UNIVERSIDAD DE GRANADA

DECSAI

Algorítmica

Grado en Ingeniería Informática

Tema 4 – Programación dinámica

Manuel Pegalajar Cuéllar

manupc@ugr.es

Departamento de Ciencias de la Computación e Inteligencia Artificial

http://decsai.ugr.es

Objetivos del tema

- Plantearse la búsqueda de varias soluciones distintas para un mismo problema y evaluar la bondad de cada una de ellas.
- Conocer los criterios de aplicación de cada una de las distintas técnicas de diseño de algoritmos.
- Comprender la técnica de resolución de problemas por programación dinámica, e identificar las diferencias con divide y vencerás y con avance rápido.
- Saber identificar problemas que cumplen el principio de optimalidad y qué es necesario para poder aplicar esta técnica.

Estudia este tema en...

- G. Brassard, P. Bratley, “Fundamentos de Algoritmia”, Prentice Hall, 1997, pp. 291-318
- J.L. Verdegay: Lecciones de Algorítmica. Editorial Técnica AVICAM (2017).

Anotación sobre estas diapositivas:

El contenido de estas diapositivas es esquemático y representa un apoyo para las clases presenciales teóricas. No se considera un sustituto para apuntes de la asignatura.

Se recomienda al alumno completar estas diapositivas con notas/apuntes propios, tomados en clase y/o desde la bibliografía principal de la asignatura.

UNIVERSIDAD DE GRANADA

DECSAI

Programación dinámica

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Los algoritmos de programación dinámica se caracterizan por:

- Construir la solución paso a paso, por etapas.
- Dividir un problema de tamaño **n** en uno o varios problemas de tamaño **n-1**, que **se solapan entre sí**.
- Mantiene en memoria la solución a los subproblemas solucionados para evitar cálculos repetidos.

Ventajas

- Eficientes en tiempo.
- Devuelven la solución óptima
- Fáciles de implementar.

Inconvenientes

- Ineficientes en espacio.
- Mayor dificultad para plantear el problema.
- Tiene varios requisitos que deben cumplirse para poder aplicar la técnica.

En comparación con DyV

- DyV se aplica cuando los **subproblemas son independientes.**
- P.D. se aplica cuando los **subproblemas se solapan.**
- DyV utiliza recursividad (+tiempo, -memoria).
- P.D. intenta evitar recursividad (-tiempo, +memoria).
- P.D. resuelve sólo problemas de optimización.
- P.D. devuelve la solución óptima (DyV no tiene porqué tener una función de optimización objetivo).
- DyV repetiría muchos cálculos.
- P.D. mantiene en memoria las subsoluciones para evitar repetir cálculos.

En comparación con Greedy (algoritmos voraces)

- Greedy selecciona un elemento en cada etapa y genera una única subsolución.
- P.D. selecciona un elemento en cada etapa, pero genera múltiples caminos de etapas a seguir.
- Greedy no asegura optimalidad.
- P.D. asegura optimalidad.
- Greedy es eficiente en tiempo y memoria.
- P.D. es eficiente en tiempo, pero no en memoria.

Idea general de los algoritmos de Programación dinámica

- Se utilizan para resolver problemas de optimización (maximización/minimización).
- Se expresan mediante ecuaciones recurrentes.
- Solucionan uno o varios subproblemas de tamaño menor para resolver un problema de tamaño mayor.
- Normalmente, los subproblemas se solapan entre sí.
- Hacen uso de memoria para evitar repetir operaciones.

Idea general de los algoritmos de Programación dinámica

Algunos algoritmos que **no son de programación dinámica**, pero que también hacen uso de la idea de abusar de la memoria para conseguir una mayor eficiencia:

- Cálculo de la sucesión de Fibonacci.
- Cálculo del coeficiente binomial.
- El problema de los Play-Offs

La sucesión de Fibonacci (I)

- Se calcula como:

$$F(n) = \begin{cases} n & n \leq 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- Algoritmo:

```
5   unsigned long Fibonacci(int n) {
6
7       if (n<=1) return n;
8       return Fibonacci(n-1)+Fibonacci(n-2);
9
10  }
```

El algoritmo es $O((1+\sqrt{5})/2)^n)$

Según la recurrencia, podemos ver que **se repiten muchas operaciones**.

La sucesión de Fibonacci (II)

- Podemos ***abusar*** de la memoria guardando los cálculos previos para hacer el cálculo de F(n):

- Algoritmo:

```
12  unsigned long FibonacciEficiente(int n) {
13
14      unsigned long t1= 1, t2= 0, tn;
15
16      if (n<=1) return n;
17      for (int i= 2; i<=n; i++) {
18          tn= t1+t2;
19          t2= t1;
20          t1= tn;
21      }
22      return tn;
23  }
```

El algoritmo es O(n)

Según el código, podemos ver que **se ahorran muchas operaciones**.

Cálculo del coeficiente binomial (I)

- Se calcula como:

$$\binom{n}{k} = \frac{n!}{(n-k)!\cdot k!} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Algoritmo:

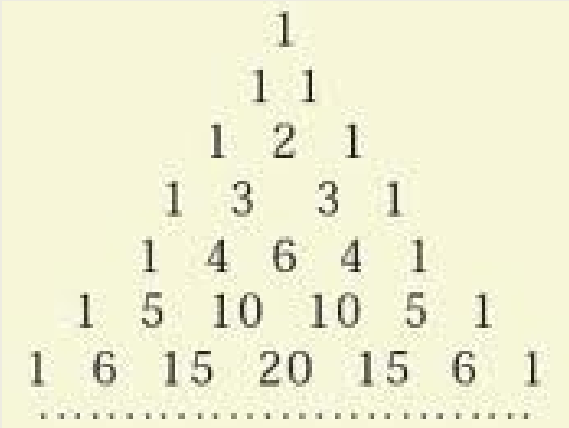
Función CoefBinomial(n, k)

Si k=0 ó k=n **entonces Devolver** 1
En otro caso,
  Devolver CoefBinomial(n-1, k-1)+CoefBinomial(n-1, k)
Fin-En otro caso

Según la recurrencia, podemos ver que **se repiten muchas operaciones**.

Cálculo del coeficiente binomial (II)

- Solución para mejorar la eficiencia: abusar de la memoria para guardar las soluciones previas calculadas (triángulo de Tartaglia):



Almacenaremos todas las subsoluciones en una tabla.

Ejemplos: C(3/2)= 3; C(4/2)= 6; C(5/1)=5

Cálculo del coeficiente binomial (III)

- Algoritmo para el cálculo del triángulo de Tartaglia:

```
Función CoefBinomialPD(n, k)

Para cada i=0 hasta n, hacer:
   Inicializar Tabla[i][i]=1, Tabla[i][0]=1
Fin-Para
Para cada d=2 hasta n, hacer:
   Para cada s=1 hasta minimo(d, k+1), hacer:
      Tabla[d][s]=Tabla[d-1][s]+Tabla[d-1][s-1]
   Fin-Para
Fin-Para
Si k=0 ó k=n, entonces:
   Devolver 1
En otro caso,
   Devolver Tabla[n-1][k]+Tabla[n-1][k-1]
Fin-En otro caso
```

¡Se ahorran muchas operaciones!

El problema de los Play-Offs (I)

- En una competición juegan dos equipos A y B. El primero que gane un total de **n** partidas gana el torneo. Sabiendo que **p** es la probabilidad de que A gane un partido y asumiendo que no puede haber empates (por tanto, **1-p** es la probabilidad de que gane B), se desea saber con qué probabilidad ganará el equipo A antes de que se juegue ningún partido.
  - Sea P(i,j) la probabilidad de que A gane cuando a A le quedan i victorias pendientes y a B le quedan j. Entonces:
    - P(0, j)= 1
    - P(i, 0)= 0
    - P(n, n) es lo que queremos saber
    - P(0, 0) no está definido.

El problema de los Play-Offs (II)

- La probabilidad P(i,j), en función de quién haya ganado el partido anterior, es:

$$P(i,j)= p*P(i-1, j)+(1-p)*P(i,j-1).$$

- El algoritmo recursivo para resolverlo sería:

```
Función PlayOffs(i, j, p)

  Si i=0 entonces Devolver 1
  Si j=0 entonces Devolver 0
  Devolver p*PlayOffs(i-1, j) + (1-p)*PlayOffs(i,j-1)
```

¡Se repiten muchas operaciones!

El problema de los Play-Offs (III)

- Como en los problemas anteriores, podemos abusar de la memoria para almacenar las subsoluciones previas hasta llegar a conocer **P(n,n)**

- El algoritmo para resolverlo sería:

```
Función PlayOffsPD(n, p)
T= Tabla de tamaño [0..n][0..n]
Para i=1 hasta n hacer:
  T[0][i]= 1; T[i][0]= 0
Fin-Para
Para i=2 hasta n hacer:
  Para j=2 hasta n hacer:
    T[i][j]= p*T[i-1][j] + (1-p)*T[i][j-1]
  Fin-Para
Fin-Para
Devolver T[n][n]
```

¡Se ahorran muchas operaciones!

UNIVERSIDAD DE GRANADA

DECSAI

Programación dinámica

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Requisitos que deben cumplirse para aplicar la técnica

- El problema a resolver debe ser un problema de optimización (maximización/minimización).
- El problema debe poder resolverse por etapas.
- El problema debe poder modelarse mediante una ecuación recurrente.
- Debe existir uno o varios casos base al problema.
- Debe cumplirse el Principio de Optimalidad de Bellman

Principio de optimalidad de Bellman (P.O.B.)

- **Si una secuencia de pasos para resolver un problema es óptima, entonces cualquier subsecuencia de estos mismos pasos también es óptima.**

Pasos para resolver un problema mediante P.D.

- El problema debe poder resolverse por etapas.
- Plantear el problema como una ecuación de minimización o maximización recursive, con uno o varios casos base.
- Encontrar una representación de la ecuación para almacenar las subsoluciones (una tabla, un vector, etc.).
- Verificar que se cumple el P.O.B.
- Diseñar el algoritmo.

UNIVERSIDAD DE GRANADA

DECSAI

Programación dinámica

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Enunciado del problema

Supongamos que compramos un refresco en una máquina y, tras pagar, esta tiene que devolver un cambio ***N*** con el mínimo número de monedas ¿Qué algoritmo debería seguir la máquina?

Asumiremos lo siguiente:

- Hay monedas de **n** valores diferentes.
- Las monedas tipo **i** tienen valor $\mathbf{d}_i > 0$.
- **Las monedas están ordenadas por su valor, en orden creciente.**
- Al cliente hay que devolverle un cambio igual a **N**.
- Objetivo: minimizar el número de monedas a devolver.

Diseño del problema del cambio de monedas: Resolución por etapas

- Asumimos que los tipos de monedas están ordenados de forma creciente, de modo que su valo es: $\mathbf{d_1<d_2<d_3<\ldots<d_n}$
- Sin pérdida de generalidad asumiremos $\mathbf{d_1}$=1 para simplificar la explicación del problema.
- Llamaremos a **T(i,j)**= número mínimo de monedas necesario para devolver una cantidad igual a **j** usando sólo monedas del tipo desde **1** hasta **i**.
- Por tanto, el **objetivo es conocer T(n, N)**
- Podemos resolver el problema **T(n, N)** por etapas para devolver la cantidad **N**, donde en cada etapa podemos comprobar si se puede devolver una moneda del máximo valor para construir la solución, o no devolverla (en ese caso sólo se usarían monedas de hasta el tipo de moneda anterior).

Diseño del problema del cambio de monedas: Ecuación (I)

- **Objetivo**: Conocer T(n,N) = Mínimo número de monedas para devolver la cantidad **N** considerando usar cualquier subconjunto de monedas de cualquier tipo, **desde el 1 hasta el n**.

- **Casos base**:
  - T(i, 0)= 0 No hay que devolver cambio
  - T(1, j)= j Sólo devolvemos monedas de tipo 1

Diseño del problema del cambio de monedas: Ecuación (II)

- **Caso general**:

$$T[i][j] = \min\{T[i-1][j],\ 1+T[i][j-d_i]\}$$

El caso general supone que para devolver la cantidad **j** usando sólo monedas **desde el tipo 1 hasta el i**, el mínimo número de monedas se conseguirá de una de las dos formas siguientes:

- No echar una moneda de tipo **i** y usar sólo **desde 1 hasta i-1**: primera parte **T[i-1][j]**
- Echar una moneda de tipo **i** , quedando por devolver sólo la cantidad **$j-d_i$** y viendo si podemos seguir echando monedas del mismo tipo **i**: Segunda parte **$1+T[i][j-d_i]$**

Diseño del problema del cambio de monedas: Verificación del P.O.B.

- Para una cantidad **j** fija, T[i-1][j] es óptimo. El caso base T[1][j] es óptimo trivial, ya que sólo hay un tipo de monedas y sólo existe un posible cambio a dar.
- Cuando hay 2 tipos de monedas, T[2][j] selecciona el mínimo número de monedas entre T[1][j] y seleccionar una moneda del tipo 2, quitando tantas como posibles hasta la cantidad que valga este tipo de monedas; es decir, $1+T[2][j-d_2]$. Este valor también es óptimo cuando $j>=d_2$.
- Cuando llegamos a T[i-1][j], este valor también tiene que ser óptimo, dado que el algoritmo ha ido seleccionando el mínimo.
- Cuando llegamos a T[i-1][j], ¿podría existir una secuencia de decisiones distintas de las tomadas para seleccionar un número de monedas inferior a T[i-1][j]? Reducción al absurdo: Sería imposible, dado que eso implicaría una solución mínima inferior a la proporcionada por la ecuación recurrente, que siempre calcula el número de monedas mínimo. Queda demostrado que T[i-1][j] es óptimo.

Diseño del problema del cambio de monedas: Representación

- Vamos a representar la solución al problema como una tabla T(i,j):
  - **Filas (i):** Asociadas a los tipos de monedas, ordenados ascendentemente por su valor $\mathbf{d_1<d_2<d_3<\ldots<d_n}$
  - **Columnas (j):** Cantidades a devolver, desde 0 hasta **N**
  - **Celdas:** Cada celda T(i,j) contendrá el mínimo número de monedas necesario para devolver la cantidad **j** asumiendo que consideramos devolver monedas desde el **tipo 1 hasta el i**.
- Ejemplo: Devolver 8 ctmos con monedas de d_1=1, d_2=4 y d_3=6 ctmos.

	0	1	2	3	4	5	6	7	8
d_1	0	1	2	3	4	5	6	7	8
d_2	0	1	2	3	1	2	3	4	2
d_3	0	1	2	3	1	2	1	2	2

Diseño del problema del cambio de monedas: Implementación

- Proyecto Code::Blocks CambioMonedas

```
10  Solucion AlgoritmoProgDinCambioMonedas(const Problema & p) {
11
12      Solucion sol;
13      int **T;
14
15      T= new int *[ p.getN() ];
16      for (int i= 0; i<p.getN(); i++)
17          T[i]= new int [ (int)p.getCantidadDevolver()+1 ];
18
19      // Caso base
20      for (int i= 0; i<p.getN(); i++)
21          T[i][0]= 0;
22
23      // Suponiendo que el sistema monetario tiene monedas de unidad
24      for (int j= 1; j<=p.getCantidadDevolver(); j++)
25          T[0][j]= j;
26
27      for (int i= 1; i<p.getN(); i++)
28          for (int j=1; j<=p.getCantidadDevolver(); j++) {
29
30              int aux;
31
32              if ( j >= p.getValorTipo(i) ) {
33                  aux= 1+T[i][j-(int)p.getValorTipo(i)];
34              } else
35                  aux= INFINITO;
36
37
38              if (T[i-1][j] < aux) {
39
40                  T[i][j]= T[i-1][j];
41
42              } else {
43
44                  T[i][j]= aux;
45              }
46          }
```

Diseño del problema del cambio de monedas: Representación (II)

- ¿Cómo conocer qué monedas se deben devolver? Partiendo desde la solución viajar hacia atrás en la tabla a través de la recurrencia, comprobando qué camino se ha seguido hasta llegar a la misma.

- Ejemplo: Devolver 8 ctmos con monedas de d_1=1, d_2=4 y d_3=6 ctmos.

	0	1	2	3	4	5	6	7	8
d_1	0	1	2	3	4	5	6	7	8
d_2	**0**	1	2	3	**1**	2	3	4	**2**
d_3	0	1	2	3	1	2	1	2	**2**

Diseño del problema del cambio de monedas: Implementación (II)

- Proyecto Code::Blocks CambioMonedas

```
50      sol.setN(0);
51      double cambioActual= p.getCantidadDevolver();
52      int fila= p.getN()-1, columna= (int)p.getCantidadDevolver();
53      int MonedasInsertadas= 0;
54      Problema paux(p);
55
56      while (cambioActual > 0) {
57
58          if ((fila > 0 && T[fila][columna] == T[fila-1][columna]) ||
59              !paux.HayMonedasTipo(fila)) {
60
61              fila--;
62
63              if (fila < 0) {
64
65                  for (int i= 0; i<p.getN(); i++)
66                      delete [] T[i];
67                  delete [] T;
68                  sol.setN(0);
69                  return sol;
70              }
71
72          } else {
73
74              if (!paux.HayMonedasTipo(fila)) {
75
76                  for (int i= 0; i<p.getN(); i++)
77                      delete [] T[i];
78                  delete [] T;
79                  sol.setN(0);
80                  return sol;
81              }
82              sol+= p.getValorTipo(fila);
83
84              paux.QuitarMonedaTipo(fila);
85              MonedasInsertadas++;
86              cambioActual-= p.getValorTipo(fila);
87              columna= (int)cambioActual;
88          }
89      }
```

UNIVERSIDAD DE GRANADA

DECSAI

Programación dinámica

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Enunciado del problema

- Tenemos una mochila con una capacidad de peso máxima ***M***, y un conjunto de ***n*** objetos a transportar.

- Cada objeto ***i*** tiene un peso w_i y llevarlo supone un beneficio b_i.

- El problema consiste en seleccionar qué objetos incluir en la mochila de modo que se maximice el beneficio, sin superar la capacidad de la misma, sabiendo que los objetos son indivisibles en fracciones y sólo se puede seleccionar llevar el objeto completo o no llevarlo. Si llevamos el objeto **i**, entonces $\mathbf{x_i = 1}$. En caso contrario, $\mathbf{x_i = 0}$.

- **Formalmente:** $$\max_{x_i} \left\{ \sum_{i=1}^{n} b_i x_i \right\}$$

Sujeto a $$\sum_{i=1}^{n} w_i x_i \leq M$$

Diseño del problema de la mochila 0/1: Resolución por etapas

- Capacidad máxima de la mochila **M**
- Número de objetos **n=1, 2, ..., n**
- Beneficio de llevar cada objeto $\mathbf{b_i}$
- Peso de cada objeto $\mathbf{w_i}$
- Supondremos los objetos ordenados de **menor a mayor** $\mathbf{b_i/w_i}$
- Función objetivo: Maximizar $\sum_{i=1}^{n} x_i b_i$ sujeto a $\sum_{i=1}^{n} x_i w_i \leq M$
- $\mathbf{x_i}$ son variables para llevar (1) o no llevar (0) el objeto **i**
- **Llamaremos T(i,j) al beneficio de haber considerado llevar los objetos desde el 1 al i, sabiendo que la capacidad de la mochila es j.**
- **El problema se puede resolver por etapas:** En cada etapa **i** seleccionaremos llevar (o no) el objeto **i**, considerando la capacidad restante de la mochila. Si lo llevamos, restaremos su peso a **j**.

Diseño del problema de la mochila 0/1: Ecuación (I)

- **Objetivo**: Conocer **T(n,M) =** Beneficio máximo de llevar (o no llevar) objetos **desde el 1 hasta el n**, para una capacidad de mochila **M**.

- **Casos base**:
  - T(i, 0)= 0 No hay capacidad de mochila disponible, no hay beneficio.
  - T(1, 1..w_1)= 0; T(1, w_1 ..M)= b_1 Si sólo consideramos llevar el objeto 1, sólo podremos llevarlo para una capacidad de mochila igual o superior a su peso.
  - T(i,j)=-∞, cuando j<0 No se puede echar en la mochila un elemento que supere su peso. Es solución no válida y usaremos el valor -∞ para plantear las recurrencias.

Diseño del problema de la mochila 0/1: Ecuación (II)

- **Caso general**:

$$\mathbf{T[i][j]} = \max\{\mathbf{T[i-1][j]}, \mathbf{b_i + T[i-1][j-w_i]}\}$$

El caso general supone que para una capacidad de mochila **j** considerando echar (o no) los objetos **desde el tipo 1 hasta el i**, el máximo beneficio se conseguirá de una de las dos formas siguientes:

- No echar el objeto de tipo **i** y considerar echar sólo **desde 1 hasta i-1**: primera parte **T[i-1][j]**
- Echar el objeto de tipo **i** , restando su peso de la capacidad de la mochila **$j-w_i$** y aumentando el beneficio en **b_i**. Seguidamente, viendo si podemos echar (o no) objetos desde el tipo 1 hasta el tipo **i-1**: Segunda parte **b_i+T[i-1][j-w_i]**

Diseño del problema de la mochila 0/1: Verificación del P.O.B.

T[i][j] es óptimo si las decisiones tomadas anteriormente para **T[i-1][j]** y **T[i-1][j-w_i]** lo son.

- Para una capacidad **j** fija, T[1][j] es óptimo ya que, tanto si w_1<=j como si no, se selecciona llevar o no llevar el objeto con beneficio óptimo en la ecuación recurrente.
- Para T[2][j] también es óptimo, ya que sólo se seleccionará llevar un objeto (o ninguno) si la suma de ambos pesos es superior a j, o ambos si no lo son, por la ecuación recurrente.
- Se sigue por inducción hasta el caso T[i-1][j] y suponemos éste óptimo. Por reducción al absurdo, si no lo fuera existiría otra decisión óptima distinta de T[i-1][j] no considerada óptima, pero esto es imposible según la ecuación recurrente dada. Por tanto, T[i-1][j] es óptimo.
- Demostrado óptimo T[i-1][j], queda demostrado también para cualquier caso T[i-1][j-p_i], como caso particular de T[i-1][j].

Diseño del problema de la mochila 0/1: Representación

- Vamos a representar la solución al problema como una tabla T(i,j):
  - **Filas (i):** Asociadas a los objetos, ordenados ascendentemente por su valor $\mathbf{b_1/w_1 < b_2/w_2 < \ldots < b_n/w_n}$
  - **Columnas (j):** Capacidades de la mochila, desde 0 hasta **M**
  - **Celdas:** Cada celda T(i,j) contendrá el máximo beneficio que se puede obtener para una capacidad de mochila **j** asumiendo que consideramos llevarnos objetos (o no) desde el **tipo 1 hasta el i**.

Diseño del problema de la mochila 0/1: Representación

- Ejemplo:

n=5, b= {1, 6, 18, 22, 28}, w= {1, 2, 5, 6, 7}, y M= 11

	0	1	2	3	4	5	6	7	8	9	10	11
b=1 w=1	0	1	1	1	1	1	1	1	1	1	1	1
b=6 w=2	0	1	6	7	7	7	7	7	7	7	7	7
b=18 w=5	0	1	6	7	7	18	19	24	25	25	25	25
b=22 w=6	0	1	6	7	7	18	22	24	28	29	29	40
b=28 w=7	0	1	6	7	7	18	22	28	29	34	35	40

Diseño del problema de la mochila 0/1: Representación (III)

- Para saber cuáles son los objetos que se insertan en la mochila, basta con comparar **T[n][M]** con **T[n-1][M]**. Si estos valores son iguales, entonces el n-ésimo objeto no se ha insertado; en caso contrario, sí que se ha insertado.

- Seguidamente, se compara T[n-1][M] con T[n-2][M] (si no se insertó el n-ésimo objeto) de igual forma, o T[n-1][M-p_n] con T[n-2][M-p_n] en el caso de que se insertase el n-ésimo objeto. Se realiza esta operación recursivamente hasta llegar al objeto 1 o una capacidad de mochila igual a 0 para conocer el resto de objetos.

UNIVERSIDAD DE GRANADA

DECSAI

Programación dinámica

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Enunciado del problema

Sea G=(V,A) un grafo **dirigido**, con arcos **ponderados con pesos no negativos**. El problema consiste en hallar el camino mínimo (secuencia de arcos que unen un nodo inicial y un nodo destino, cuya suma de pesos es mínima) **entre cualquier par de nodos** del grafo.

¡CUIDADO!

- **El enunciado no es el mismo que el estudiado en Greedy.**
- La solución equivale a aplicar el algoritmo de Dijkstra **n** veces, considerando cada vez como nodo de partida un nodo distinto del grafo.

Diseño del problema de caminos mínimos: Resolución por etapas

- Numeramos los nodos desde **1..n**.
- Asumimos que **D** es la matriz de adyacencia del grafo, y **D[i][j]** es la distancia para ir directos desde el nodo **i** al nodo **j**.
- **D[i][j]**= +∞ si no hay arco entre **i** y **j**.
- Llamaremos $\mathbf{D_k[i][j]}$= coste del camino mínimo entre **i** y **j**, con nodos intermedios en el conjunto **{1...k}**. Si k=0, no hay nodos intermedios.
- Función objetivo: Minimizar $\mathbf{D_n[i][j]}$, con **n** el número de nodos del grafo, para todo **i** y todo **j**.
- **El problema es resoluble por etapas:** En cada etapa se considera pasar por un nodo intermedio ***k*** para cada par de nodos **i,j** de origen y destino.

Diseño del problema de caminos mínimos: Ecuación (I)

- **Objetivo**: Conocer **D_n[i][j] =** Coste mínimo para ir desde **i** hasta **j**, considerando pasar (o no) por cualquier nodo intermedio desde **1 hasta n**.

- **Caso base**:
  - D_0[i][j]= D[i][j] Camino de **i** a **j** sin pasar por ningún nodo intermedio= Matriz de adyacencia.

Diseño del problema de caminos mínimos: Ecuación (II)

- **Caso general**:

$$D_k[i][j] = \min\{D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j]\}$$

El caso general supone que para viajar desde **i** hasta **j**, pudiendo pasar (o no) por los nodos **1 a k** como intermedios, tiene dos posibilidades:

- Que el camino de **i** a **j** no pase por **k**. Primera parte $D_{k-1}[i][j]$
- Que el camino de **i** a **j** pase por **k**. Segunda parte:

$$D_{k-1}[i][k] + D_{k-1}[k][j]$$

Diseño del problema de caminos mínimos: Verificación del P.O.B.

- **D_0[i][j]** es óptimo: El mejor camino de **i** a **j** sin pasar por ningún nodo es **D[i][j]**. **D_1[i][j]** también es óptimo: Selecciona el mejor camino entre ir directos de **i** a **j** o pasando por el nodo 1.
  - ... **D_k[i][j]** es óptimo: En caso contrario, habría otros nodos en el camino de **i** a **j** pasando por **{1...k-1}** tal que su coste sea menor que el considerado. Esto es imposible, dado que la ecuación recurrente siempre selecciona el menor coste.

Diseño del problema de caminos mínimos: Representación

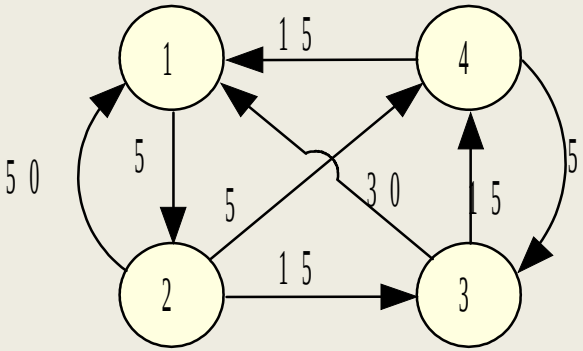
- Vamos a representar la solución al problema con dos tablas:
  - D(i,j), que va a contener la distancia mínima entre **i** y **j**.
  - P(i,j)=k, que representa que **k** es un nodo intermedio en el camino entre **i** y **j**. Por tanto, para recuperar el camino, tendremos que calcular también P(i,k) y P(k,j)

Diseño del problema de caminos mínimos: Implementación

- **Algoritmo de Floyd**:

```
Procedimiento Floyd(MatridAdy[1..n][1..n])
Para i=1 hasta n, hacer:
   Para j=1 hasta n, hacer:
      D[i][j]= MatrizAdy[i][j] // Long. del camino mínimo conocido inicial
      P[i][j]= 0 // Para ir desde i a j inicialmente no se pasa por ningún nodo
   Fin-Para
Fin-Para
Para k=1 hasta n, hacer:
   Para i=1 hasta n, hacer:
      Para i=1 hasta n, hacer:
         Si D[i][j]> D[i][k]+D[k][j], entonces:
            D[i][j]= D[i][k]+D[k][j]
            P[i][j]= k
         Fin-Si
      Fin-Para
   Fin-Para
Fin-Para
Devolver D,P
```

Ejemplo del problema de caminos mínimos



$$D_0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Solución:

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

1. Características de la programación dinámica
2. La técnica de programación dinámica
3. El problema del cambio de monedas
4. El problema de la mochila 0/1
5. Caminos mínimos
6. Multiplicación encadenada de matrices

Introducción al problema

- Sean 2 matrices $A_{n,m}$ y $B_{m,p}$. Su producto es $C_{n,p}$= A*B.
- Cada componente de la matriz C, c_{ij}, se calcula:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj}$$

- **Operaciones**: Hay que calcular n*p coeficientes c_{ij}, y para calcular cada coeficiente es necesario hacer m multiplicaciones. Total: n*p*m.

Enunciado del problema

- Tenemos varias matrices a multiplicar en un orden dado, G= A*B*C*D*...
- Por la propiedad asociativa, podemos decidir realizar antes unas multiplicaciones que otras. Por ejemplo:
  - Sean las matrices $A_{3,100}$, $B_{100,5}$, $C_{5,5}$, y queremos calcular G=A*B*C
  - Podemos parentizar de 2 formas diferentes:
    - (A*B)*C requiere 1500+75= 1575 operaciones
    - A*(B*C) requiere 1500+2500= 4000 operaciones

- **¿Cuál es la parentización óptima que debemos hacer para realizar el mínimo número de operaciones posible en total, para realizar la multiplicación de todas las matrices?**

- El número posible de parentizaciones que se pueden probar es de $\mathbf{4^n/n^2}$, donde n es el número de matrices a multiplicar, si comprobamos todas las posibilidades. **Con P.D. $O(n^3)$**

Diseño de la multiplicación encadenada de matrices: Resolución por etapas

- Sean **n** matrices a multiplicar $M_1*M_2*...*M_n$
- Llamamos **N(i,j)** al número mínimo de operaciones necesarias para multiplicar todas las matrices entre las posiciones **i** y **j**: $M_i*M_{i+1}*...*M_j$
- Supongamos ahora que parentizamos en una matriz **k**, entre **i** y **j.**

$$(M_i*M_{i+1}*...*M_k) * (M_{k+1}*M_{k+2}*...*M_j)$$

- ¿Cuál es el número de operaciones en este caso?

$$\mathbf{N(i,j)= N(i,k)+N(k+1,j)+p_{i-1}*p_k*p_j}$$

Llamamos $\mathbf{p_i}$**= número de columnas de la matriz i=número de filas de la matriz i+1.**

$\mathbf{p_{i-1}*p_k*p_j}$ es el número de operaciones requerido para multiplicar las 2 matrices resultantes de la parentización.

- **El problema se puede resolver por etapas**. En cada etapa, comprobaremos cuál es el **k** óptimo para multiplicar las matrices en las posiciones **i,j**, de acuerdo a la ecuación anterior.

Diseño de la multiplicación encadenada de matrices: Ecuación (I)

- **Objetivo**: Conocer **N(1,n) =** Mínimo número de operaciones para multiplicar las matrices **desde la 1 hasta la n**.

- **Caso base**:
  - Cuando i=j, N(i,j)= 0 Sólo hay 1 matriz, no hay que multiplicar.

Diseño de la multiplicación encadenada de matrices: Ecuación (II)

Caso general:

$$N(i,j)= \min_{i<=k<j}\{N(i,k)+N(k+1,j)+p_{i-1}*p_k*p_j\}$$

El caso general supone que, para multiplicar las matrices entre la posición **i** y la posición **j**, hay que encontrar una posición central que haga que multiplicar desde **i** hasta **k** por desde **k+1** hasta **j** sea óptimo.

Diseño de la multiplicación encadenada de matrices: P.O.B.

- El caso base es óptimo. El caso de tamaño 1 también.
- Para resolver el caso de tamaño **n**, el coste de esta solución será óptimo si las subsoluciones N(i,k) y N(k+1,n) es óptimo.
- Necesariamente, ambas soluciones lo son. En caso contrario, existiría alguna otra solución que las sustituyese con menor número de operaciones, lo cual va en contra de la ecuación en recurrencias.

Diseño de la multiplicación encadenada de matrices: Representación

- Vamos a representar la solución al problema con dos tablas:
  - N(i,j), que va a contener el mínimo número de operaciones para multiplicar las matrices entre las posiciones **i** y **j**.
  - Solucion(i,j)=k, que representa que, para multiplicar las matrices entre las posiciones **i** y **j**, hay que multiplicar primero desde **i hasta k** y luego desde **k+1 hasta j**

Diseño de la multiplicación encadenada de matrices: Implementación

Procedimiento MultiplicacionEncadenada(p, n)

```
Para i= 1 hasta n
    N[i,i] = 0
Para l = 2 hasta n
    Para i = 1 hasta n-l+1
        j=i+l-1
        N[i,j] = inf.
        Para k=i hasta j-1
            q=N[i,k] + N[k+1,j] + p_{i-1}p_k p_j
            Si q < N[i,j]
                N[i,j] = q
                Solucion[i,j] = k
            Fin-Si
        Fin-Para-k
    Fin-Para-i
Fin-Para-l
Devolver Solucion, N[1,n]
```

Ejemplo de multiplicación encadenada de matrices

- Hacer una traza del algoritmo para la multiplicación de n=4 matrices, cuyos tamaños son: 10x20, 20x50, 50x1, 1x100)
- Matriz de solución para N:

	j= 1	2	3	4
i=1	0	10.000	1.200	2.200
2		0	1.000	3.000
3			0	5.000
4				0

- Matriz de solución para Solucion (mejor k):

Mejork	j= 1	2	3	4
i=1	-	1	1	3
2		-	2	3
3			-	3
4				-

UNIVERSIDAD DE GRANADA

DECSAI

Algorítmica

Grado en Ingeniería Informática

Tema 4 – Programación dinámica

Manuel Pegalajar Cuéllar

manupc@ugr.es

Departamento de Ciencias de la Computación e Inteligencia Artificial

http://decsai.ugr.es