que usaremos en *BT* o su inverso el método *retrocede(a)*.

- *CombinaSolucionesParciales:* Es el mecanismo para calcular la solución parcial de un problema p_i a partir de la solución del siguiente problema alcanzado p_{i+1} siguiendo la alternativa a_i. Si la solución parcial de p_{i+1} es (a', v') la de p_i será dada por $cs(a_i, (a', v') = (a_i, v)$ siendo v el valor de la función objetivo a determinar que se calculará en función de v', a_i y algunas propiedades adicionales de p_i.
- *SeleccionaAlternativa:* Es el mecanismo para elegir una de las distintas soluciones parciales no nulas calculadas según las diferentes alternativas. Lo solemos expresar como $sA((a', v'), (a'', v''), \dots)$. Usualmente se escoge la solución óptima (mejor o peor). Otros mecanismos para problemas que no son de optimización se explican en lo apuntes.
- *getObjetivo():* Es el valor de la función objetivo del problema inicial evaluado en el caso base. Es por tanto el valor f_n.
- *getObjetivoEstimado(a):* Es el valor de la función objetivo del problema inicial evaluado en un caso intermedio p_i. Con la notación anterior es el valor $fa_i + ct(a_i)$.

Como hemos dicho antes la función de cota $ct(a_i)$ pretende encontrar una cota superior (o inferior dependiendo de si el problema es de maximización o minimización) de la función objetivo para el problema actual p_i asumiendo que se escoge a_i para pasar al siguiente problema p_{i+1} y en adelante tomando la alternativa, de entre las disponibles, que consiga la mejor cota superior para pasar de cada problema al siguiente hasta alcanzar un caso base. La función de cota es, por lo tanto, un algoritmo voraz.

- *Reconstruye solución:* Busca la solución a partir de una función que asociada cada problema p_i con la solución parcial encontrada para él (a_i, v_i). Esta asociación se mantiene en un *Map<Problema,SolucionParcial>.* Asumiendo que s_{i+1} es la solución calculada del problema p_{i+1}, y haciendo $s_n = s_b$ la solución del caso base, podemos plantear una solución recursiva al cálculo de la solución:

$$sr\big(p_i, (a_i, v_i)\big) = s_b, \qquad p_i \; es \; un \; caso \; base$$

$$sr(p_i, (a_i, v_i), s_{i+1}) = (a_i, v_i) \oplus_i s_{i+1}, \qquad p_i \; no \; es \; un \; caso \; base$$

Aquí el operador $\oplus_i$ calcula la solución reconstruida de un problema a partir de la solución del hijo, la solución parcial del padre y posiblemente otras propiedades del problema p_i. En el caso de *BT* no existe el concepto de solución parcial. La solución hay que calcularla (reconstruirla) en el caso base a partir de la secuencia de alternativas que hemos tomado para llegar al caso base en la forma:

$$x = \varphi(la_n) + x_b$$

Este cálculo puede hacerse iterativo o recursivo.

Como dijimos arriba si las restricciones que definen el problema son lineales entonces el problema puede ser formulado mediante Programación Lineal Entera con las alternativas como variables del problema.

En el caso de querer resolver el problema mediante algoritmos genéticos en algunos casos podemos guiarnos por la lista de alternativas para escoger el cromosoma. Es decir para que la lista de alternativas sea equivalente a la devuelta por el método *decode()* del cromosoma escogido. De esa forma podemos definir la función de *fitness* usando la notación anterior en la forma:

$$ft = fa_n - K(le(ua_n) + ge(va_n) + eq(wa_n))$$

$$le(x) = \begin{cases} 0, & x \leq 0 \\ x * x, & x > 0 \end{cases}$$

$$ge(x) = \begin{cases} x * x, & x < 0 \\ 0, & x \geq 0 \end{cases}$$

$$eq(x) = x * x$$

Las funciones anteriores se suponen definidas sobre números reales. Si su dominio son enteros hay que hacerles algunos pequeños ajustes. Y recordando que las restricciones del dominio son de la forma:

$$\Omega(x) = \begin{cases} u(x) \leq 0 \\ v(x) \geq 0 \\ w(x) = 0 \end{cases}$$

Los valores fa_n, ua_n, va_n, wa_n se calculan siguiendo ideas similares a las usadas en PD. Las restricciones de la forma específica siguiente son denominadas restricciones de rango:

$$0 \leq a_i \leq m_i, \;\; i{:}\, 0..n-1$$

Se utilizan para filtrar las alternativas o para escoger adecuadamente el cromosoma.

Tipos de problemas ejemplo

Vamos a considerar varias clases de problemas según el tipo de restricciones de los mismos. En todos ellos aparecen listas de objetos a partir de los cuales debemos encontrar soluciones en forma de conjuntos, multiconjuntos, maps o listas. La forma de las restricciones, como hemos explicado arriba nos orientará para diseñar las alternativas, la solución del caso base, etc.

Problemas tipo Mochila

El problema se formaliza así:

Dada una lista ls de objetos de tamaño n. Cada objeto e_i de tipo E tiene varias propiedades $e_i = (v_i, p_i, m_i)$ se pretende:

$$\max \sum_{i=0}^{n-1} a_i v_i$$

$$0 \leq a_i \leq m_i, \;\; i{:}\, 0..n-1$$

$$\sum_{i=0}^{n-1} a_i c_i \leq C$$

Dónde (v_i, c_i, m_i) podrían ser el valor unitario, el peso unitario y el número máximo de unidades disponibles. C es la capacidad de recipiente dónde debemos ubicar las unidades escogidas de cada objeto.

Las soluciones del problema son *Multiset<E>*.

Reescribimos la restricción de desigualdad sobre la capacidad en la forma:

$$C - \sum_{i=0}^{n-1} a_i c_i \geq 0$$

Las alternativas a_i representan el número de unidades escogida de cada elemento e_i. Cada problema puede ser representado por las propiedades:

$$p_i = \begin{cases} i \\ la_i \\ fa_i = \sum_{j=0}^{i-1} a_j v_j \\ ca_i = C - \sum_{j=0}^{i-1} a_j c_j \end{cases}$$

El problema p_i busca escoger a_i unidades de cada elemento $e_j, j: i..n-1$ asumiendo que la capacidad disponible es ua_i y el valor acumulado por las unidades ya escogidas de los elementos $e_j, j: i..i-1$ es fa_i.

Con los elementos anteriores podemos diseñar las alternativas en la forma:

$$A_i = \{a: 0..m_i |\ ca_{i+1} \geq 0\ \}$$

El caso base más simple es $i = n$ y su solución parcial (_,0) dónde la alternativa es irrelevante. Combina soluciones es de la forma $cs(a_i, (a', v') = (a_i, v' + a_i v_i)$.

Como hemos visto antes *getObjetivoEstimado(a) e*s el valor de la función objetivo del problema inicial evaluado en un caso intermedio p_i. Es decir $fa_i + ct(a_i)$. Dónde $ct(a_i)$ es la función de cota.

La función de cota (ahora una cota superior) se calcula mediante un algoritmo voraz donde la primera alternativa ya nos viene fijada. Por eso la función de cota la expresamos dependiendo de un parámetro $ct(a_i)$. La función de cota usa, además, las propiedades del problema p_i.

El algoritmo voraz tiene la estructura de un bucle *while*. Comienza en el problema i escoge la mejor alternativa posible (salvo en el primer caso que la alternativa ya viene en el parámetro), calcula los acumuladores del problema según esa alternativa y pasa al problema siguiente y así hasta que llega al caso base (en este caso $i = n$). Asumimos en esta estrategia voraz que la lista ls está ordenada por la propiedad v_i/c_i de mayor a menor. Es decir por el valor unitario. Decidimos escoger en cada paso la alternativa $\overline{a_j} = \min(m_j, \frac{ca_j}{c_j})$. Como vemos $\overline{a_j}$ sería la estrategia voraz consistente en escoger en cada paso un número entero con el mayor número de unidades posibles y pasar al siguiente problema. De las definiciones anteriores podemos ver que $ca_{i+1} = ca_i - a_i c_i$, $fa_{i+1} = fa_i + a_i v_i$. Representaremos ca_i por cr. El algoritmo voraz es por lo tanto:

```
double ct(int a, int i, int cr){
	double r =0;
	cr = cr-a* c(j);
	r =  r + a* v(j);
	while(i<n){
		a = min(m(j),cr/c(j));
		cr = cr-a* c(j);
		r =  r + a* v(j);
	}
	return r;
}
```

Otra estrategia voraz aún mejor es hacer la división $\frac{ca_j}{c_j}$ entre número reales. Es decir la estrategia voraz consiste en escoger en cada paso el mejor número de unidades posible aunque fuera un número real y no entero. Es decir escoger partes de una unidad. Es cambio es:

```
double ct(int a, int i, int cr){
       double r =0;
       double b = a;
       double dr;
       dr = cr-b* c(j);
       r =  r + b* v(j);
       while(i<n){
              b = min(m(j),dr/c(j));
              dr = dr-b* c(j);
              r =  r + b* v(j);
       }
       return r;
}
```

Como vemos las alternativas escogidas en el segundo caso son números reales y la capacidad restante también es un número real.

La función *getObjetivo* devolvería, puesto que siempre hay solución, el valor acumulado fa_n.

La solución reconstruida del problema dada la lista de alternativas es:

$$m = \bigcup_{i=0}^{n-1} a_i \times e_i$$

Es decir un multiconjuto formado por a_i unidades de cada elemento e_i. La solución recursiva es:

$$sr\big((a_i, v_i)\big) = \{\},\ p_i\ es\ un\ caso\ base(i == n)$$

$$sr((a_i, v_i), m_{i+1}) = a_i \times e_i + s_{i+1}, \qquad p_i\ no\ es\ un\ caso\ base$$

Donde p_i aparece de forma implícita.

Las ecuaciones que definen el problema son lineales por lo que podemos formularlo directamente como un problema de Programación Lineal Entera con variables enteras.

Para resolver el problema mediante Algoritmos Genéticos podemos escoger un cromosoma de rango (*IndexRangeChromosome*) donde los límites superiores de los valores vienen definidos por las restricciones de rango y el tamaño n. La función de *fitness* viene definida por:

$$ft = fa_n - K(ge(va_n))$$

$$fa_n = \sum_{i=0}^{n-1} a_i v_i$$

$$va_n = C - \sum_{i=0}^{n-1} a_i c_i$$

Problemas tipo cambio de monedas

El problema se formaliza así:

Dada una lista ls de objetos de tamaño n. Podría ser una lista de billetes disponibles en un sistema monetario. Cada objeto e_i de tipo E tiene varias propiedades $e_i = (v_i)$ se pretende:

$$\min \sum_{i=0}^{n-1} a_i$$

$$\sum_{i=0}^{n-1} a_i v_i = C$$

Dónde (v_i) podría ser el valor de ese billete. C la cantidad a cambiar en distintas unidades de billetes e_i. Las soluciones del problema son *Multiset<E>*.

Siguiendo los pasos del caso anterior las alternativas a_i representan el número de unidades escogida de cada elemento e_i y las propiedades individuales del problema serán:

$$p_i = \begin{cases} i \\ la_i \\ fa_i = \sum_{j=0}^{i-1} a_i \\ wa_i = C - \sum_{j=0}^{i-1} a_i v_i \end{cases}$$

El problema p_i busca escoger a_i unidades de cada elemento $e_j, j: i..n-1$ asumiendo que la cantidad disponible es wa_i y el número acumulado por las unidades ya escogidas de los elementos $e_j, j: i..i-1$ es fa_i.

La restricción de igualdad debe cumplirse en el problema p_n (es decir $wa_i = 0$) para el resto de problemas podemos relajarla en la forma $wa_i \geq 0$.

Usando las ideas del problema anterior podemos diseñar las alternativas en la forma:

$$A_i = \{a_i: 0.. \mid wa_{i+1} \geq 0\}$$

Como podemos ver que $wa_{i+1} = wa_i - a_i v_i \geq 0$, $wa_i \geq 0$. Luego $wa_i \geq a_i v_i$ y por lo tanto los valores de $a_i \leq wa_i / v_i$. Las alternativas se pueden escribir en la forma:

$$A_i = \{a_i: 0..k\}$$

$$k = wa_i / v_i$$

Por lo tanto k es el mayor número de unidades posibles del tipo e_i. El caso base más simple es $i = n$ y su solución parcial, si hay solución, (_,0), dónde la alternativa es irrelevante. Habrá solución si se cumple $wa_i = 0$. Si no se cumple no la habrá. Las restricciones de igualdad hay que tenerlas en cuenta justo en este momento.

La conclusión que obtenemos es que las restricciones de igualdad deben ser comprobadas en los casos base y en el método *getObjetivo* (o *getSolución* en BT) para decidir si este tiene solución o no. Por otra parte una versión relajada en forma de desigualdad nos sirve para filtrar las posibles alternativas. Las restricciones de desigualdad sólo se usan para filtrar las posibles alternativas.

Combina soluciones es de la forma $cs(a_i, (a', n') = (a_i, n' + a_i)$.

Como hemos visto antes *getObjetivoEstimado(a) e*s el valor de la función objetivo del problema inicial evaluado en un caso intermedio p_i. Es decir $fa_i + ct(a_i)$. Dónde $ct(a_i)$ es la función de cota.

La función de cota (ahora una cota inferior) podemos expresarla como un algoritmo voraz que siempre escoge la alternativa 0.

$$ct(a) = a + \sum_{j=i+1}^{n-1} 0 = av_i$$

Asumiendo la lista ls ordenada de mayor a menor valor unitario otra estrategia voraz posible es escoger el máximo número de unidades de las monedas con mayor valor unitario.

$$ct(a) = a + \sum_{j=i+1}^{n-1} \overline{a_j}$$

Como antes asumimos, en esta estrategia voraz, que estando en el problema p_i escogemos la alternativa $\overline{a_j}$ y pasamos al problema p_{i+1} y que la lista ls está ordenada por la propiedad v_i de mayor a menor. Es decir por el valor unitario. Ahora $\overline{a_j} = \frac{wa_j}{v_j}$. Como vemos $\overline{a_j}$ sería la estrategia voraz consistente en escoger en cada paso el mayor número de unidades posibles y pasar al siguiente problema. En el último paso $j = n - 1$ escogeríamos $\overline{a_{n-1}} = 0$ si la división no fuera exacta. Esto nos permitirá acumular la cantidad (o una cercana menor) con el menor número de billetes.

La función *getObjetivo* devolvería el valor acumulado fa_n si el caso base tiene solución. Si no la tiene entonces MAX_VALUE.

La solución es decir un *multiconjunto* formado por a_i unidades de cada elemento e_i que se calcula como en el ejemplo anterior.

Las ecuaciones que definen el problema son lineales por lo que podemos formularlo directamente como un problema de Programación Lineal Entera con variables enteras.

Para resolver el problema mediante Algoritmos Genéticos podemos escoger un cromosoma de rango (*IndexRangeChromosome*) donde los límites superiores de los valores vienen definidos por las restricciones de rango y el tamaño n. La función de *fitness* viene definida por:

$$ft = fa_n - K(eq(va_n))$$

$$fa_n = \sum_{i=0}^{n-1} a_i$$

$$wa_n = C - \sum_{i=0}^{n-1} a_i v_i$$

Problemas tipo: Escoger con incompatibilidades

El problema pretende recoger diferentes tipos de incompatibilidades entre los objetos a elegir y se formaliza así:

Dada una lista ls de objetos de tamaño n. Cada objeto e_i de tipo E tiene varias propiedades $e_i = (v_i, p_i)$ que representan su valor unitario y su peso unitario. Por alguna razón existe una lista de incompatibilidades de pares de objetos para ser elegidos. Esas incompatibilidades la recogemos en una lista de conjuntos binarios lb (de tipo *List<Set<E>>*) de tamaño m. Cada conjunto de esta lista representa un par de objetos incompatibles. Adicionalmente podemos contar con un conjunto de objetos $s1$ de los incluidos en la lista ls de entre los cuales se pueden escoger como máximo $n1$ de ellos y otro $s2$ entre los que tenemos que elegir exactamente $n2$. Los conjuntos $s1,\ s2$ son de tipo *Set<E>.* Esta información la podemos representar, también, mediante conjuntos de índices li (de tipo *List<Set<Integer>>*), $r1, r2$ de tipos *(Set<Integer).*

Se pretende:

$$\max \sum_{i=0}^{n-1} a_i\, v_i$$

$$\sum_{i=0}^{n-1} a_i p_i \leq C$$

$$\sum_{i \in li[j]} a_i \leq 1, \qquad j: 0..m-1$$

$$\sum_{i \in r1} a_i \leq n1$$

$$\sum_{i \in r2} a_i = n2$$

En la descripción anterior ln es una lista de enteros de tamaño m cuyo valor indica el número de objetos de $li[j]$ que escogemos. Por otra parte $ns1,\ ns2$ son el número de objetos que escogemos de los conjuntos $s1,\ s2$. Las alternativas a_i son valores enteros binarios que indican si se escoge o no el objeto e_i. Las soluciones del problema son S*et<E>.*

Con esas ideas el problema generalizado podemos dotarlo de las siguientes propiedades:

$$p_i = \begin{cases} i \\ la_i \\ fa_i = \sum_{j=0}^{i-1} a_i\, v_i \\ pa_i = C - \sum_{j=0}^{i-1} a_i p_i \\ aln_i[j] = \sum_{k:0..i-1 | k \in li[j]} a_k\,, \qquad j: 0..m-1 \\ as1_i = \sum_{k:0..i-1 | k \in r1} a_k \\ as2_i = \sum_{k:0..i-1 | k \in r2} a_k \end{cases}$$

Las alternativas pueden definirse de la forma:

$$A_i = \{a_i: 0..1 |\, pa_{i+1} \geq 0, as1_{i+1} \leq n1, as2_{i+1} \leq n2, \bigvee_{j=0}^{m-1} aln_{i+1}[j] \leq 1\}$$

Donde los nuevos valores de los acumuladores se calculan de la forma siguiente:

$$\begin{cases} i+1 \\ la_{i+1} = la_i + a_i \\ fa_{i+1} = fa_i + a_i v_i \\ pa_{i+1} = pa_i - a_i p_i \\ aln_{i+1}[j] = aln_i[j] + df[j], \qquad j: 0..m-1 \\ as1_{i+1} = as1_i + c1(a_i, i) \\ as2_{i+1} = as2_i + c2(a_i, i) \end{cases}$$

$$c1(a_i, i) = \begin{cases} 0, & a_i = 0 \\ 1, & a_i = 1, e_1 \in s1 \\ 0, & a_i = 1, e_1 \notin s1 \end{cases}$$

$$c2(a_i, i) = \begin{cases} 0, & a_i = 0 \\ 1, & a_i = 1, e_i \in s2 \\ 0, & a_i = 1, e_1 \notin s2 \end{cases}$$

$$df[j] = \begin{cases} 0, & a_i = 0 \\ 1, & a_i = 1, e_i \in li[j] \\ 0, & a_i = 1, e_1 \notin li[j] \end{cases}$$

El caso base más simple es $i = n$ y su solución parcial, si hay solución, (_,0), dónde la alternativa es irrelevante. Habrá solución si se cumple $as2_i = 0$. Si no se cumple no la habrá. Son solo las restricciones de igualdad las que hemos tenido en cuenta para decidir si el caso base tiene solución.

Igualmente ocurre, como hemos visto arriba, en el método *getObjetivo* (o *getSolución* en BT) para decidir si este tiene solución o no. Por otra parte una versión relajada, de la restricción de

igualdad, en forma de desigualdad nos sirve para filtrar las posibles alternativas. Las restricciones de desigualdad siempre se usan para filtrar las posibles alternativas.

Combina soluciones es de la forma $cs(a_i, (a', n') = (a_i, n' + a_i v_i)$.

La solución es decir un *conjunto* formado por los elementos cuyas $a_i = 1$. La solución reconstruida es un conjunto formado por cada elemento e_i tal que $a_i = 1$ siempre dentro de la solución óptima encontrada por el algoritmo:

$$sr\big((a_i, v_i)\big) = \{\},\ p_i\ es\ un\ caso\ base(i == n)$$

$$sr((a_i, v_i), s_{i+1}) = \begin{cases} e_i + s_{i+1}, & a_i = 1 \\ s_{i+1}, & a_i = 0 \end{cases}$$

Las ecuaciones que definen el problema son lineales por lo que podemos formularlo directamente como un problema de *Programación Lineal Entera* con variables binarias.

Para resolver el problema mediante Algoritmos Genéticos podemos escoger un cromosoma binario (Binary*Chromosome2*) de tamaño n. La función de *fitness* viene definida por:

$$ft = fa_n - K(ge(pa_n) + le(as1_{i+1} - n1) + eq(as2_{i+1} - n2) + \sum_{j=0}^{m-1} le(aln_n[j] - 1))$$

El problema de la asignación y sus variantes

El problema parte de una lista de agentes lg y una lista de tareas lt ambas del mismo tamaño n. El coste de que el agente i realice la tarea j es $c(i, j)$. A cada agente se le asigna una sola tarea y cada tarea debe ser llevada a cabo por un solo agente.

La solución del problema será de tipo $Map < G, T >$ dónde G es el tipo del agente y T el tipo de la tarea. La solución también se puede representar por una lista $[a_0, a_1, \dots, a_{n-1}]$ de enteros en el rango $0..n-1$. Con la lista indicamos que el agente i realizará la tarea a_i. Estas serán las alternativas de este problema.

Se pretende:

$$\min \sum_{i=0}^{n-1} c(i, a_i)$$

$$0 \le a_i \le n-1, \quad i: 0..n-1$$

$$|toSet([a_0, a_1, \dots, a_{n-1}])| = n$$

La última restricción indica que no existen alternativas repetidas.

De nuevo vamos a pensar en un problema generalizado con un conjunto de acumuladores sugeridos por las restricciones del problema.

$$p_i = \begin{cases} i \\ la_i \\ fa_i = \sum_{j=0}^{i-1} c(j, a_j) \\ ar_i = toSet([a_0, a_1, \dots, a_{n-1}] - la_i) \end{cases}$$

Donde ar_i es el conjunto de las alternativas que todavía no se han escogido. El conjunto de alternativas es de la forma:

$$A_i = \{a_i : 0..n-1 |\ a_i \in ar_i\}$$

El caso base más simple es $i = n$ y su solución parcial (_,0).

Combina soluciones es de la forma $cs(a_i, (a', n') = (a_i, n' + c(i, a_i))$.

Como hemos visto antes *getObjetivoEstimado(a) e*s de la forma $fa_i + ct(a_i)$. Dónde $ct(a_i)$ es la función de cota.

La función de cota (ahora una cota inferior) podemos expresarla como un algoritmo voraz que asume que el coste de las asignaciones restantes es 0.

$$ct(a) = c(i, a) + \sum_{j=i+1}^{n-1} 0 = c(i, a)$$

Otra estrategia voraz posible es asumir que el coste de cada una de las asignaciones restantes es igual al mínimo de todas las posibles que designaremos como cmr.

$$ct(a) = c(i, a) + (n - i - 1)cmr$$

La función *getObjetivo* devolvería el valor acumulado fa_n.

La solución es decir un $Map < G, T >$ que se reconstruye de la siguiente forma:

$$sr\big((a_i, v_i)\big) = \{\},\ p_i\ es\ un\ caso\ base(i == n)$$

$$sr\big((a_i, v_i), s\big) = s + (i, a_i)$$

La restricción siguiente no es lineal:

$$|toSet([a_0, a_1, \dots, a_{n-1}])| = n$$

Tampoco es lineal la función $c(j, a_j)$ que implicaría depender de algún tipo de datos que dados los índices variables (j, k) devolviera el valor de $c(j, k)$.

Por ello el problema no se puede resolver tal como está formulado mediante Programación Lineal Entera. Para poder hacerlo hay que expresarla, si es posible, mediante restricciones lineales. Una forma de hacerlo es escoger las variables binarias x_{ij} que indican que al agente i se le asigna la tarea j con las restricciones:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\, x_{ij}, \quad i \in [0, n-1],\ j \in [0, n-1]$$

Ahora las cantidades c_{ij} son constantes conocidas que pueden ser escritas en las restricciones.

Para resolver el problema mediante Algoritmos Genéticos podemos escoger un cromosoma que genera todas las permutaciones posibles (*IndexPermutationChromosome*) de tamaño n. La función de *fitness* viene definida simplemente por:

$$ft = fa_n$$

El problema de tareas y procesadores: Relación entre PDR y BT

El problema se formula de la siguiente manera: Dado una lista lt de n tareas con duraciones d_i y un conjunto de m procesadores buscar la asignación de tareas a procesadores tal que el tiempo total de ejecución sea mínimo.

Las soluciones del problema pueden se codificadas por listas de alternativas la de tamaño n y donde la alternativa a_i, toma un valor de $0..m-1$, indica que la tarea i se asigna al procesador a_i.

Denotaremos por $t = [t_0, \ldots, t_{m-1}]$. Siendo $t[j]$ los tiempos acumulados en cada procesador debido a las tareas asignadas. Es decir:

$$t[j] = \sum_{i:0..n-1 | a_i = j} d_i\,, \quad j: 0..m-1$$

La notación escogida es un sumatorio con un filtro. El enunciado del problema es entonces:

$$\min T$$

$$T = \max_{j:0..m-1} t[j]$$

$$t[j] = \sum_{i:0..n-1 | a_i = j} d_i\,, \quad j: 0..m-1$$

$$0 \le a_i \le n-1, \quad i: 0..n-1$$

El problema generalizado es de la forma:

$$p_i = \begin{cases} i \\ la_i \\ T_i = \max\limits_{j:0..m-1} t_i[j] \\ t_i[j] = \sum\limits_{k:0..i-1 | a_k=j} d_k\,, \quad j: 0..m-1 \end{cases}$$

El conjunto de alternativas es de la forma:

$$A_i = \{a_i: 0..n-1\}$$

El caso base más simple es $i = n$. Si usamos la técnica de *BT* entonces el método *getSolución()* proporciona una solución con la información acumulada (T_n, t_n, la_n).

En algunos casos podemos diseñar un algoritmo de PDR que aproveche la solución acumulada en el caso base, como en al algoritmo de BT, y no volver a calcularla en la fase ascendente de *PDR*. En *PDR (Programación Dinámica con Reducción)* cada solución del problema inicial está asociada a una rama del grafo de problemas que va del problema inicial a un caso base. La mejor de esas soluciones tendrá una rama asociada y definirá la solución buscada del problema inicial. Para cada problema concreto p_i vamos a definir como solución la mejor solución del problema inicial de entre las ramas que pasen por p_i. Así cada problema tiene en cuenta la carga previa de los procesadores por las tareas ya asignadas. Esta definición de solución vale para los casos base y para el problema original. Además hemos de tener en cuenta, con la definición anterior, que cuando se pasa de un problema p_i a otro p_{i+1} siguiendo la alternativa a_i la solución de p_{i+1} es una solución de p_i (puede que no la mejor) por estar ambos en la misma rama. Con estas ideas podemos definir la solución del caso base y el método combina soluciones.

El caso base más simple es $i = n$. El caso base escogido no añade ninguna tarea adicional. Con la definición de solución anterior la solución parcial del caso base es *(-1,t).* Igualmente el método combina soluciones es de la forma $cs(a_i, (a', t') = (a_i, t')$. Es decir una de las soluciones de p_i es la proporcionada por p_{i+1}. El método selecciona alternativa se encargará de escoger la rama con el valor óptimo y por lo tanto la solución de p_i.

Las ideas anteriores nos permiten reformular un algoritmo de *BT* como otro de *PDR* con las ventajas que eso pueda suponer. Una de las ventajas de usar *PDR* puede ser el uso de memoria. Las técnicas de filtro se pueden usar indistinta en ambos tipos de algoritmos tal como os tenemos implementados.

La solución es decir *Map<Integer, List<Tarea>>* que se puede reconstruir a partir de la secuencia de alternativas la_n de la rama óptima si estamos usando *BT* o de forma recursiva con el método *getSoluciónReconstruida*, si estamos usando *PDR* (asumiendo que estamos en el problema p_i)

getSolucionReconstruida((_f))= {}
getSolucionReconstruida((a,f),m)= m[a]+(i)

La segunda línea indica que si la solución del problema hijo es el *Map m*, entonces la solución de padre se construye añadiendo la tarea i a la lista de tareas, *m[a]*, del procesador *a*.

La solución del problema mediante algoritmos genéticos puede encontrarse escogiendo un cromosoma de rango (*IndexRangeChromosome*) de tamaño n y un rango definido para cada una de las posiciones definido por $0 \leq r_i \leq n-1, \quad i: 0..n-1$.

La función de *fitness* viene definida por:

$$ft = -T_n = -(\max_{j:0..m-1} t_j)$$

$$t_j = \sum_{k:0..n-1 | a_k = j} d_i\,, \quad j: 0..m-1$$

Las restricciones en la formulación del problema no son lineales. Para resolver el problema mediante Programación Lineal Entera es necesario otro enfoque del problema. Esto puede hacerse escogiendo las variables binarias x_{ij} que toman valor 1 si la tarea i es asignada al procesador j el problema puede ser formulado como:

$$\min \ T$$

$$\sum_{i=0}^{n-1} d_i x_{ij} \leq T, \quad j = 0, \dots, m-1$$

$$\sum_{j=0}^{m-1} x_{ij} = 1, \quad i = 0, \dots, n-1$$

$$bin \ x_{ij}, \quad i = 0, \dots, n-1, \quad j = 0, \dots, m-1$$

Buscar una solución, todas las soluciones, contar el número de soluciones: El problema de la reinas

El problema consiste en colocar n reinas en un tablero de ajedrez $n \times n$ de tal manera que ninguna de ellas amenace a ninguna de las demás. Una reina amenaza a los cuadrados de la misma fila, de la misma columna y de las mismas diagonales. Las filas y columnas toman valores en $0..n-1$.

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas. Aquí pretendemos contar el número de ellas.

El problema puede ser modelado usando las variables (alternativas) enteras $a_i, \quad i: [0, n-1]$ y las restricciones:

$$|toSet(a_{0,}\dots,a_{n-1})| = n$$
$$|toSet(a_{0,}a_1 - 1,\dots,a_{n-1} - (n-1))| = n$$
$$|toSet(a_{0,}x_1 + 1,\dots,a_{n-1} + n - 1)| = n$$
$$0 \le a_i < n, \quad i{:}\,[0, n-1]$$
$$int\ \ a_i, \quad i{:}\,[0, n-1]$$

Con $(a_{0,}\dots,a_{n-1})$ representamos una lista de alternativas. Con $toSet(a_{0,}\dots,a_{n-1})$ la conversión a conjunto de esa lista y con || el operador que calcula el tamaño *(.size())* de una lista o un conjunto. La alternativa a_i representa colocar una reina en la casilla (i, a_i). La primera restricción indica que las ys de las reinas colocadas tienen que ser todas diferentes. La segunda restricción indica que las diagonales principales tienen que ser diferentes. La tercera restricción indica que las diagonales secundarias tienen que ser diferentes. Recordamos que la diagonal principal se puede calcular como $dp = y - x$ y la secundaria $ds = y + x$. Esto resulta de la ecuación de las rectas paralelas a la diagonal principal ($y = x + k$) y a la secundaria ($y = -x + k$).

Definamos los siguientes acumuladores obtenidos de las restricciones del problema:

$$la_i = (a_{0,}\dots,a_{i-1})$$

$$fo_i = toSet(a_{0,}\dots,a_{i-1})$$

$$dp_i = toSet(a_{0,}a_1 - 1,\dots,a_{i-1} - (i-1))$$

$$ds_i = \ toSet(a_{0,}a_1 + 1,\dots,a_{i-1} + i - 1)$$

Dónde la_i, fo_i, dp_i, ds_i son la lista de alternativas (filas ya ocupadas), conjunto de filas ocupadas, diagonales principales ocupadas y diagonales secundarias ocupadas respectivamente. Las tres últimas son propiedades derivadas de la primera.

El problema generalizado puede ser formulado así:

$$p_i = \begin{cases} i \\ la_i \\ fo_i \\ dp_i \\ ds_i \end{cases}$$

El conjunto de alternativas para el problema es de la forma:

$$A_i = \{a_i{:}\,0..n-1 | a_i \notin fo_i, a_i \notin dp_i, a_i \notin ds_i\}$$

Así escogidas garantizamos que cada nueva alternativa coloca una reina que cumple la restricciones con respecto a la ya colocadas.

El subproblema p_{i+1} que se alcanza siguiendo la alternativa a_i se obtiene incrementando i, añadiendo a_i a la lista de alternativas y calculando las propiedades derivadas restantes. Esto es equivalente para el método $avanza(a_i)$ en *BT.* El método $retrocede(a_i)$ en *BT* decrementa la i, elimina la a_i de la lista de alternativas y calcula el resto de la propiedades derivadas.

El caso base más simple es $i = n$. Claramente dado la lista de alternativas la_n podemos reconstruir la solución como un conjunto de casillas donde se ubican las reinas.

Con estos elementos el método *BT* es muy adecuado para encontrar una solución (que puede ser realizada más eficientemente por la técnica *Randomize*), varias soluciones, todas ellas o el número de las mismas.

La técnica de la *PDR* puede ser adaptada para contar el número de soluciones para problemas de este tipo. Se trata de escoger adecuadamente la solución del caso base, la combinación de las soluciones parciales y el mecanismo de seleccionar alternativas.

Como se muestra abajo la solución del caso base es $(null, 1)$. Cada vez que alcanzamos el caso base establecido tenemos una solución. Combina soluciones parciales pasa al padre el número de soluciones del hijo alcanzado con esa alternativa. El método selecciona alternativa suma las soluciones alcanzadas según cada una de las alternativas y le asigna *null* a la alternativa en la solución parcial.

```
public Sp<Integer> getSolucionCasoBase() {
	Integer n = 1;
	return Sp.create(null, (double) 1);
}

public Sp<Integer> combinaSolucionesParciales(Integer a,List<Sp<Integer>> ls){
	return Sp.create(a, ls.get(0).propiedad);
}

public Sp<Integer> seleccionaAlternativa(List<Sp<Integer>> ls) {
	Double r = ls.stream().mapToDouble(sp->sp.propiedad).sum();
	return Sp.create(null, r);
}
```

El problema de encontrar una solución al problema anterior puede resolverse mediante *PLI* pero necesitamos escoger otras variables y sobre las que podamos establecer restricciones lineales. Esto puede hacerse escogiendo las variables binarias x_{ij} que toman valor 1 si ublicamos una reina en la casilla i, j el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j = 0, \dots, n-1$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i = 0, \dots, n-1$$

$$\sum_{(i,j):\varphi 1(d1)} x_{ij} \leq 1, \quad d1 = -(n-1), \dots, n-1$$

$$\sum_{(i,j):\varphi 2(d2)} x_{ij} \leq 1, \quad d2 = 0, \dots, 2n-2$$

$$bin\ x_{ij}, \quad i, j = 0, \dots, n-1$$

Dónde $\varphi 1(d1)$, $\varphi 2(d2)$ son los conjuntos de casillas (pares de índices) asociados a la diagonal principal $d1$ y a la diagonal secundaria $d2$ respectivamente. Con más detalle:

$$\varphi 1(d1) = \{(i,j) | j - i = d1 \wedge i \in [0..n-1] \wedge j \in [0..n-1]\}$$
$$\varphi 2(d2) = \{(i,j) | j + i = d2 \wedge i \in [0..n-1] \wedge j \in [0..n-1]\}$$

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

Para resolver el problema mediante AG escogemos un cromosoma *IndexPermutationChromosome* de tamaño n. El método *decode()* de este cromosoma nos va a proporcionar las permutaciones de la lista $[0,1,2,\dots,n-1]$. Los valores de la lista permutada los usaremos como las filas donde ubicar la reina correspondiente. Con esta elección del cromosoma ya conseguimos que se cumpla la restricción que obliga a que todas las filas sean distintas. El problema queda planteado con la función de *fitness*:

$$f = -(2n - |dp| - |ds|)$$

Con

$$dp = toSet(d(0), d(1) - 1, \dots, d(n-1) - (n-1))$$

$$ds = toSet(d(0), d(1) + 1, \dots, d(n-1) + i - 1)$$

Heurísticas para encontrar una solución: Problema del Sudoku

El problema consiste en rellenar con los enteros $[1..n]$ las casillas de un tablero $n \times n$ de tal manera cada fila, cada columna y cada subtabla contengan todos los enteros $[1..n]$ y una sola vez cada uno de ellos. Hay un conjunto de casillas que ya tienen asignado un número de $[1..n]$. Son las casillas definidas. El resto son las casillas libres a rellenar. Siendo $n = mk$ con k el tamaño de cada subtabla y m el número de subtablas. A cada casilla le podemos asignar las coordenadas i, j que toman valores en $0..n-1$, siendo la casilla $(0,0)$ la inferior izquierda. A cada subtablas le asociamos el índice t que toma valores en $0..n-1$ y se cumple $t = \left(\frac{j}{k}\right)k + i/k$ siendo / la división entera. Por ejemplo para $(i,j) = (2,4)$, $n = 9, k = 3$ tenemos $t = \left(\frac{4}{3}\right)3 + \frac{2}{3} = 3$. Concretemos el problema para $n = 9$. Para cada casilla definimos, además, su posición p en una lista construida poniendo una fila tras otra. Así $p = i + jn$. Una casilla tiene por lo tanto las coordenadas i, j, t, p. Dadas i, j podemos calcular las demás. También dada p podemos calcular las demás con las expresiones $i = p\%n, j = \frac{p}{n}$. Una casilla se puede definir por el par (i,j) o por su posición p. Las casillas definidas las proporcionamos con un *Map<Casilla,Integer>.* Las casillas que no están en el dominio del *Map* están libres.

Para enfocar el problema definimos una lista lp, de tamaño r igual al número de casillas libres, que contiene las posiciones libres del sudoku. Ahora buscamos una lista de alternativas la tal que la alternativa $la(k) = a_k$ definirá el valor a colocar en la casillas de posición $lp(k)$. Con k en $[0..r-1]$.

Con estos elementos el problema se puede definir como:

$$\begin{cases} la_r = (a_{0,} \dots, a_{r-1}) \\ |co_r[i]| = n, \quad i{:}\,[0..n-1] \\ |fo_r[j]| = n, \quad i{:}\,[0..n-1] \\ |to_r[t]| = n, \quad i{:}\,[0..n-1] \end{cases}$$

Y el problema generalizado:

$$p_k = \begin{cases} k \\ la_k \\ co_k[i], \quad i{:}\,[0..n-1] \\ fo_k[j], \quad i{:}\,[0..n-1] \\ to_k[t], \quad i{:}\,[0..n-1] \end{cases}$$

Donde $co_k[i], fo_k[j], to_k[t]$ son el conjunto de valores ocupados en la columna i, fila j y subtabla t. En estos valores están incluidos los de las casillas definidas correspondientes más las alternativas ya escogidas incluidas en la_k. Es decir $co_0[i], fo_0[j], to_0[t]$ ya contienen los valores en las casillas definidas.

Con el planteamiento anterior el problema se resuelve de forma similar a la de las reinas anterior.

El caso base más simple es $k = r$. Claramente dado la lista de alternativas la_r podemos reconstruir la solución como un conjunto valores para las casillas correspondientes.

El conjunto de alternativas para el problema es de la forma:

$$A_k = \{a_k{:}\,0..n-1 | a_k \notin co_k[i], a_i \notin fo_k[j], a_i \notin to_k[t]\}$$

Donde los valores i, j, t son los asociados a la casilla en posición $lp(k)$ y $C_{pk} = co_k[i] \cup fo_k[j] \cup to_k[t]$.

Con estos elementos el método *BT* es muy adecuado para encontrar una solución (que puede ser realizada más eficientemente por la técnica *Randomize*), varias soluciones, todas ellas o el número de las mismas.

Este problema admite una heurística consistente en ordenar la lista de posiciones según el grado de restricción de cada casilla. Si a la casilla en posición p le asociamos el número $s_p = n - |C_{p0}|$. Ordenando la lista de tal forma que la propiedad s_p vaya de menor a mayor tendremos la posibilidad de buscar primero alternativas para las casillas más restringidas. O dicho de otra forma con menos valores libres para asignar.

Para resolver el problema mediante *PLI* hay que escoger varias binarias adecuadas. Sea $\varphi(t) = \{(i,j) | t = \left(\frac{j}{k}\right) k + i/k\}$ y escogiendo las variables binarias x_{ijv} que toman valor 1 si la casilla i, j toma el valor v el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ijv} = 1, \qquad j, v = 0, \dots, n-1$$

$$\sum_{j=0}^{n-1} x_{ijv} = 1, \qquad i, v = 0, \dots, n-1$$

$$\sum_{(i,j):\varphi(t)} x_{ijv} = 1, \qquad t, v = 0, \dots, n-1$$

$$bin\ x_{ijv}, \qquad i, j, v = 0, \dots, n-1$$

La primera restricción indica que un color v sólo puede aparece una vez en la fila j. La segunda restricción que un color v sólo puede aparece una vez en la columna i. La tercera restricción indica que un color v sólo puede aparece una vez en la subtabla t.

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

Modelar el problema usando la restricción $allDifferent(x_{0,} \dots, x_{n-1})$ se deja como ejercicio.

Mediante AG también es posible resolver el problema pero hay que escoger un cromosoma específico que no veremos aquí.

Modelado a partir de grafos.

La solución de muchos problemas se puede encontrar construyendo previamente un grafo que lo modele. Para construir ese grafo pueden existir diferentes alternativas. Una vez construido el grafo podemos usar *BT*, *PDR* o algoritmos de tipo caminos mínimos como *A** o *Dijkstra* o con algoritmos similares al algoritmo de *Floyd.* Veamos algunos ejemplos.

Hemos de recordar que cuando usamos las técnicas de *BT*, *PDR* definimos implícitamente un grafo de problemas. Este grafo de problemas es dirigido y sin ciclos. En él existe un problema inicial (un vértice sin antecesores) y un conjunto de vértices finales (los casos base o caso final). En el caso de *BT* y *PDR* sólo existe un sucesor para cada arista que sale de un vértice (hay un solo subproblema) y las soluciones del problema inicial vienen definidas por caminos desde el vértice inicial a uno de los casos base. A cada problema del conjunto de problemas podemos asociarle un tamaño con la propiedad de que el tamaño del subproblema es siempre menor que el del problema.

Las ideas anteriores nos pueden servir para resolver un problema modelándolo como un grafo dirigido y sin ciclos dónde los vértices se corresponderán con problemas y las aristas salientes las posibles alternativas. Alternativamente a partir de un grafo de problemas (asociado a un problema de *BR* o *PDR*)

Una segunda forma es modelar la solución al problema original como el camino mínimo (en algunos casos máximo) sobre un grafo con pesos en las aristas (en este caso dirigido o no y

posiblemente con ciclos). Si lo hacemos así tenemos disponibles los algoritmos de *A** o *Dijkstra* (que podemos usar en grafos reales y virtuales) y si en el grafo podemos numerar de forma eficiente los vértices del grafo entonces tenemos disponible también el algoritmo de *Floyd*. Versiones inspiradas en este último algoritmo, cuando se pueden usar, son más versátiles al poder abordar el cálculo del camino máximo.

Para elegir entre *A**, *Dijkstra o Floyd* debemos tener en cuenta la información de que disponemos. Si la información de que disponemos es e_i, un número real que es el peso de cada arista, y queremos minimizar la suma de los pesos de las aristas del camino:

$$|C| = \sum_{i \in C} e_i$$

Entonces el algoritmo adecuado es *Dijkstra.*

Si además de e_i, el peso de cada arista, conocemos v_i el peso asociado a un vértice, w_{ijk} el peso asociado al vértice *i* asumiendo que se ha llegado a él mediante la arista *j* y se sale de él mediante la arista *k* y h_{ij} el costo estimado del camino del vértice entonces el algoritmo adecuado es el A*.

Como ya vimos los algoritmos *A** tratan con la idea más general que llamaremos **ruta**. Una ruta viene definida por un camino que va desde el vértice inicial hasta el vértice final pasando por un vértice intermedio (vértice actual). La ruta está definida por el vértice inicial V_i, el vértice final V_f y un vértice intermedio (el vértice actual) V_a. El coste asociado a la misma es la suma del coste ya conocido del camino *C* que va de V_i a V_a más el coste estimado h_{af} para llegar al final. El coste de esa ruta es:

$$|R| = \sum_{i \in C} e_i + \sum_{i \in C} v_i + \sum_{i \in C} w_{ijk} + h_{af}$$

Donde C es el camino que va de V_i a V_a. Si V_a es el vértice final entonces la ruta coincide con el camino y h_{af} es cero. Los algoritmos A* encuentran la ruta que minimiza $|R|$ aunque se pueden ajustar para resolver problemas de maximización.

Si el problema es modelado por un grafo pero no tenemos la información adecuada para usar los algoritmos *A** o *Dijkstra* entonces podemos enfocar el problema de forma similar al algoritmo de *Floyd*.

Por otra parte cuando resolvemos un problema vía *BT*, *PDR* aparece un conjunto de problemas que definen un grafo acíclico como hemos dicho. Si a las aristas de ese grafo le asociamos los pesos $\Delta fa_i = fa_{i+1} - fa_i$ (tal como hemos visto arriba) entonces el problema original que habíamos enfocado mediante *BT* o *PDR* se puede resolver mediante el algoritmo de *Dijkstra.* Recordamos que Δfa_i es el incremento del valor acumulado de la función objetivo al pasar del problema p_i al p_{i+1} siguiendo la arista dada.

Veamos algunos ejemplos.

Ejemplo 1

Dadas dos cadenas de caracteres X e Y, se desea transformar X en Y usando el menor número de operaciones básicas, que pueden ser de tres tipos: eliminar un carácter en una posición, añadir un carácter en una posición, y cambiar un carácter en una posición por otro. Por ejemplo, para transformar la cadena "*carro*" en la cadena "*casa*" se puede proceder de la siguiente forma:

- "*carr*" (eliminando *o* de la posición 5), "*cara*" (cambiando *r* por *a* en la posición 4) y "*casa*" (cambiando *r* por *s* en la posición 3)

Para diseñar un grafo tenemos que definir el tipo de vértices y de aristas y en su caso un vértice inicial y otro final. Una primera posibilidad es que los vértices tengan dos propiedades: s (una cadena de caracteres) e i un entero igual a la longitud de s. Con un invariante adicional: los prefijos de longitud i de s y de Y son iguales. Añadimos una propiedad derivada adicional $n1 = |s|$ y sea $n2 = |Y|$. Las variables dadas X, Y serán dos variables globales fijas. El vértice inicial será $(X, 0)$ y el vértice final $(Y, |Y|)$. El grafo sería dirigido y con pesos. Etiquetamos las aristas posibles como A, E, C, M (respectivamente añadir, eliminar, cambiar o mover), con pesos respectivos $(1,1,1,0)$. Las acciones a ejecutar cuando se transita por la arista correspondiente son $(A, i), (E, i), (C, i), (M, i)$ que respectivamente representan añadir el carácter en la posición i de Y al final de s, eliminar el carácter en la posición i de s, cambiar el carácter en la posición i de s por el de la posición i de Y, incrementar i en uno. En definitiva cada arista tiene asociada una etiqueta que representa una acción a llevar a cabo.

Dependiendo del valor en un vértice de i y los tamaños de $n1, n2$ de s, Y las aristas salientes serían:

$$\boldsymbol{Aristas}: = \begin{cases} \{A\}, & n1 - i = 0, n2 - i > 0 \\ \{E\}, & n1 - i > 0, n2 - i = 0 \\ \{C, E\}, & n1 - i > 0, n2 - i > 0, s(i) \neq Y(i) \\ \{M\}, & n1 - i > 0, n2 - i > 0, s(i) = Y(i) \end{cases}$$

Los vértices alcanzados según la arista escogida son:

$$Vértices\ siguientes = \begin{cases} (s, i) \xrightarrow{A} (s + Y(i), i + 1) \\ (s, i) \xrightarrow{C} (s(i) = Y(i), i + 1) \\ (s, i) \xrightarrow{E} (s - i, i) \\ (s, i) \xrightarrow{M} (s, i + 1) \end{cases}$$

Con el grafo así modelado la solución del problema es el camino mínimo desde el vértice inicial al final eliminando las aristas etiquetadas con M que no aportan a la solución. Para encontrar la solución podemos usar el algoritmo de *A** o *Dijkstra.*

Este grafo no tiene ciclos pero si puede tener vértices compartidos entre los caminos que conducen del vértice inicial al final. El conjunto de problemas en PDR (o de estados en BT) y sus relaciones de problema-subproblema define un grafo dirigido acíclico. Esto no orienta para buscar una solución en *PDR o BT*.

Para buscar una solución mediante *PDR o BT* asumimos que cada problema (o cada estado en el caso de BT) p_i del conjunto de problemas está asociado a un vértice. El problema inicial está asociado al vértice inicial y el caso base (problema final en BT) al vértice final. Las alternativas están asociadas a cada una de las aristas saliendo del vértice correspondiente. El problema siguiente, dada una alternativa (el método avanza en *BT*), es el vértice que se obtiene al tomar la alternativa correspondiente. Los vértices compartidos significan problemas compartidos por lo tanto es conveniente usar PDR que al usar memoria elimina los cálculos duplicados de las soluciones de los problemas compartidos.

Si el grafo no tiene ciclos ni vértices compartidos (o estos son muy pocos) entonces BT es una técnica adecuada.

Si el grafo tiene ciclos y vértices compartidos los algoritmos *A** o *Dijkstra* son los más adecuados.

Ejemplo 2

La circulación viaria de una ciudad viene representada por un grafo dirigido en el que los vértices se corresponden con intersecciones de calles y las aristas con las propias vías de tráfico, de manera que se dispone de la anchura, en número de carriles, de cada vía. Así, *N[i, j]* representa el número de carriles de la vía que une la intersección i con la intersección j, en el sentido "desde *i* a *j*" (0 si no hay una vía que una directamente a las dos intersecciones en el sentido indicado). Se define la anchura de un trayecto entre dos intersecciones a la correspondiente al tramo de menor anchura. Aplicar la técnica de Programación Dinámica para:

a) Implementar un algoritmo que obtenga la anchura correspondiente al trayecto de mayor anchura entre cada par de intersecciones.
b) Implementar un algoritmo que obtenga el trayecto de mayor anchura desde una intersección *origen* hasta otro *destino*.

Aquí cada vértice del grafo es una intersección. Cada vértice del grafo lo indexamos mediante un entero. Asumimos que el índice está comprendido en $[0, n-1]$ dónde n es el número de intersecciones. Hay una arista entre el vértice i y el j si $N(i,j) > 0$ siendo $N(i,j)$ el peso asignado a la arista. La anchura de un camino es la anchura mínima de cada una de sus aristas.

Queremos encontrar el *Camino de Anchura Máxima* entre dos vértices dados que representaremos por i, j. Con la información disponible este problema se puede resolver de forma similar al algoritmo de Floyd. El problema generalizándolo a este otro: encontrar el *Camino de Anchura Máxima* de i a j usando como camino intermedio ciudades cuyos índices estén en el conjunto $[0, k]$. Con este planteamiento cada problema lo podemos representar por (i, j, k). Donde i, j toman valores en $[o, n-1]$ y k en $[-1, n-1]$. El valor de $k = -1$ indica que el camino intermedio no contiene ninguna ciudad.

Representamos por g el grafo de partida y por (i, j) la arista, si la hay entre i y j, y por $|(i, j)|$ su peso. La primera decisión es escoger el tipo de alternativas. Para cada problema tenemos dos alternativas $\{Y, N\}$. La primera alternativa representa pasar por la intersección k. La segunda no pasar. Un camino en el grafo anterior lo vamos a representar por una secuencia de intersecciones conectadas cada una a la siguiente mediante una arista. Cada camino tiene una anchura que la anchura mínima de sus aristas. La no existencia de camino lo representaremos

por ⊥. Como soluciones parciales escogemos, de forma similar a otros casos, el par formado por la alternativa y la anchura del camino.

El problema lo podemos esquematizar de la forma:

$$A(p) = \begin{cases} (N, 0), & i = j \\ \perp, & i \neq j \wedge k = -1 \wedge (i,j) \notin g \\ (N, |(i,j)|), & i \neq j \wedge k = -1 \wedge (i,j) \in g \\ \max(\min\left(A(sp_{Y,1}), cmg(sp_{Y,2})\right), A(sp_{N,1})), & i \neq j \wedge k = -1 \wedge (i,j) \in g \end{cases}$$

$p = (i, j, k)$
$sp_{Y,1} = (\, i, k, k-1)$
$sp_{Y,2} = (\, k, j, k-1)$
$sp_{N,1} = (\, i, k, k-1)$

El problema inicial es $(v_i, v_f, n-1)$.