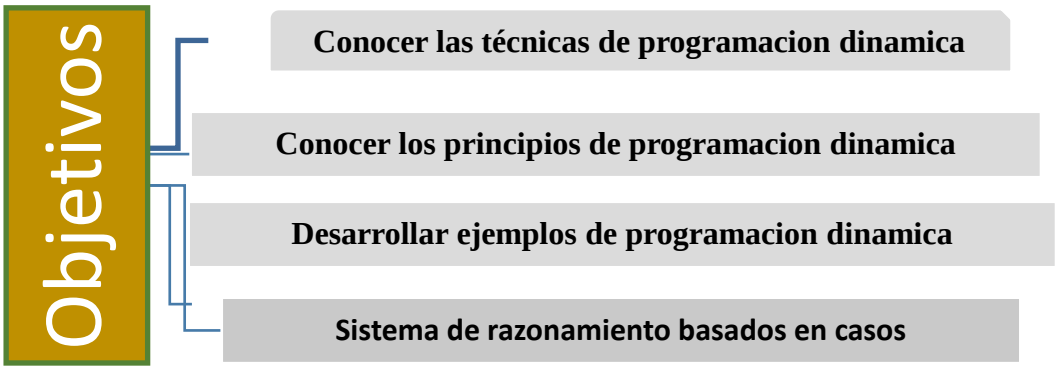
GUIA 6 *Programación dinámica*

David Hilbert (23 de enero de 1862, Konigsberg, Prusia oriental- 14 de febrero de 1943, Gotinga Alemania) fue un matemático alemán, reconocido como uno de los más influyentes del siglo XIX y principios del XX. Estableció su reputación como gran matemático y científico inventando y/o desarrollando un gran abanico de ideas, como la teoría de las invariantes, la axiomatización de la geometría y la noción de espacio de Hilbert, uno de los fundamentos del análisis funcional. Hilbert y sus estudiantes proporcionaron partes significativas de la infraestructura matemática necesaria para la mecánica cuántica la relatividad general. Fue uno de los fundadores de la teoría de la demostración, la lógica matemática y la distinción entre matemática y la metamatemática. Adoptó y defendió vivamente la teoría de conjuntos y los números transfinitos de Cantor. Un ejemplo famoso de su liderazgo mundial en la matemática es su presentación en 1900 de un conjunto de problemas abiertos que incidió en el curso de gran parte de la investigación matemática del siglo XX



Introducción

La programación dinámica es un enfoque general para la solución de problemas en los que es necesario tomar decisiones en etapas sucesivas. Las decisiones tomadas en una etapa condicionan la evolución futura del sistema, afectando a las situaciones en las que el sistema se encontrará en el futuro (denominadas estados), y a las decisiones que se plantearán en el futuro.

El procedimiento general de resolución de estas situaciones se divide en el análisis recursivo de cada una de las etapas del problema, en orden inverso, es decir comenzando por la última y pasando en cada iteración a la etapa antecesora. El análisis de la primera etapa finaliza con la obtención del óptimo del problema.

Algunos autores definen un algoritmo dinamico como un procedimiento que intenta resolver problemas disminuyendo su coste espacial aumentando el coste computacional

Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

Decir que un problema tiene *subproblemas superpuestos* es decir que un mismo subproblema es usado para resolver diferentes problemas mayores. Por ejemplo, en la sucesion de fibonacci, $F_3 = F_1 + F_2$ y $F_4 = F_2 + F_3$ — calcular cada término supone calcular F_2. Como ambos F_3 y F_4 hacen falta para calcular F_5, una mala implementación para calcular F_5 acabará calculando F_2 dos o más veces. Esto ocurre siempre que haya subproblemas superpuestos: una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a subproblemas que ya han sido resueltos anteriormente.

Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama memorización. Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que sabemos que vamos a necesitar de antemano.

En resumen, la programación dinámica hace uso de:

- Subproblemas superpuestos
- Subestructuras optimas
- Memorizacion

La programación dinámica toma normalmente uno de los dos siguientes enfoques:

- **Top-down**: El problema se divide en subproblemas, y estos subproblemas se resuelven recordando las soluciones en caso de que sean necesarias nuevamente. Es una combinación de memorizacion y recursion.
- **Bottom-up**: Todos los subproblemas que puedan ser necesarios se resuelven de antemano y después son usados para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Originalmente, el término de *programación dinámica* designaba únicamente a la resolución de ciertos problemas operaciones fuera del ámbito de la Ingeniería Informática, al igual que lo hacía *programación lineal*.

En este contexto no tiene ninguna relación con la programacion en absoluto; el nombre es una coincidencia. El término también se usaba por Richard Bellman, un matemático americano, para describir el proceso de resolver problemas donde hace falta calcular la mejor solución consecutivamente.

Principio de optimalidad

Cuando hablamos de *optimizar* nos referimos a buscar la **mejor** solución de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cual es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz.

A menudo, aunque no sea posible aplicar la estrategia voraz, se cumple el **principio de optimalidad de Bellman** que dicta que «dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima». En este caso sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en subproblemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y venceras, técnica similar a la de Programación Dinámica. La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

- Una enorme cantidad de subproblemas.
- Subproblemas cuyas soluciones parciales se solapan.
- Grupos de subproblemas de muy distinta complejidad.

Ejemplo de fibonacci

Una implementación de una función que encuentre el **n**-ésimo término de la sucesion de fibonacci basada directamente en la definición matemática de la sucesión realiza mucho trabajo redundante:

```
function fib(n)
    if n = 0 or n = 1
        return n
    else
        return fib(n - 1) + fib(n - 2)
```

Si llamamos, por ejemplo, a `fib (5)`, produciremos un árbol de llamadas que contendrá funciones con los mismos parámetros varias veces:

1. `fib(5)`
2. `fib(4) + fib(3)`
3. `(fib(3) + fib(2)) + (fib(2) + fib(1))`
4. `((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`
5. `(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`

En particular, `fib(2)` ha sido calculado dos veces desde cero. En ejemplos mayores, muchos otros valores de `fib`, o *subproblemas*, *son recalculados, llevando a un algoritmo de complejidad exponencial.*

Ahora suponemos que tenemos un simple objeto mapa, *m*, que guarda cada valor de `fib` que ya ha sido calculado y modificamos la función para que lo use y actualice. La función resultante tiene complejidad O(*n*), en lugar de exponencial:

```
var m := map(0 → 1, 1 → 1)
function fib(n)
    if n not in keys(m)
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

Esta técnica de guardar los valores que ya han sido calculados se llama memorizacion; esto es una implementación top-down, puesto que primero dividimos el problema en otros más pequeños y después calculamos y almacenamos los valores.

En este caso, también podemos evitar que la función use una cantidad de espacio lineal (O(*n*)) y use una cantidad constante en su lugar cambiando la definición de nuestra función y usando una implementación bottom-up que calculará valores pequeños de `fib` primero para calcular otros mayores a partir de éstos:

```
function fib(n)
    var previousFib := 1, currentFib := 1
    repeat n − 1 times
        var newFib := previousFib + currentFib
        previousFib := currentFib
        currentFib  := newFib
    return currentFib
```

En ambos ejemplos, `fib(2)` sólo se ha calculado una vez, y a continuación se ha usado para calcular tanto `fib(4)` como `fib(3)`, en lugar de calcularlo cada vez que era evaluado.

Ejercicios resueltos con Programación Dinámica

- Ejecución de n tareas en tiempo mínimo en un sistema de dos procesadores A y B
- Problema de la mochila
- Camino de costo minimo en un grafo dirigido
- Cambio de monedas con programación dinamica

¿Cuándo usar programación dinamica ?

Para utilizar programación dinámica, debe cumplirse dos condiciones:

Subestructura optima Un problema tiene subestructura optima cuando la solución a un problema se puede componer a partir de soluciones optimas de sus subproblemas

Superposición de problemas El cálculo de la solución optima implica resolver muchas veces un mismo subproblema. Siendo la cantidad de subproblemas muy pequeña

Ejemplo 1

Hallar factorial de N

$$N! = \begin{cases} 1 & N<=1 \\ N\ (N-1)! & \text{caso contrario} \end{cases}$$

Ejemplo 2

Calculo del coeficiente binomial

$$nCk = \begin{cases} 1 & k=0 \text{ o } k=n \\ n-1Ck-1 + n-1Ck & 0<k<n \\ 0 & \text{caso contrario} \end{cases}$$

Supongamos que $0 \leq k \leq n$ si calculamos directamente nCk mediante

```
Funcion C(n, k)
Inicio
    Si k =0 o k=n  devolver 1
    Sino
        Devolver C(n-1,k-1) + C(n-1,k)
    FinSi
Fin
```

Entonces muchos de los valores de C(i,j), con i<n y j<k se calculan una y otra vez. Por ejemplo, el algoritmo calcula C(5,3) como la suma de C(4,2) y C(4,3). Ambos resultados intermedios exigen calcular C(3,2). De modo similar, el valor de C(2,2) se utiliza varias veces.

	0	1	2		k-1	k
0	1					
1	1	1				
2	1	2				
n-1					$C_{(n-1,k-1)}$	$C_{(n-1,k)}$
N						$C_{(n,k)}$

Esta tabla de resultados intermedios nos permite obtener un algoritmo mas eficaz

Funcion Combinacion()

Inicio

RECUPERAR(F)
N_k =tomar8F,V)
R_k = tomar(F,V)
Leer N,R
Mientras ()
Si (N<= N_k y R<=R_k)
Retornar $A_{[N,R]}$
Sino
Mientras()
// Incremetar la matriz
FinMientras

FinSi

Leer N,R
Fin Mientras
Salvar(F,A,N_k,R_k)

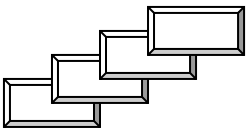
Fin

Ejemplo 3

Problema de la mochila

La idea básica es que existen N tipos distintos de artículos que pueden cargarse en una mochila; cada artículo tiene asociados un peso y un valor. El problema consiste en determinar cuántas unidades de cada artículo se deben colocar en la mochila para maximizar el valor total. Nótese que este enfoque resulta útil para la planificación del transporte de artículos en algún medio, por ejemplo: carga de un buque, avión, camión etc. También es utilizable este modelo en planificación de producción, por ejemplo enrutamiento de la producción a través de varias máquinas

Sea N objetos y una mochila M



Oi i= 1..n Oi tiene un peso W_i , un valor V_i

M puede llevar un peso que no sobre pase W
Por cada objeto Oi se puede llevar una fracción Xi de el

Se quiere llenar la mochila de tal forma que se maximice el valor de los objetos
Formalmente

Max { $\sum_{i=1} Xi\ Vi$ tq. $\sum_{i=1} X_i\ W_i \leq \boldsymbol{W}$

Precondicion. $V_i > 0$ $W_i > 0$ $0 \leq X_i \leq 1$, $\forall\ 1 \leq i \leq n$

Algoritmos voraz

Utilizaremos un vector X: (X_1, X_2,..... X_n) en donde guardaremos la fraccion X_i del objeto Oi hay que incluir.

```
Función mochila (W[1...n], V[1....n], W ) : matriz [1....n]
Inicio
        Para i de 1 a n
            X[i] =0
        FinPara
        Peso = 0
        Mientras (Peso < W)
            i = elegir el mejor objeto restante
            Si ( Peso  + W [i] ≤ W
                    X[i] = 1
                    Peso = Peso + W [i]
            Sino
                X[i] = (W - Peso) / W [i]
                Peso = W
            FinSi.
        FinMientras
        Retornar X
Fin
```

Ejemplo

N= 5 $\boldsymbol{W}$ = 100

	1	2	3	4	5
W	10	20	30	40	50
V	20	30	66	40	60
V/W	2.0	1.5	2.2	1.0	1.2

Para elegir el mejor objeto restante, se pueden utilizar tres criterios descritos mediante las siguientes funciones

Función de selección $\varnothing_1, \varnothing_2, \varnothing_3$

∅1 Seleccionar el objeto mas valioso
incrementa el valor de la carga del modo más rápido posible.
∅2 Seleccionar el objeto más pequeño restante. La capacidad se agota de la forma más lenta posible.
∅3 Seleccionar el objeto cuyo valor porcentual de peso sea el mayor posible.

Si seleccionamos los objetos por orden decreciente de valores seleccionamos.

Objeto	0_3 ,	0_5	, $\frac{1}{2}$	0_4	total
Valor	66	60	40/2		146
Peso	30	50	40/2		100

Si seleccionamos los objetos en orden de peso creciente

Objeto	0_1 ,	0_2 ,	0_3 ,	0_4	total
Valor	20	30	66	40	156
Peso	10	20	30	40	100

Si seleccionamos los objetos por orden decreciente de V/W

Objeto	0_3 ,	0_1 ,	0_2 ,	0_5	total
Valor	20	30	66	4/5 (60)	164
Peso	10	20	30	4/5 (50)	100

Problema de la mochila mediante programación dinámica

En esta sección se demostrará que este problema puede resolverse fácilmente aplicando el método de programación dinámica. El problema de la mochila se define como sigue: se tienen *n* objetos y una mochila. El objeto *i* pesa *Wi* y la mochila tiene una capacidad *M*. Si el objeto *i* se coloca en la mochila, se obtiene una ganancia *Pi*.
Sea X_i la variable que denota si el objeto *i* se selecciona o no. Es decir, se hace que X_i = 1 si el objeto *i* se escoge y 0 en caso contrario.

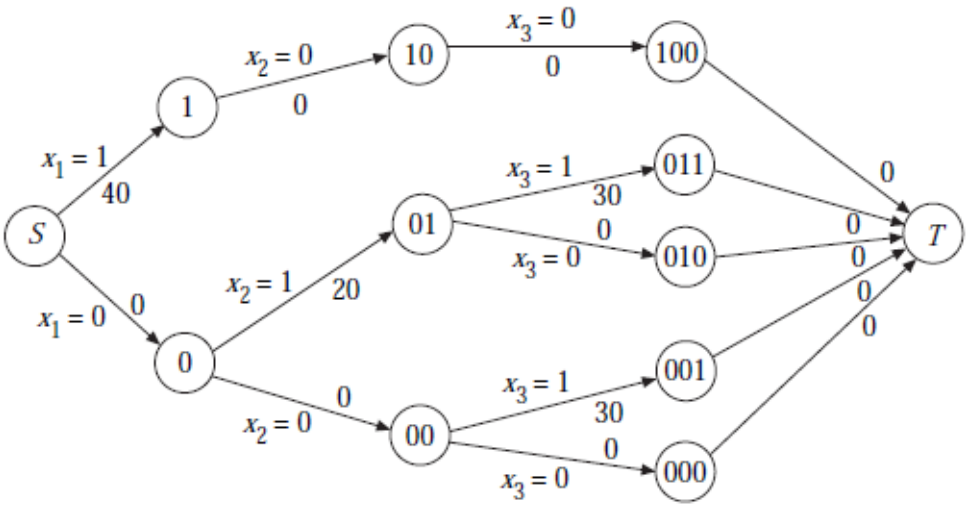
Si a X_1 se asigna 1 (el objeto 1 es escogido), entonces el problema restante se convierte en un problema de la mochila modificado, donde *M* se vuelve M_W_1.

Supongamos que se tienen tres objetos y una mochila con capacidad 10. Los pesos y las ganancias de estos objetos se muestran en la tabla siguiente:

Pesos y ganancias de tres objetos.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

Método de programación dinámica para resolver el problema de la mochila.



En cada nodo se tiene una etiqueta que especifica las decisiones que ya se han tomado hasta este nodo. Por ejemplo, 011 significa $X_1 = 0$, $X_2 = 1$ y $X_3 = 1$. En este caso se tiene interés en la ruta más larga, y es fácil darse cuenta de que ésta es

$$S \rightarrow 0 \rightarrow 01 \rightarrow 011 \rightarrow T$$

que corresponde a

$$X_1 = 0,$$
$$X_2 = 1$$
y $$X_3 = 1$$

con el costo total igual a $20 + 30 = 50$.

Ejemplo 4

Se quiere desarrollar un algoritmo para pagar una cierta cantidad a un cliente, empleando el menor número posible de monedas.

Supongamos que vivimos en un país en el que están disponibles monedas
S/. 100, S/. 25, S/. 10, S/. 5, S/. 1.

289 ⟶ 11 monedas : 2 x 100 + 3 x 25 + 2 x 5 + 4 x 1

Algoritmo voraz

```
Función DevolverCambio(N) : número de monedas.
Inicio
        C       = { 100, 25, 10, 5, 1}
        S       = ∅
        SUM     = 0
        Mientras ( SUM ≠ N )
                X = mayor elemento de C  tal que
                        S + X ≤ N
                Si no existe  ese elemento
                        Retornar "No existe solucion"
                Sino
                        S = S U{ una moneda de valor X}
                        SUM = SUM + X
                FinSi
        FinMientras
        Devolver S
Fin
```

N : cantidad a cambiar
S : conjunto de monedas
SUM: cantidad acumulada

Supongamos que se disponen monedas de 1, 4 y 6 unidades cambiar 8 implicaría 1x 6x 2x 1 = 8 Sin embargo sabemos que la mejor solución es 2 x 4

Solución con programación dinámica

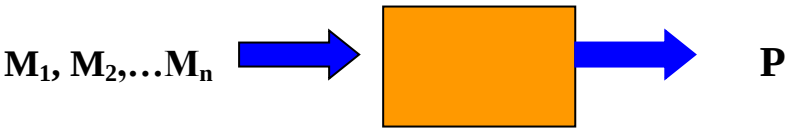
Cantidad	0	1	2	3	4	5	6	7	8
D1=1	0	1	2	3	4	5	6	7	8
D2=4	0	1	2	3	1	2	3	4	2
D3=6	0	1	2	3	1	2	1	2	2

```
Funcion Monedas()

Inicio

        Vector D[1..n] ={1, 4, 6}

        Matriz C[1..n, 0..N]

        Para i desde 1 hasta n hacer C[i, 0] = 0

        Para i desde 1 hasta n

                Para j desde 1 hasta N

                        Si   i = 1  ∧ j < D[i]   entonces C[i , j] = + ∞

                        Sino

                                Si   i = 1    entonces C[i , j] = 1 +  C[1 , j – D[1]]

                        Sino

                                Si   j < D[i]    entonces C[i , j] =  C[i-1 , j ]

                        Sino

                                C[i , j] =  min {C[i-1 , j ], 1 + C[i , j-D[i] ]

                        FinSi

                FinPara

        FinPara

        Devolver C[n,N]

Fin
```

Se observa que si existe una cantidad suficiente de monedas de una unidad, entonces siempre podemos hallar una solución para nuestro problema. De no ser así puede existir algún N para el cual no exista solución.

Ejemplo 5 Multiplicación de matrices

```
Accion MultiplicarMatrices()
Inicio
        Para i desde 1 hasta p
                Para j desde 1 hasta r
                        Cij = 0
                        Para k desde 1 hasta q
                                Cij = Cij + Aik * Bkj
                        FinPara
                FinPara
        FinPara
Fin
```



Si tenemos n matrices $M_1, M_2,\ldots M_n$

Podemos multiplicarlas de muchas maneras

Supongamos que A $_{13, 5}$

B $_{5, 89}$

C $_{89, 3}$

D $_{3, 34}$

A xB requiere 5785 multiplicaciones escalares

(A xB) xC requiere 3471 multiplicaciones escalares

((AxB)xC)xD requiere 1324 multiplicaciones escalares

Total requiere 10582 multiplicaciones escalares

Posibles formas de hallar AxBxCxD

((A B) C)D requiere 10582 producto escalares

(A B) (C D) requiere 54201 producto escalares **

(A (B C)) D requiere 2856 producto escalares

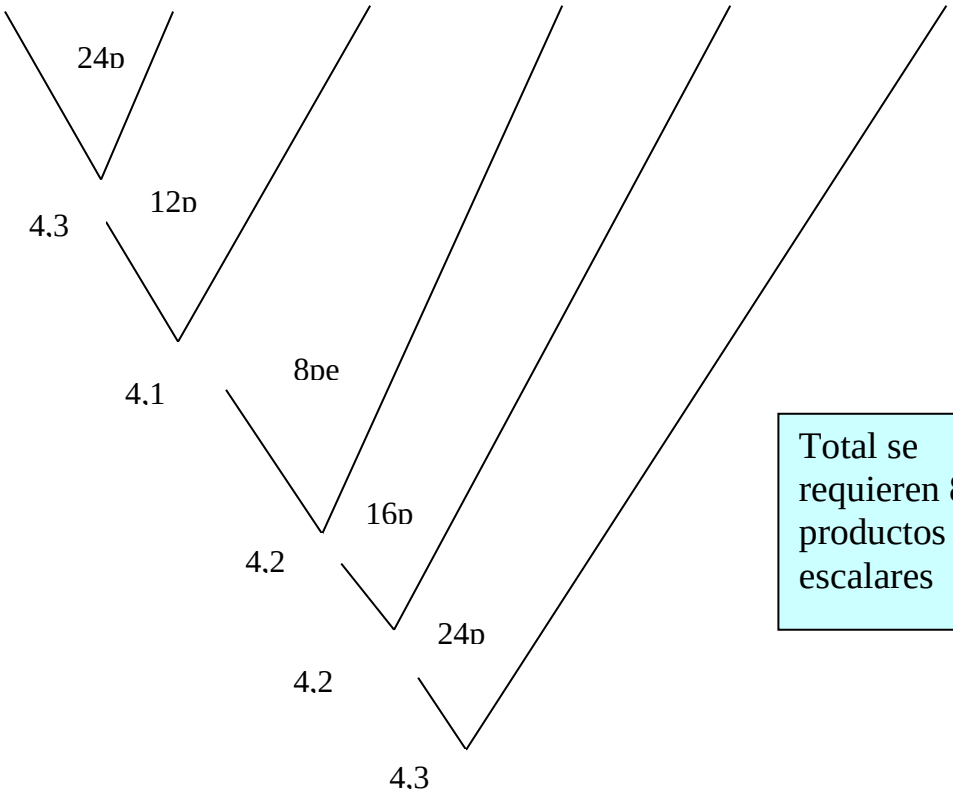
A ((B C) D) requiere 4055 producto escalares *

A (B (C D)) requiere 26418 producto escalares

* es mas eficiente en casi 19 veces que el mas lento **

Ejemplo

A4,2 B2,3 C3,1 D1,2 E2,2 F2,3



Total se requieren 84 productos escalares

Solucion dinamica

Multiplicar pares

$M_1 \times M_2$
$M_2 \times M_3$

….
$M_{n-1} \times M_n$

Solo existe una solucion para Mi x Mj

Multiplicar triples

$M_1 \times M_2 \times M_3$
$(M_1 \times M_2) \times M_3$
$M_1 \times (M_2 \times M_3)$

Se elige el menor de todas las posibilidades y se memoriza

	B	C	D	E	F
A	Ax B (24 pe)	AxBxC(14 pe)	AxBxCxD(24 pe)		
B		BxC (6 pe)			
C			CxD (6 pe)		
D				DxE (pe)	
E					ExF (24 pe)

Sistemas de razonamiento basado en casos

El Razonamiento Basado en Casos (CBR) es una metodología de Inteligencia Artificial para realizar aprendizaje que ha logrado buenos resultados en muchos campos de aplicación [11].

¿Qué es Razonamiento basado en casos?

> "Más sabe el diablo por viejo que por diablo"
> "La experiencia es la madre de la Ciencia"

Básicamente, resuelve un nuevo problema recordando situaciones similares anteriores y reutiliza el conocimiento y la información de estas situaciones.

El Razonamiento Basado en Casos (CBR) es un método para resolver problemas recordando situaciones previas similares y reutilizando la información y el conocimiento sobre esa situación [10, 11]. La idea original básica de Riesbeck y Schank [13] es la siguiente: CBR resuelve problemas nuevos adaptando las soluciones dadas a otros resueltos con anterioridad. Aamod y Plaza [1] describen CBR como un proceso que consta de cuatro pasos:

a) RECUPERAR el caso o casos más similares.
b) REUTILIZAR la información y el conocimiento de ese caso para resolver el problema.
c) REVISAR la solución propuesta.
d) GUARDAR las partes de esta experiencia que se consideren útiles para resolver futuros problemas

Ejemplo 6

Caso 1 Un alumno solicita un certificado de estudios en la oficina de matrícula. El encargado conoce el procedimiento dado que lo repite frecuénteme. Sin embargo si solicitan un revalida de un diploma de título de la Universidad de Mozambique, no es usual, en este caso tendrá que averiguar la solución, y una vez obtenida, lo memoriza para uso posterior.

Caso 2 Pensemos en un médico que examinando un nuevo paciente recuerda un caso parecido unas semanas atrás; encuentra una similitud importante de los síntomas y decide asumir que posee la misma enfermad y le trata de la misma manera, pues el tratamiento resultó efectivo en la ocasión anterior.

Caso 3 Un bróker de la bolsa le recomiendan una aparente buena inversión pero ciertos síntomas del mercado le recuerdan que hace un año con una situación parecida perdió mucho dinero y decide no invertir en esta ocasión.

Caso 4 Un CHEF es un planificador basado en casos que toma como entrada una conjunción de submetas[12] que necesita lograr para conseguir un plan como salida. Su dominio es la creación de recetas. Las recetas son vistas como planes.

Las recetas proporcionan la secuencia de pasos a seguir para preparar un plato. Por tanto la entrada de CHEF son las metas que se pueden conseguir con las recetas (por ejemplo, incluye pescado, sofreír, sabor salado) y la salida es una receta (plan), que puede obtener esas metas.

Como planificador basado en casos, CHEF crea sus planes a partir de viejos planes que funcionaron en situaciones similares y los modifica para adaptarlos a la nueva situación. Por tanto el primer paso en la creación de un plan es recuperar una vieja receta que cumpla el mayor número posible de metas de la entrada. Para recordar esta clasificación indexa los planes por las metas que logra. Ternera con brócoli es indexado por varias metas, entre ellas, incluye carne, incluye una verdura fresca, sofreír y lograr sabor salado.

El siguiente paso es adaptar el viejo plan a la nueva situación. Esto se hace en dos etapas. Primero, se reinstancia el viejo plan, es decir, crea una instancia en la que -5- sustituye los nuevos objetos por los del viejo plan. Por ejemplo, si está creando una receta de pollo con guisantes desde la receta de ternera con brócoli, sustituye ternera por pollo y brócoli por guisantes. Para poder hacer esto, necesita conocer los roles que desempeñan los objetos en la vieja receta. CHEF tiene un conocimiento bastante limitado sobre esto y para saber qué objetos sustituir y por cuáles busca las similitudes entre los objetos del viejo plan y los del nuevo y sustituye los objetos del nuevo plan por los más similares del viejo plan. Por ejemplo, si el pollo y la ternera están definidos como carne, los sustituye.

En la segunda etapa, aplica criticas de objeto para adaptar el viejo plan a la nueva situación. Un ejemplo es, el pato no debe tener grasa antes de sofreírlo. Se expresaría de la siguiente manera: Después del paso: deshuesar el pato hacer: quitar la grasa al plato porque: en este momento el pato tiene grasa Esta crítica está asociada al objeto pato, y cada vez que se usa el pato en una receta se dispara la crítica. Si hay un paso de deshuesar el pato en la receta que se está creando, se añadiría detrás un paso indicando que se debería de quitar la grasa al pato. Las críticas de objeto generalmente añaden pasos de preparación especiales a la receta (deshuesar, quitar la grasa…).

Las críticas son la forma en que CHEF codifica el conocimiento sobre procedimientos especiales asociados al uso de objetos de su dominio. Su uso durante la adaptación muestra la interacción entre el uso de experiencia y de conocimiento general de los sistemas CBR. Después de la reinstanciación y la aplicación de críticas, CHEF obtiene un plan completo. Para validar el plan lo ejecuta en un simulador muy parecido al mundo real y si es correcto lo almacena, si no crea una explicación causal de porqué no funcionó el plan y lo usa como índice para reparar el plan. Los TOPs (paquetes de organización temática) son estructuras que encierran conocimiento general del dominio Relacionan el conocimiento acerca de cómo interaccionan entre sí los objetivos del caso considerado.

En CHEF, se usan para caracterizar de un modo abstracto las interacciones entre las etapas de una receta. Así, el sistema incluye un TOP EFECTOS LATERALES: CONDICIÓN INVALIDADA : CONCURRENCIA. CONCURRENCIA significa que dos objetivos han sido llevados a cabo a través de una sola acción. EFECTO LATERAL: CONDICIÓN INVALIDADA significa que ha tenido lugar un efecto lateral: intentar satisfacer uno de los objetivos impide la satisfacción del otro. Este TOP caracteriza, por ejemplo, el fallo que ocurre cuando se cocina brécol con ternera y el brécol se reblandece. Los TOPS son importantes porque capturan conocimiento a veces independiente del dominio de resolución de problemas. En CHEF, una solución al problema es dividir la acción en dos etapas de modo que los objetivos no interfieran entre sí (cocinar los ingredientes en dos etapas). Otra solución basada en un TOP sería añadir otra acción que contrarrestara el efecto de la anterior (añadir algo que absorbiera el agua que reblandece el brécol). Es importante observar que estas estrategias son bastante -6- generales y podrían constituir sugerencias interesantes para la reparación de planes en otros muchos dominios. CHEF emplea tres tipos de estrategias de reparación: (1) dividir y reformular los pasos del plan, (2) alterar el plan para que se ajuste a la solución esperada o (3) buscar un plan alternativo. En primer lugar CHEF elige la estrategia de reparación buscando casos similares en el TOP que sugieran una reparación y después chequea la posibilidad de aplicar las condiciones para cada plan reparado. Así encuentra el más apropiado. Aplica el plan de reparación apropiado, resolviendo el plan fallido. CHEF realiza otra etapa más antes de acabar, actualiza su conocimiento del mundo para poder anticiparse y no repetir el error que acaba de cometer. Para ello, encuentra unos predictores del tipo de fallo y lo usa como índice para advertir de la posible ocurrencia del fallo. Evidentemente estos índices son extraídos de la explicación previa del fallo, aislando la parte de la explicación que describe características del medio responsables del fallo. La siguiente figura muestra la arquitectura de CHEF. Los rectángulos representan sus unidades funcionales, los círculos sus fuentes de conocimiento y los óvalos las entradas y salidas de cada proceso funcional.

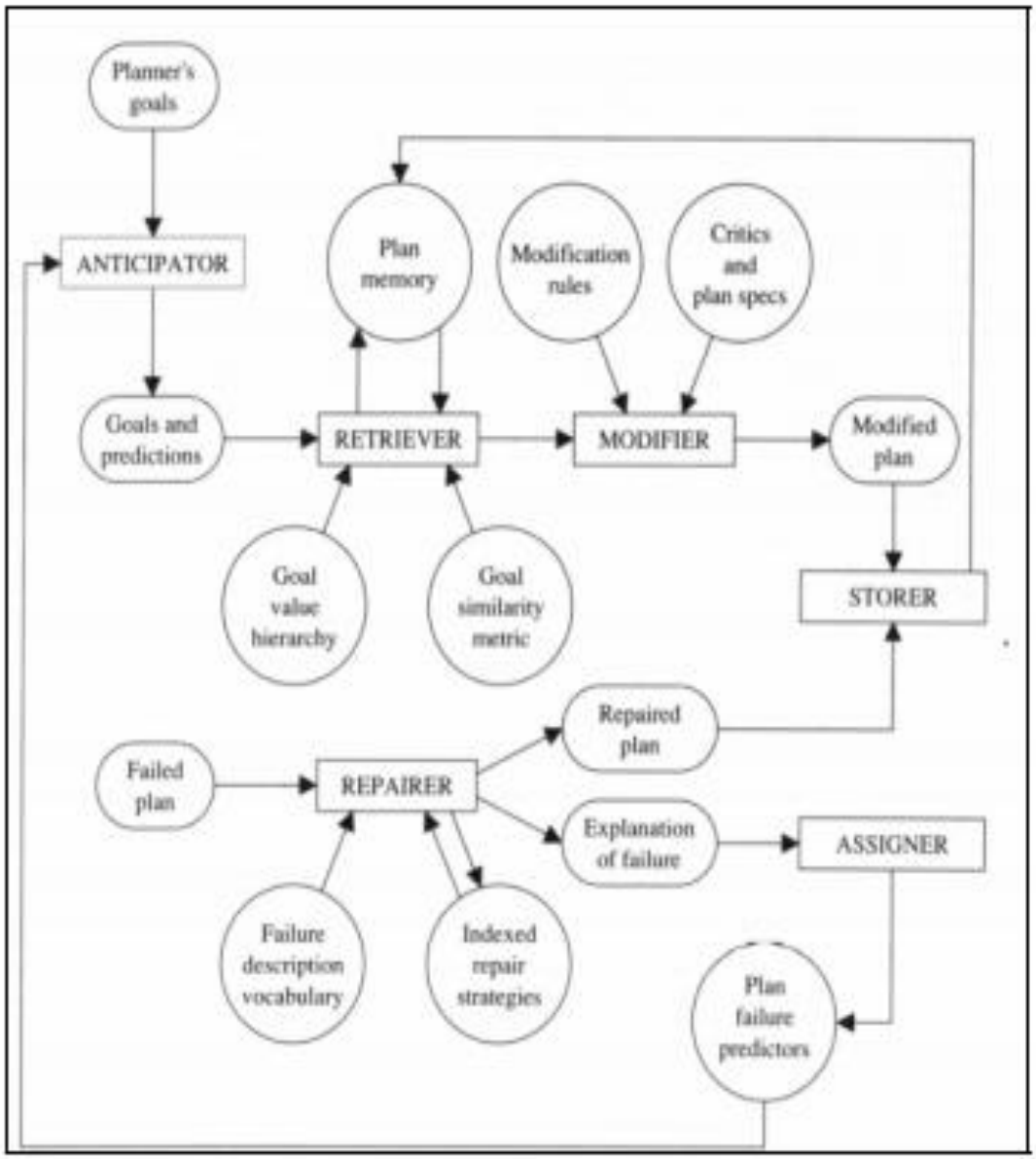

Planner's goals
ANTICIPATOR
Plan memory
Modification rules
Critics and plan specs
Goals and predictions
RETRIEVER
MODIFIER
Modified plan
Goal value hierarchy
Goal similarity metric
STORER
Repaired plan
Failed plan
REPAIRER
Explanation of failure
ASSIGNER
Failure description vocabulary
Indexed repair strategies
Plan failure predictors

Figura 1. Arquitectura Funcional de CHEF

Ejercicios propuestos

a) **En que consiste en principio de optimalñidad de Bellman**
b) **EL proceso de CBR consta de cuánto pasos? ¿Cuáles son?**

Mediante programación dinámica

a) Hallar el factorial de N
b) Hallar fibonacci de N
c) Construya un algoritmo de programación dinámica para hallar el coeficiente polinomial ${}_nC_k$
d) Utilizando programación dinámica, construya un algoritmo para el problema de la mochila. Asuma que los objetos no se pueden fragmentar en trozos más pequeños, así que se tiene que decidir si se toman un objeto o se deja, pero no se puede tomar una fracción.
e) Mediante programación dinámica construya un algoritmo para hallar los caminos más cortos entre un nodo origen y los demás
f) Construya una solución dinámica para producto de varias matrices
g) Mediante programación dinámica construya un algoritmo para hallar los caminos más cortos entre todos los nodos

REFERENCIAS BIBLIOGRAFICAS

1) [AHO 1988], Alfred V.
"Estructura de Datos y algoritmos" Addison Wesley. 1988.
2) [ALMEIDA 2008], Francisco Almeida "Introducción a la programación paralela"
Paraninfo 2008 España ISBN 978-84-9732-674-2.

3) [ANDERSON 2008] Anderson James "Redes Neuronales" Edit AlfaOmega México
ISBN 978-970-15-1265-4

4) [BRASSARD 2001], G. / BRATLEY, T.
"Fundamentos de Algoritmia". Prentice Hall. 2001

5) [CORTEZ 2010] Cortez Vásquez Augusto. "Algoritmia". Edit EsVega Lima Perú
ISBN 978-612-00-0257-5

6) [CORTEZ 2012] Cortez Vásquez Augusto. "Algoritmia, Técnicas algorítmicas"
Edit San Marcos Lima Peru ISBN 978-612-00-0964-2

7) [LEE 2005] R.Lee. "Introducción al diseño y análisis de algoritmos"
Edit Mc Graw Hill Mexico ISBN 978-970-10-6124-4

8) [LEIJA 2009] Lorenzo Leija "Metodos de procesamiento avanzado"
Edit Reverte Mexico 209 ISBN 978-607-7815-01-3

9) [WEISS 2001] ALLEN WEISS, Mark
"Lenguaje de programación Java". Addison Wesley. Madrid 2001.

10) [WEISS 2003] ALEN WEISS, Mark
"Estructura de Datos en Java". Addison Wesley. 2003

11) [BREGON 2003] ANIBAL BREGON Un sistema de razonamiento basado en casos para la clasificación de fallos en sistemas dinámicos
Dept. de Informática Escuela Politécnica Superior ETS Ingeniería Informática Univ. de Burgos Univ. de Valladolid Burgos 47011 Valladolid
http://www.lsi.us.es/redmidas/CEDI/papers/824.pdf

12) [LOZANO 2003] LAURA LOZANORazonamiento Basado en Casos: Una Visión General Universidad de Valladolid
HTTPS://WWW.INFOR.UVA.ES/~CALONSO/IAI/TRABAJOALUMNOS/RAZONAMIENTO%20BASADO%20EN%20CASOS.PDF