Estructuras de datos y algoritmos (2): recursión y arboles

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

Outline

Previously en la clase

- Las recursiones y la estructura de arboles están fundamentalmente ligadas
- La estrategia Divide and Conquer es un primer tipo de algoritmo recursivo adaptado al caso de problemas separables en sub-estancias que forman una partición del problema inicial
- Lleva a formulas recursivas en cuanto a la complejidad
- Otra clase de algoritmos recursivos: programación dinámica

Outline

Problemas no independientes. . .

La llave de la eficiencia de Divide And Conquer es que los sub-problemas en que se descompone el problema son independientes!

Si no es el caso, un esquema DaC puede ser muy ineficiente!
Ejemplo: números de Fibonacci

```
int F(int i) {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
  }
```

Complejidad? Por qué?

Problemas no independientes. . .

$$\begin{cases} T_0 & = & 1 \\ T_1 & = & 1 \\ T_n & = & T_{n-1} + T_{n-2} + 1 \end{cases}$$

Comportamiento asintótico: $(\frac{1+\sqrt{5}}{2})^n$

Cada llamada a un calculo $F(i)$ esta hecho dos veces: por $F(i+1)$ y $F(i+2)$

Problemas no independientes...

En cambio, se puede hacer algo muy simple y mucho mas eficiente:

```
F[0]=0;
F[1]=1;
for (int i=2;i<=N;i++)
        F[i]=F[i-1]+F[i-2];
```

de complejidad lineal!

- No requiere tanto espacio (en el caso de Fibonacci, el valor F_{45} es el maximo representable sobre 32 bits)
- Eventualmente, se puede usar solo los dos valores previos
- Manera de aliviar redundancia algorítmica usando recursos en memoria: programación dinámica

Problemas no independientes. . .

Versión de $O(1)$ en memoria

```
a=0;
b=1;
for (int i=2;i<=N;i++) {
        c=a+b;
        a=b;
        b=c;
}
```

Programación dinámica

- Caso previo: programación dinámica *bottom-up* (se calcula todos los valores)
- Integrando el concepto en un esquema recursivo (sin manipulacion explicita de los valores previos) da la programación dinámica *top-down*; se salva el valor calculado

```
int F(int i){
  static int knownF[maxN];
  if (knownF[i] != 0) return knownF[i];
  int t = i;
  if (i < 0) return 0;
  if (i > 1) t = F(i-1) + F(i-2);
  return knownF[i] = t;
}
```

Memoization

Programación dinámica

Dibujar arbol de llamadas para calcular F_8 por DaC y Programación dinámica. . . manera de cortar ramas del árbol

Técnica posible para la clase de problemas implicando recurrencia (seguidas de enteros calculados por una formula recursiva) y cuyos elementos previos en la recurrencia se "intersectan"

Programación dinámica: C_k^n

Los coeficientes binomiales dan otro ejemplo clásico:

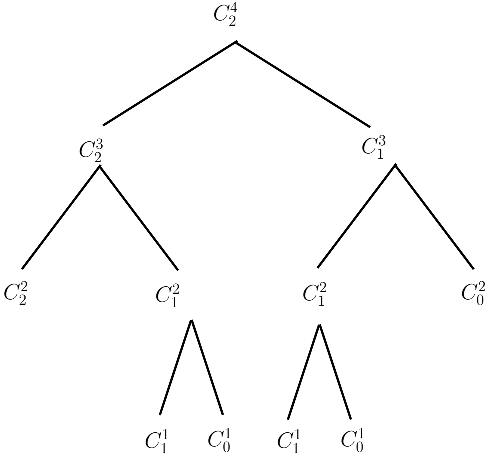
$$\begin{cases} C_0^n & = & 1 \\ C_n^n & = & 1 \\ C_k^n & = & C_k^{n-1} + C_{k-1}^{n-1} \end{cases}$$

Formulación recursiva:

```
int C(unsigned int k,unsigned int n)  {
    if (k< 1) return 1;
    if (k==n) return 1;
    return C(k-1,n-1) + C(k,n-1);
  }
```

Programación dinámica: C_k^n

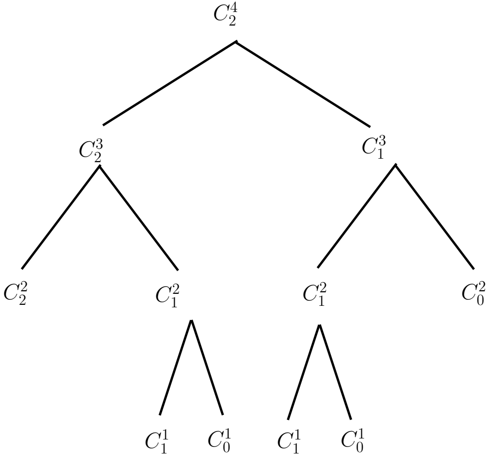
Arbol de llamadas:



Cuantas llamadas?

Programación dinámica: C_k^n

Arbol de llamadas:



Cuantas llamadas? C_k^n !, ya que el árbol de llamadas tiene la misma estructura que el calculo sí mismo

Programación dinámica: C_k^n

Ahora, no hay mas que $\frac{n(n+1)}{2}$ valores! Triángulo de Pascal:

C_0^0				
C_0^1	C_1^1			
C_0^2	C_1^2	C_2^2		
C_0^3	C_1^3	C_2^3	C_3^3	
C_0^4	C_1^4	C_2^4	C_3^4	C_4^4

Programación dinámica: C_k^n

Idem que para Fibonacci, pero con 2 dimensiones:

- *Bottom-up*, se construye explícitamente todos los elementos de la tabla del triangulo

```
int C[n+1][n+1];
for (int i=0;i<=n;i++) {
    C[0][i]=1;
    C[i][i]=1;
}
for (int i=0;i<=n;i++)
   for (int j=1;j<i;j++) {
      C[j][i] = C[j][i-1] + C[j-1][i-1];
   }
```

- *Top-down*: con la formulación de función recursiva pero con test sobre un Cknown

Programación dinámica: C_k^n

Complejidad:

- En tiempo, en $O(n^2)$ (o $O(nk)$ para el único C_k^n)
- En espacio memoria, en $O(n^2)$, que puede estar reducida en $O(n)$

Mucho mejor que el DaC!

DP = recursion + caching

Programación dinámica: C_k^n

Complejidad:

- En tiempo, en $O(n^2)$ (o $O(nk)$ para el único C_k^n)
- En espacio memoria, en $O(n^2)$, que puede estar reducida en $O(n)$ (guardando la ultima linea solo, y no todo el arreglo)

Mucho mejor que el DaC!

DP = recursion + caching

Programación dinámica: la mochila

El problema de la mochila:

- Somos ladrones y hemos entrado en la casa de un rico ex-presidente
- Tenemos una mochila de capacidad M
- Podemos llevar objetos entre una collección de N objetos A, B, C...
- Cada objeto tiene un volumen *size* y una valor *val*

Como optimizar el valor de la mochila rellenada?

Programación dinámica: la mochila

Otra manera de verlo: maximizar, con un conjunto de objetos $i = 1..N$ de valor v_i, tamaño t_i:

$$\sum_{i=1}^{N} v_i x_i$$

con $x_i \in \{0, 1\}$, bajo la restricción:

$$\sum_{i=1}^{N} t_i x_i \leq M$$

donde M es un umbral. Problema probado como NP

Programación dinámica: la mochila

La llave de la resolución: para construir una solución optima, si considero uno de los objetos i_0 que pongo en la mochila, y si esta elección es buena, solo me falta usar la misma función de optimización sobre una mochila amputada del volumen del objeto puesto

Prueba por contradicción: suponer que hay combinación optima para el problema a $N-1$ objetos, capacitad $M-t_{i_0}$ que sea mejor que la derivada de la solución optima del problema (N, M) entonces la combinación entre ésta y el objeto i_0 da una mejor solución al problema global y satisface la restricción!

Programación dinámica: la mochila

Se dice que el problema tiene propiedad de sub-estructura óptima: una solución óptima al problema con n variables puede estar deducida de la solución del problema a $n-1$ variables.

Programación dinámica: la mochila

En este caso, si nos damos un numero de objetos i y una capacidad m, entonces, la solución $S(i, m)$ es la que lleva el máximo entre:

- o la solución del problema a $i-1$ objetos y capacidad m, $S(i-1, m)$, con $x_i = 0$,
- o la solución del problema a $i-1$ objetos y capacidad $m - t_i$, $S(i-1, m-t_i)$, con $x_i = 1$.

Nos incumbe calcular $S(N, M)$.

Programación dinámica: la mochila

Versión recursiva:

```
int knap(int n,int cap) {
    if (n==0 || cap==0)
        return 0;
    int space,t;
    int max=knap(n-1,cap);
    if ((space = cap-items[n-1].size) >= 0)
        if ((t=knap(n-1,space)+items[n-1].val)>max)
            max=t;
    return max;
}
```

Llamar $knap(N, M)$... Muchos cálculos están hechos varias veces!
Claramente da una complejidad exponencial...

Programación dinámica: la mochila

Organización de una versión PD: $(5, 2)$, $(4, 3)$, $(1, 2)$, $(3, 4)$, $(6, 6)$ para $M = 10$

M

N

0	0	0	0	0	0	0	0	0	0	0
0										
0										
0										
0										
0										

Programación dinámica: la mochila

Organización de una versión PD: $(5,2)$, $(4,3)$, $(1,2)$, $(3,4)$, $(6,6)$ para $M = 10$

M

	0	0	0	0	0	0	0	0	0	0	0
	0	0	5	5	5	5	5	5	5	5	5
N	0										
	0										
	0										
	0										

Programación dinámica: la mochila

Organización de una versión PD: $(5,2)$, $(4,3)$, $(1,2)$, $(3,4)$, $(6,6)$ para $M = 10$

M

	0	0	0	0	0	0	0	0	0	0	0
	0	0	5	5	5	5	5	5	5	5	5
N	0	0	5	5	5	9	9	9	9	9	9
	0										
	0										
	0										

Programación dinámica: la mochila

Organización de una versión PD: $(5,2)$, $(4,3)$, $(1,2)$, $(3,4)$, $(6,6)$ para $M = 10$

M

	0	0	0	0	0	0	0	0	0	0	0
	0	0	5	5	5	5	5	5	5	5	5
N	0	0	5	5	5	9	9	9	9	9	9
	0	0	5	5	6	9	9	10	10	10	10
	0										
	0										

Programación dinámica: la mochila

Organización de una versión PD: (5, 2), (4, 3), (1, 2), (3, 4), (6, 6) para $M = 10$

M

	0	0	0	0	0	0	0	0	0	0	0
	0	0	5	5	5	5	5	5	5	5	5
N	0	0	5	5	5	9	9	9	9	9	9
	0	0	5	5	6	9	9	10	10	10	10
	0	0	5	5	6	9	9	10	10	12	12
	0										

Programación dinámica: la mochila

Organización de una versión PD: $(5,2)$, $(4,3)$, $(1,2)$, $(3,4)$, $(6,6)$ para $M = 10$

M

N

0	0	0	0	0	0	0	0	0	0	0
0	0	5	5	5	5	5	5	5	5	5
0	0	5	5	5	9	9	9	9	9	9
0	0	5	5	6	9	9	10	10	10	10
0	0	5	5	6	9	9	10	10	12	12
0	0	5	5	6	9	9	10	11	11	12

Programación dinámica: la mochila

Organización de una versión PD: (5, 2), (4, 3), (1, 2), (3, 4), (6, 6) para $M = 10$

M

	0	0	0	0	0	0	0	0	0	0	0
	0	0	5	5	5	5	5	5	5	5	5
N	0	0	5	5	5	9	9	9	9	9	9
	0	0	5	5	6	9	9	10	10	10	10
	0	0	5	5	6	9	9	10	10	12	12
	0	0	5	5	6	9	9	10	11	11	12

Programación dinámica: la mochila

Solución 1, DP *bottom-up*

```
int maxValues[N+1][M+1],space,t;
for (int i=0;i<=N;i++) maxValues[i][0]=0;
for (int j=0;j<=M;j++) maxValues[0][j]=0;
for (int i=1;i<=N;i++)
  for (int j=1;j<=M;j++) {
    maxValues[i][j] = maxValues[i-1][j];
    if ((space = j - items[i].space)>=0)
      if ((t=maxValues[i-1][space]+items[i].val)>max)
        maxValues[i][j] = t;
  }
```

Programación dinámica: la mochila

DP *top-down*

```
int knownValues[N+1][M+1]; // Init to -1

int knap(int n,int cap) {
    if (n==0 || cap==0)
        return 0;
    if (knownValues[n][cap]>=0)
        return knownValues[n][cap];
    int space,t;
    int max=knap(n-1,cap);
    if ((space = cap-items[n-1].size) >= 0)
        if ((t=knap(n-1,space)+items[n-1].val)>max)
            max=t;
    return (knownValues[n][cap]=max);
}
```

Programación dinámica: la mochila

Otra versión: una solución recursiva, con recorrido horizontal

```
int knap(int cap) {
    int i, space, max, t;
    for (i = 0, max = 0; i < N; i++)
      if ((space = cap-items[i].size) >= 0)
        if ((t = knap(space) + items[i].val) > max)
          max = t;
    return max;
  }
```

¿Resuelve el mismo problema?

Programación dinámica: la mochila

Otra versión: versión programación dinámica:

```
int knap(int M) {
   int i, space, max, maxi = 0, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
      if ((space = M-items[i].size) >= 0)
        if ((t = knap(space) + items[i].val) > max) {
         max = t; maxi = i;
        }
    maxKnown[M] = max; itemKnown[M] = items[maxi];
    return max;
  }
```

Top-down o *bottom-up*?

Programación dinámica: la mochila

Otra versión: En la versión bottom-up, se hubiera calculado uno a uno todos los elementos de *maxKnown*[], llegando a una complejidad similar. Se puede preferir la otra:

- no hacemos necesariamente todos los calculos (arreglo disperso)
- se expresa mas naturalmente

Programación dinámica: la mochila

Complejidad ?

- En el peor de los casos se va a tener que rellenar un arreglo 2D $N \times M$ (solución 1) o encontrar M maxima (solución 2) lo que nos lleva : $O(NM)$
- Polinomial? Entonces no es NP?

Cuidado a que M no es un tamaño del input, es un número! Lo que constituye una indice del tamaño del input seria mas bien el numero de bits necesario para representar M! Ver también que la complejidad en espacio memoria no es despreciable. . .

Programación dinámica: la mochila

Complejidad ?

- En el peor de los casos se va a tener que rellenar un arreglo 2D $N \times M$ (solución 1) o encontrar M maxima (solución 2) lo que nos lleva : $O(NM)$
- Polinomial? Entonces no es NP?

Cuidado a que M no es un tamaño del input, es un número! Lo que constituye una indice del tamaño del input seria mas bien el numero de bits necesario para representar M! Ver también que la complejidad en espacio memoria no es despreciable. . .

Programación dinámica: la mochila

Complejidad ?

- En el peor de los casos se va a tener que rellenar un arreglo 2D $N \times M$ (solución 1) o encontrar M maxima (solución 2) lo que nos lleva : $O(NM)$
- Polinomial? Entonces no es NP?

Cuidado a que M no es un tamaño del input, es un número! Lo que constituye una indice del tamaño del input seria mas bien el numero de bits necesario para representar M! Ver también que la complejidad en espacio memoria no es despreciable...

Programación dinámica: la mochila

Propiedad: en regla general, se puede mostrar que la programación dinámica permite reducir la complejidad en tiempo de una función recursiva en, en el peor de los casos, el tiempo requerido para evaluar la función en todos los argumentos inferiores o iguales al argumento corriente (con llamadas en tiempo constante).

Programación dinámica: la mochila

Versión glotona: algoritmos que realicen una aproximación importante para llevar a un resultado que puede convenir, pero sin (en general) prueba de optimalidad (tienes optimalidad local al momento de elegir el próximo elemento de tu solución global pero no sabemos si lograra un optimum global)

Ejemplo: ordenar los objetos por una función de eficiencia (ratio valor/tamaño) y incorporarle cuando se puede recorriéndoles por orden decreciente de eficiencia

Programación dinámica: productos

El algoritmo estándar para multiplicar matrices $m \times n$ por $n \times p$ es de complejidad mnp, imagina ahora que queremos multiplicar varias matrices entre ellas $ABCD$, cual es lo mejor;

- (AB)(CD)
- A(BC)D
- A(B(CD))
- ...

y generalizar con matrices M_i de tamaño $d_i \times d_{i+1}$

Programación dinámica: productos

Expresión de la combinatoria, en DaC:

$$(M_1 \dots M_i)(M_{i+1} \dots M_n)$$

Entonces el numero de posibilidades resultantes es:

$$\begin{cases} T_1 & = & 1 \\ T_n & = & \sum_{i=1}^{n-1} T_i T_{n-i} \end{cases}$$

Son los números de Catalan, asintoticamente $\Omega(\frac{4^n}{n^2})$... explosivo!

Programación dinámica: productos

Buscaremos el numero mínimo de productos entre escalares necesarios para calcular $M_i \dots M_j$: p_{ij}

Suponemos que sabemos cual es la mejor manera de poner paréntesis en $M_i \dots M_j$:

$$(M_i \dots M_k)(M_{k+1} \dots M_j)$$

- Numero de productos para la matriz $d_i \times d_{k+1}$ $M_i \dots M_k$: p_{ik}
- Numero de productos para la matriz $d_{k+1} \times d_{j+1}$ $M_{k+1} \dots M_j$: $p_{(k+1)j}$
- Para la matriz resultado:

$$p_{ij} = p_{ik} + p_{(k+1)j} + d_i d_{k+1} d_{j+1}$$

Programación dinámica: productos

Luego, notar que para todo k: $p_{kk} = 0$

j

i		1	2	3	4	5
	1	0				
	2		0			
i	3			0		
	4				0	
	5					0

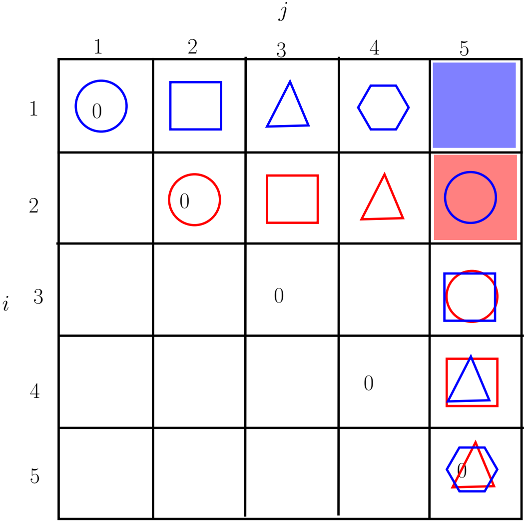
Programación dinámica: productos

Luego, notar que para todo k: $p_{kk} = 0$

i \ j	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

Programación dinámica: productos

Luego, notar que para todo k: $p_{kk} = 0$



Programación dinámica: productos

- Para calcular los elementos de una diagonal, solo se necesita elementos de las previas diagonales
- Si $d = j - i > 0$ representa la diagonal actual, se usa para calcular p_{ij} los elementos $(i, k), k < j$ y $(k + 1), j$
- Inicializar la diagonal $d = 0$, luego calcular diagonal por diagonal

Programación dinámica: productos

Ejemplo con 4 matrices: $d_1 = 13,\ d_2 = 5,\ d_3 = 89,\ d_4 = 3,\ d_5 = 34,$

i \ j	1	2	3	4
1	0			
2		0		
3			0	
4				0

Programación dinámica: productos

Ejemplo con 4 matrices: $d_1 = 13,\ d_2 = 5,\ d_3 = 89,\ d_4 = 3,\ d_5 = 34,$

$$
\begin{array}{cc|c|c|c|c|}
 & & & & j & \\
 & & 1 & 2 & 3 & 4 \\
\hline
 & 1 & 0 & 5785 & & \\
\hline
 & 2 & & 0 & 1335 & \\
\hline
i & 3 & & & 0 & 9078 \\
\hline
 & 4 & & & & 0 \\
\hline
\end{array}
$$

Programación dinámica: productos

Ejemplo con 4 matrices: $d_1 = 13$, $d_2 = 5$, $d_3 = 89$, $d_4 = 3$, $d_5 = 34$,

$i \backslash j$	1	2	3	4
1	0	5785	1530	
2		0	1335	1845
3			0	9078
4				0

Programación dinámica: productos

Ejemplo con 4 matrices: $d_1 = 13$, $d_2 = 5$, $d_3 = 89$, $d_4 = 3$, $d_5 = 34$,

$i \backslash j$	1	2	3	4
1	0	5785	1530	2856
2		0	1335	1845
3			0	9078
4				0

Programación dinámica: productos

Complejidad en tiempo:

- Para cada elemento, se necesita considerar $d = j - i$ posibilidades
- Cada diagonal tiene $n - d$ elementos
- En total:

$$T_n = \sum_{d=1}^{n-1} (d+2)(n-d)$$

- O sea $T_n \in \theta(n^3)$

Programación dinámica: productos

Para determinar la posición de todas las paréntesis:

```
int p[n-1][n-1];
int bestChoice[n-1][n-1];
for (int i=0;i<n;i++) p[i][i]=0;
for (int l=1;l<n;l++)
  for (int i=0;i<n-l;i++) {
        int j=i+l;
        p[i][j] = std::numeric_limits<int>::max();
        for (int k=i;k<j;k++) {
          int q=p[i][k] + p[k+1][j] + d[i]*d[k+1]*d[j
          if (q<p[i][j]) {
            p[i][j] = q;
            bestChoice[i][j]=k;
          }
        }
}
```

Programación dinámica: productos

Para usar el resultado:

```
void multiplySetOfMatrices(const Mat *matSet,
                           int i,int j,
                           int **best, Mat &result) {
if (i==j) {
       copy(matSet[i],result);
       return;
}
int k=best[i][j];
Mat left(d[i],d[k+1]),right(d[k+1],d[j+1]);
multiplySetOfMatrices(matSet,i,k,best,left);
multiplySetOfMatrices(matSet,k+1,j,best,right);
multiply(left,right,result);
}
```

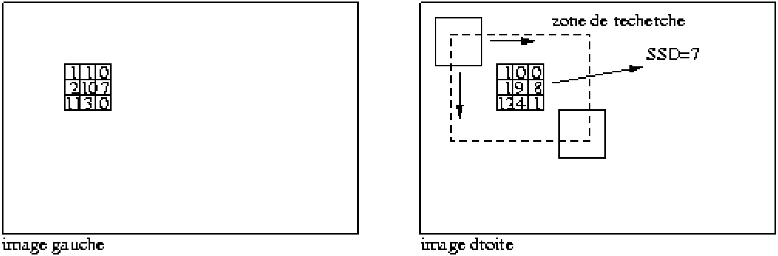
Programación dinámica: estéreo

- Dos imágenes sincronizadas de la misma escena están analizadas para inferir la profundidad
- Necesita sistema calibrado prealablemente: parámetros intrínsecos de las cámaras (focal,...) y transformación euclidiana entre las dos cámaras
- Establecimiento de correspondencias
- Reconstrucción 3D a partir de correspondencias discretas o buscando un mapa suave $\delta v(i, j)$
- Discretas:
    - Regiones
    - Segmentos
    - Puntos de interes

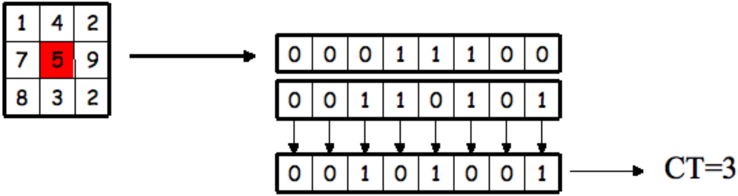
  Necesita mecanismos de interpolación...

Programación dinámica: estéreo

Programación dinámica: estéreo



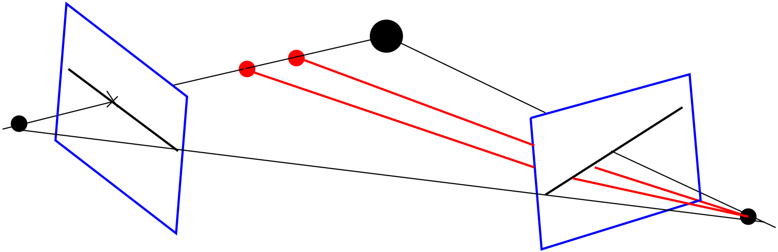
Correlación de patrón (correlación centrada, normalizada)



Census

Stereo: búsqueda de correspondencias

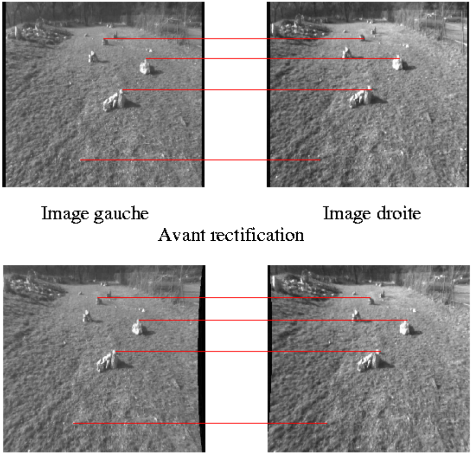
Por la geometria epipolar, la busqueda es solo 1D:



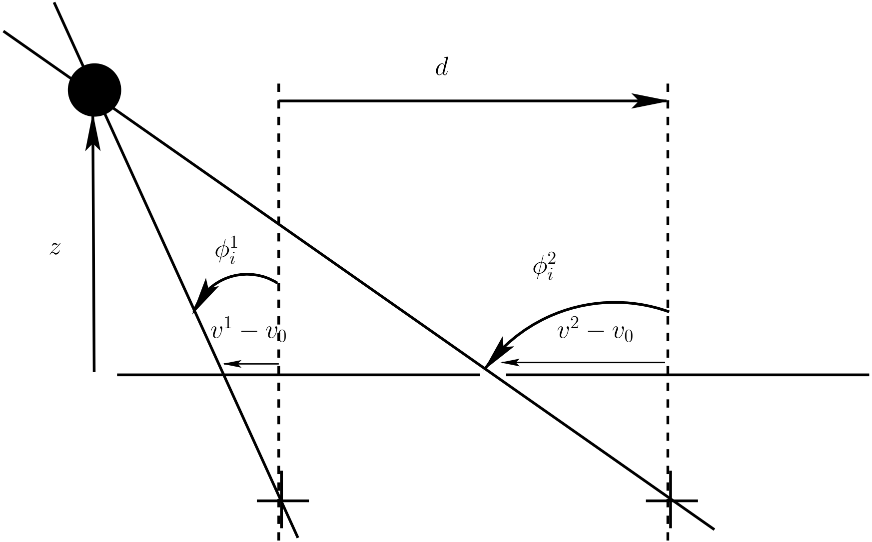
Pero la búsqueda no es eficiente si las lineas epipolares son cualquieras

Stereo: búsqueda de correspondencias

Existen un par de homografias H_1, H_2 que permiten transformar las dos vistas izquierda/derecha de tal manera que las lineas epipolares estén horizontales y de misma altura en la imagen (rectificación)!



Stereo: disparidad y profundidad



Stereo: disparidad y profundidad

Con simple trigonometría:

$$\begin{aligned} d &= (z_i + f^2)\tan\phi_i^2 - (z_i + f^1)\tan\phi_i^1 \\ &= (z_i + f^2)\frac{v^2 - v_0^2}{\alpha_v^2} - (z_i + f^1)\frac{v^1 - v_0^1}{\alpha_v^1} \end{aligned}$$

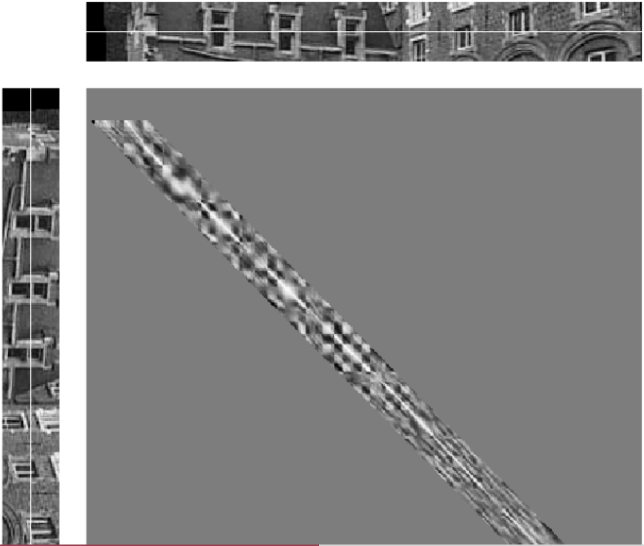
En el caso simple que las cámaras sean idénticas:

$$d = (z_i + f)\frac{v^2 - v^1}{\alpha_v}$$

O sea:

$$z_i = -f + \frac{\alpha_v d}{\delta v}$$

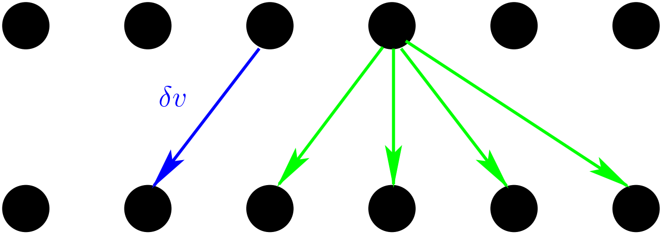
Programación dinámica: estéreo



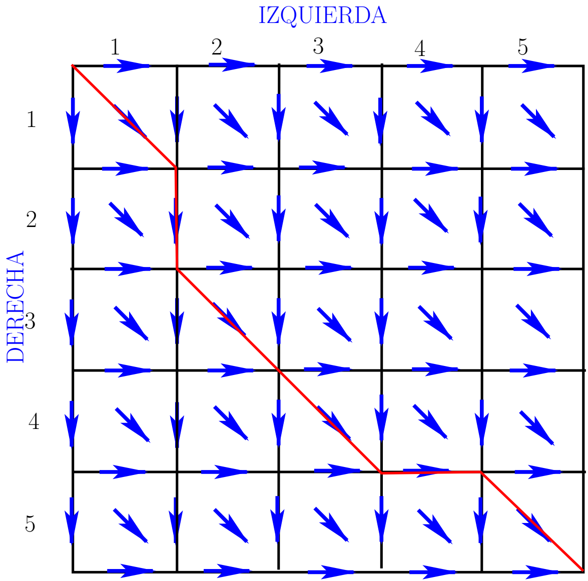
Programación dinámica: estéreo

Problema: buscar para cada linea de scan u un mapeo $\delta v(v)$ donde (u, v) es un pixel de la imagen izquierda, δv la disparidad, con las restricciones:

- los puntos de la imagen de izquierda pueden estar emparejado con nada (oclusión) o con uno y uno solo otro punto de la imagen derecha. Idem para la imagen de derecha
- el orden de los puntos originales y de sus emparejados tiene que estar igual!



Programación dinámica: estereo



Programación dinámica: estereo

Construcción de un mapa de costos mínimos para emparejar los pixels $1 \dots i$ de la imagen izquierda, y $1 \dots j$ de la imagen derecha:

$$C(i,j) = \min \left\{ \begin{array}{ccc} C(i-1,j) & + & c_o \\ C(i,j-1) & + & c_o \\ C(i-1,j-1) & + & \delta(i,j) \end{array} \right.$$

con condiciones iniciales:

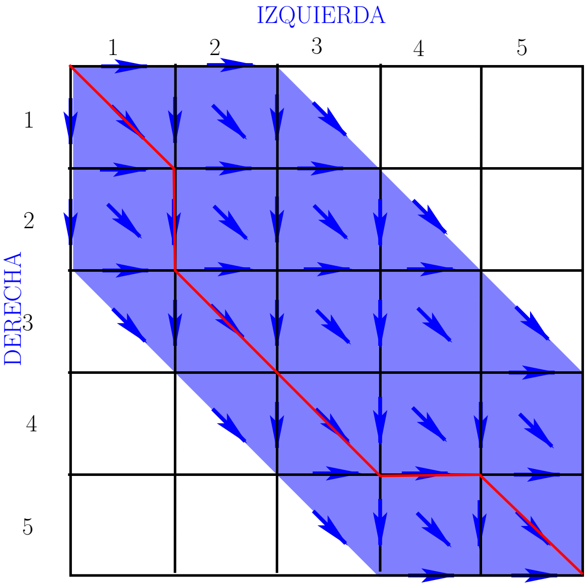
$$\begin{array}{rcl} C(i,1) & = & \delta(i,1) \\ C(1,j) & = & \delta(1,j) \end{array}$$

Programación dinámica: estereo

Complejidad:

- Algoritmo repasando todas las combinaciones: $O(3^N)$
- Tal que enseñado, $O(N^2)$ (rellenar un arreglo $N \times N$)
- En práctica, no se busca un camino dentro de $[\delta_v^1, \delta_v^2]$: linear !

Programación dinámica: estereo



En resumen

DP es ideal para problemas de optimización:

- Con decisiones consecutivas (proceso temporal)
- Con funciones de costo aditivas (eso justifica la sub-optimalidad)

Otras aplicaciones:

- Camino mas corto en un grafo (algoritmo de Floyd)
- Alineamiento de secuencias: muy importante hoy en biotecnologias (amino ácidos, ADN)
- ...