Programación dinámica: Presentación

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Programación dinámica

Metodología de resolución

de problemas de optimización (minimización o maximización)

Nombrada por Richard Bellman

En 1950 mientras trabajaba para la RAND Corporation (una historia interesante!)

Divide el problema en subproblemas

con una jerarquía entre ellos (de menor a mayor tamaño).

Cada subproblema

Puede ser utilizado ser reutilizado en diferentes subproblemas mayores

Propiedades

Un problema debe contener las siguientes propiedades

para poder resolverse de forma optima mediante un algoritmo programación dinámica

- **Subestructura óptima**
- **Subproblemas superpuestos**

Subestructura óptima

Un problema

Contiene una subestructura óptima

Si la solución optima global del mismo

Contiene en su interior las soluciones optimas de sus subproblemas

Subproblemas superpuestos

Un problema

Contiene una subproblemas superpuestos

Si en la resolución de sus subproblemas

Vuelven a aparecer subproblemas previamente calculados

Relación de recurrencia

Se puede resolver el problema recursivamente

Donde por cada problema se abrirán un conjunto de subproblemas

Utilizaremos una Ecuación de recurencia para representarlo

Cada término de la secuencia es definido como una función de términos anteriores

$$T_n = F(T_{n-1}, T_{n-2}, \ldots)$$

Existe uno o varios términos base o iniciales desde los cuales se calculan los siguientes.

Memorización

Técnica que consiste en

- En almacenar los resultados de los subproblemas previamente calculados

Para evitar repetir su resolución

- Cuando vuelva a requerirse

De esa forma reducen la cantidad total de subproblemas a calcular

- Consiguiendo reducir significativamente la complejidad temporal de la solución

Ejemplo: Corte de soga

Sea

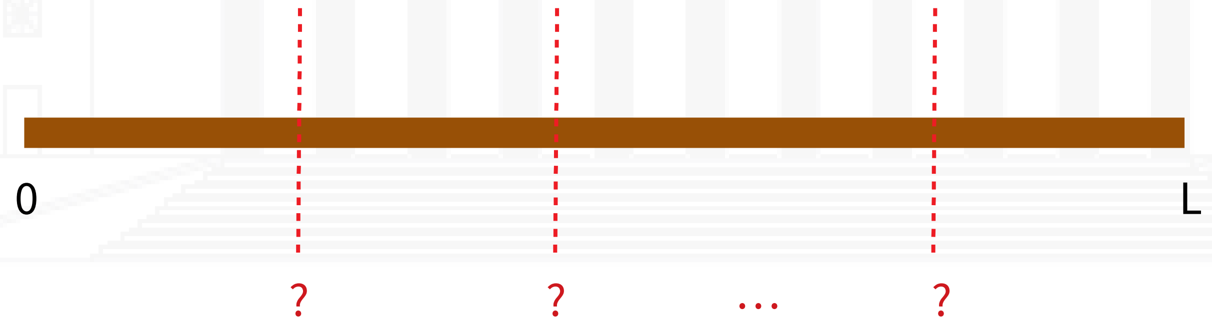
Una soga de longitud L divisible

Una tabla de precios por longitud de la soga

Long.	Gan.
1	p_1
2	p_2
3	p_3
4	p_4
5	p_5

Queremos

Saber que cortes realizar para maximizar la ganancia



Análisis del problema

Cada corte realizado de longitud l_i en la soga de longitud L

- Nos brindara una ganancia de p_i
- Dejara una nueva soga de longitud $L - l_i$ (un subproblema)

Debemos pensar como cortar la soga

- Podemos elegir con un corte inicial y luego continuar cortando con algún criterio

Existe una elección greedy válida?

- No!
- Debemos evaluar todos los corte posibles y elegir el máximo

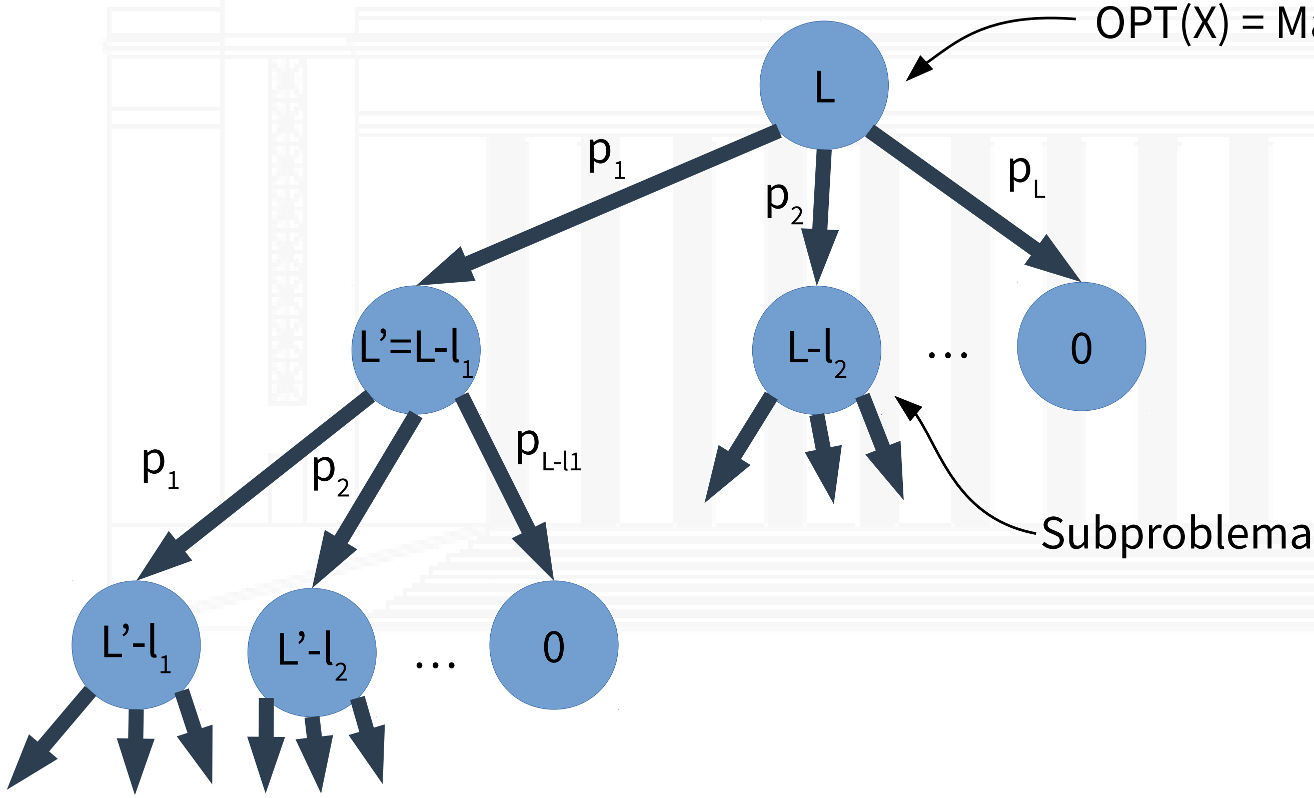
Árbol de decisión

Podemos representar en un árbol

Las diferentes elecciones a realizar

Podemos calcular la solución de un subproblema como:

OPT(X) = Max $_{i=1...n}$ {pi + OPT(X-li)}



Cada subproblema x nos dirá cuanto es lo máximo que se puede ganar con su longitud OPT(x)

Solución recursiva – relación de recurrencia

Expresamos cada subproblema de la forma

$$OPT(X) = Max_{i=1}^{X}\{p_i + OPT(X-i)\}, \qquad \text{si } X>0$$

El caso base

$$OPT(X) = 0, \qquad \text{si } X \leq 0$$

¿Cuantos subproblemas tengo que resolver?

Si revistamos completamente el árbol de decisión vemos que existen un numero exponencial de nodos (subproblemas).

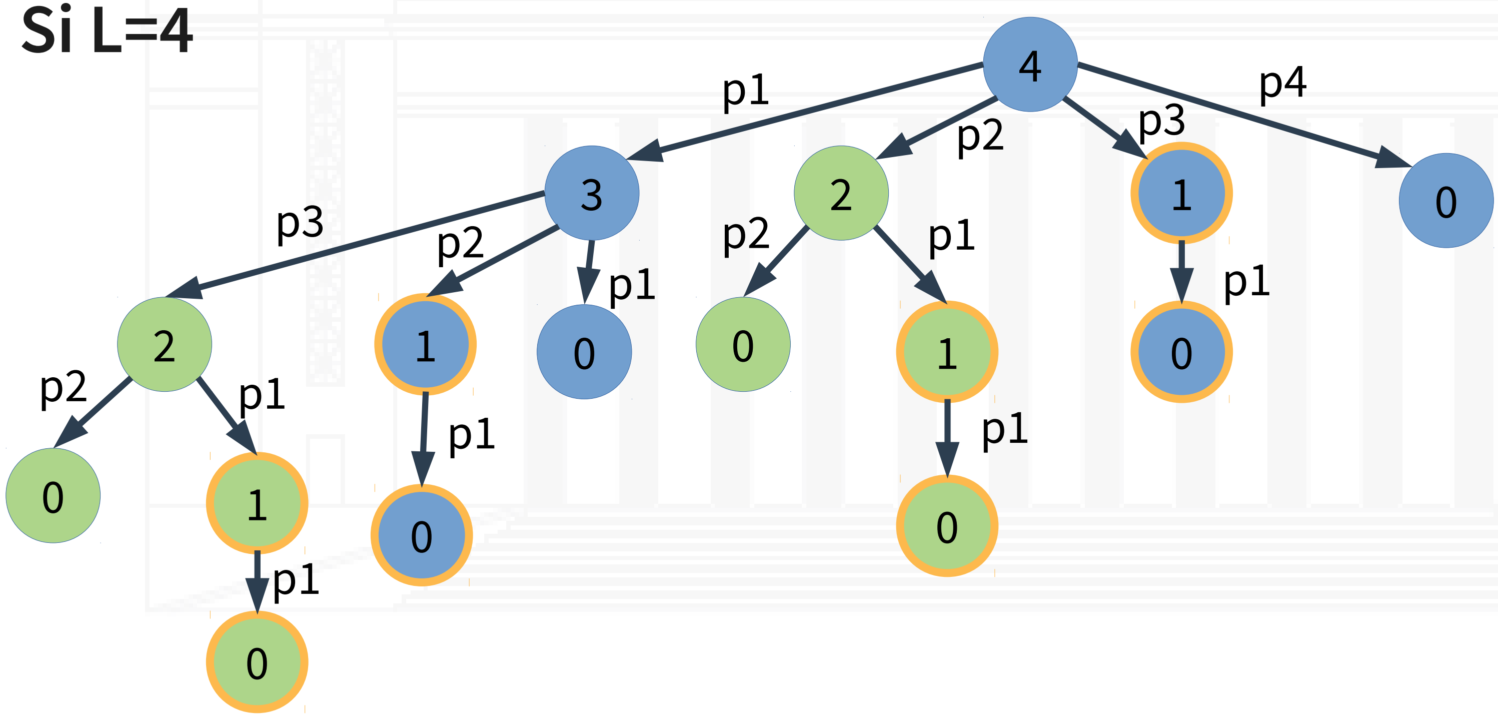
Sin embargo...

Subproblemas superpuestos

Podemos ver que ....

En nuestro árbol de decisión ciertos problemas se repiten.

Si L=4



El Subproblema L=1 se repite 4 veces
El Subproblema L=2 se repite 2 veces

Su resultado no cambia, sin importar por el camino que se llega al mismo

Memorización

Podemos

Calcular la primera vez el resultado y luego utilizarlo.

Almacenaremos en una tabla los suproblemas resueltos

OPT	Gan.
1	g_1
2	g_2
3	g_3
4	$\mathbf{g_4}$

Solo debemos calcular L subproblemas!

(en vez de un numero exponencial!)

La ganancia óptima

estará en la última fila de nuestra tabla de memorización

Solución iterativa

La expresión recursiva

Tiene como inconveniente que dificulta la memorización

(tenemos que recorrer el arbol de raiz a hojas)

Podemos resolver el problema

invirtiendo el orden de la resolución de los subproblemas

De los mas pequeños a los mas grandes,

En nuestro ejemplo:

OPT(0)=0

OPT(1)=p1

OPT(2)= max {p2 + OPT(0) ; p1 + OPT(1)}

…

OPT(L)=...

Pseudocódigo

```
OPT[0] = 0

Desde i=0 a L
     OPT[i] = 0
     Desde j=0 a i

          val = p[j] + OPT(i-j)

          si val>OPT[i]
               OPT[i]=val

Retornar OPT[L]
```

Complejidad espacial

Necesito guardar en la tabla L óptimos → O(L)

Complejidad temporal

Requiero calcular L suproblemas, y para cada uno de ellos evaluar sus subproblemas → $O(L^2)$

Si en hubiesen “c” cortes posibles la complejidad seria O(cL) → O(L)

Reconstrucción de las decisiones

Además de la ganancia

Seria muy útil saber como cortar la soga

Se puede conseguir

Almacenando para cada subproblema cuál corte fue el óptimo

Luego reconstruimos las elecciones partiendo del OPT(L) para atrás

```
OPT[0] = 0
Eleccion[0]=0

Desde i=0 a L
    OPT[i] = 0
    Desde j=0 a i

        val = p[j] + OPT(i-j)

        si val>OPT[i]
            OPT[i]=val
            Eleccion[i]=j

Imprimir OPT[L]

resto=L
Elegido = Eleccion[resto]
Mientras Elegido <> 0
    Imprimir Elegido
    resto = resto - Elegido
    Elegido =Eleccion[resto]
```

Presentación realizada en Octubre de 2020

Programación dinámica: Weighted Interval Scheduling

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Weighted Interval Scheduling

Sea

P un conjunto de n pedidos $\{p_1, p_2, \ldots, p_n\}$

Cada pedido i tiene

un tiempo s_i donde inicia

Un tiempo t_i donde finaliza

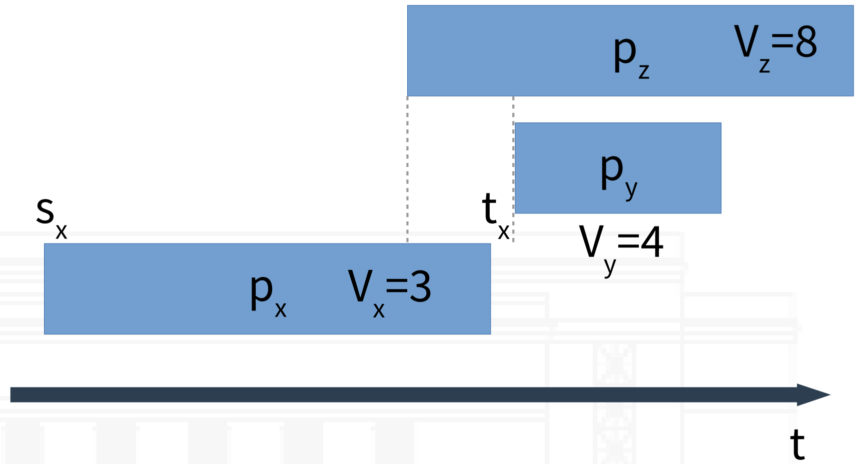
un valor v_i

Un par de tareas $p_x, p_y \in P$

Son compatibles entre si, si - y solo si – no hay solapamiento en el tiempo entre ellas

Queremos

Seleccionar el subconjunto P con tareas compatibles entre si y que con la suma de sus valores lo mayor posible



Si bien puedo seleccionar x e y con un valor de 7, es preferible z con un valor total de 8

¿Existe un algoritmo greedy para resolverlo?

Para el caso particular

$V_i = 1$ para todo i

Se corresponde

al problema de maximizar la cantidad de tareas compatibles a realizar

En ese caso

Funcionaba una estrategia greedy

Sin embargo

No se conoce una para el problema general

Un orden inicial

¿Qué hacíamos en la solución greedy?

Ordenábamos en orden creciente en tiempo de finalización de la tarea

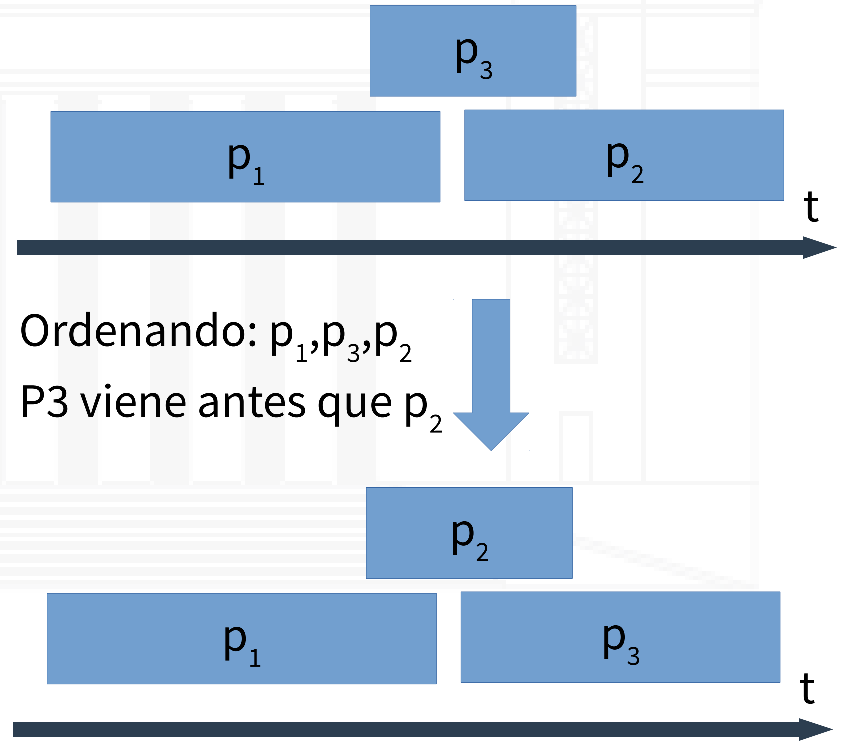
$f_1 \leq f_2 \leq \ldots \leq f_n$

Con este orden, podemos decir

Que la tarea i viene antes que la j, si i<j

Para construir nuestra solución

Utilizaremos el mismo ordenamiento



Tareas compatibles anteriores

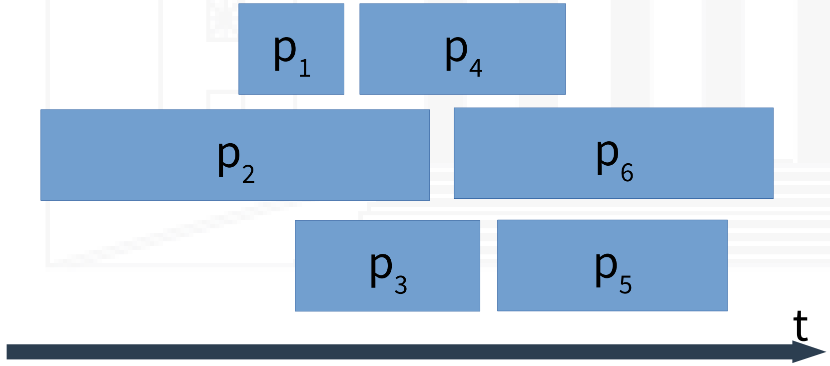
Para cada tarea i

Nos interesará conocer la primera tarea anterior con la que es compatible P(i)

Si nomenclamos las tareas utilizando el orden establecido

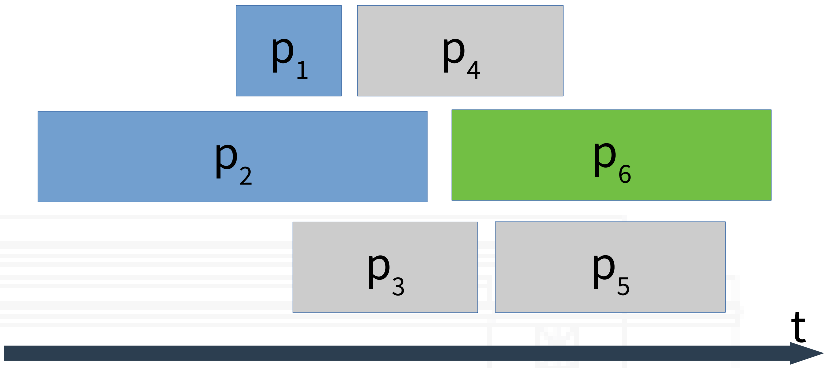
el indice de la tarea anterior compatible sera menor a la tarea → P(i)=x con x<i

Y todas las tareas con indice menor a x también serán compatibles



Tenemos:
P(6)=2 (y por lo tanto la tarea 1 también es compatible con la tarea 6)
P(5)= 3
P(3) = 0 (no hay ninguna tarea anterior compatible)

Pertenencia de una tarea al óptimo



Dada

Una instancia del problema

En la solución óptima O

Podrá pertenecer o no una determinada tarea i

Si pertenece

Entonces no podrán pertenecer aquellas incompatibles con ella

Recién P(i) podría pertenecer a la solución optima

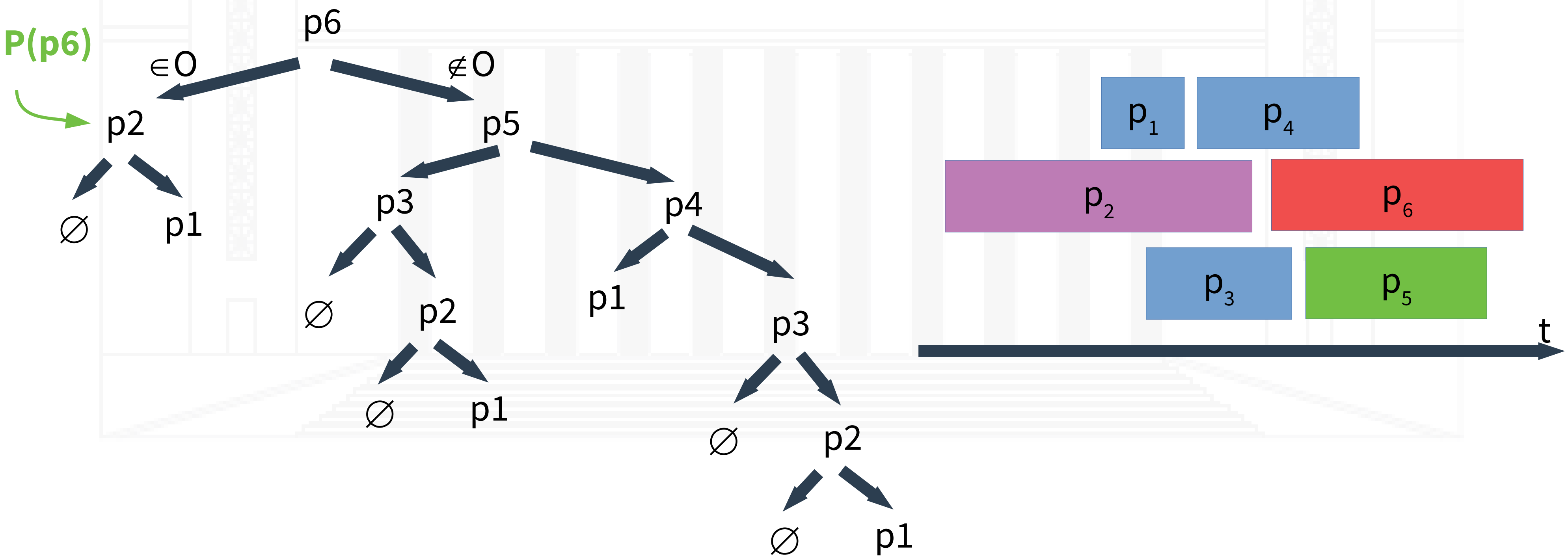
Si no pertenece

Entonces la tarea i-1 podría pertenecer a la solución optima

Árbol de decisión

Podemos utilizar este criterio

Comenzando por la tarea n y descendiendo hasta la tarea 1

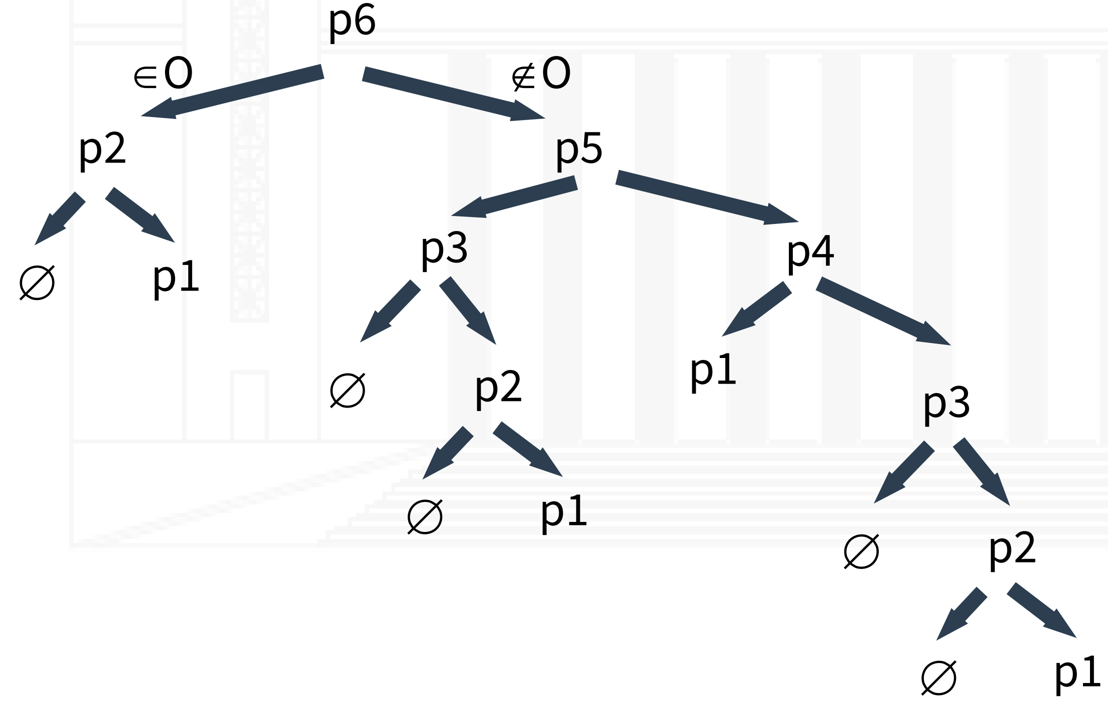


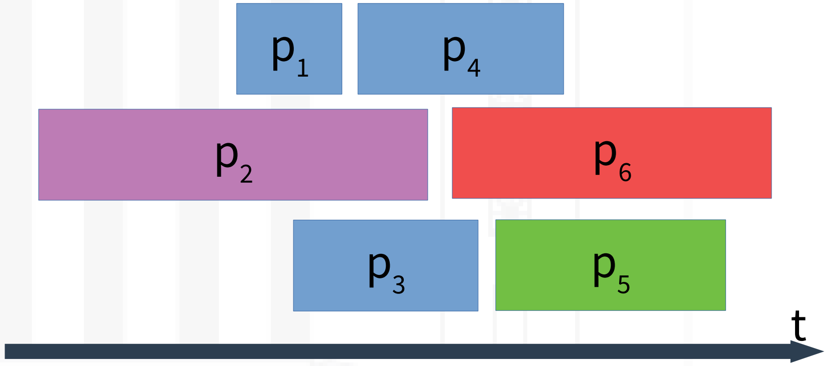
Tareas que pertenecen al óptimo (II)

¿Qué determina que una tarea i este o no en el óptimo?

Si conocemos el optimo de sus subproblemas: O(i-1) y O(p(i))

Elegimos el mayor valor entre v(pi)+O(i-1) y O(p(i))



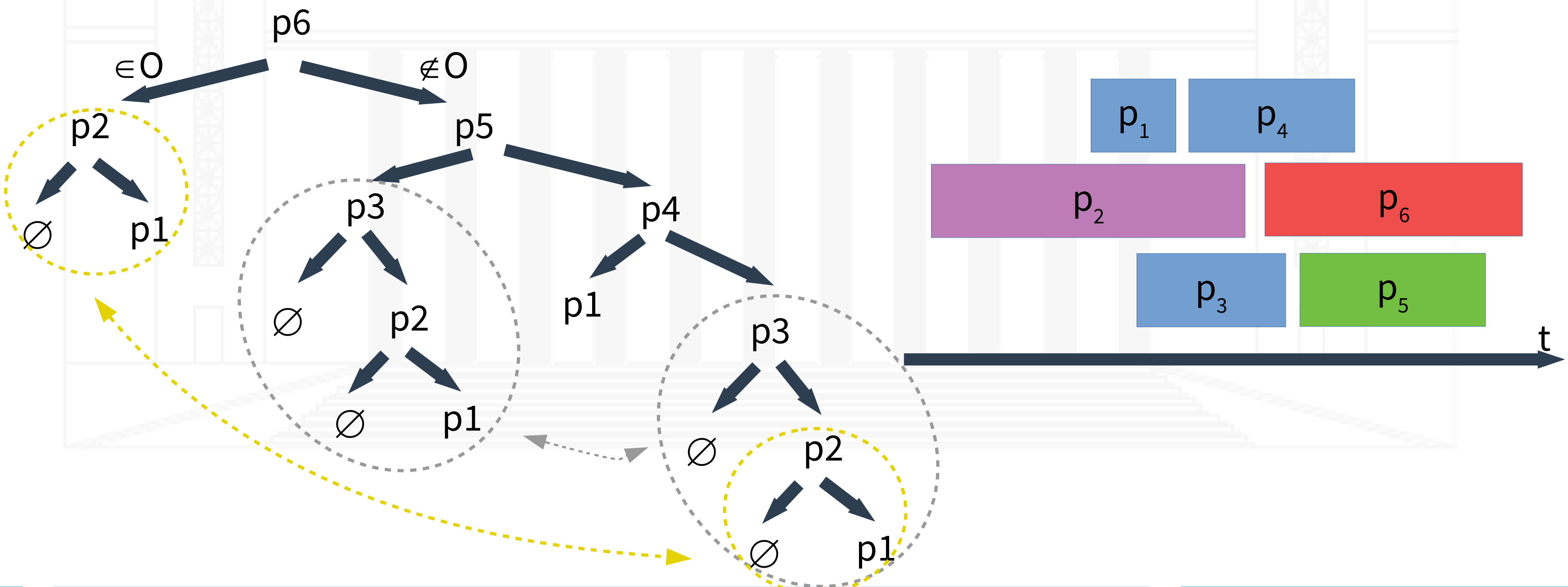


Memorización

¿Tenemos que recorrer (y calcular) todo el árbol?

Si observamos atentamente, algunos subproblemas se repiten

Alcanza con calcularlos solo 1 vez (aplicar memorización)



Recurrencia

Podemos expresar el problema como:

$$\begin{cases} OPT(x) = 0 & , si\ x = 0 \\ OPT(x) = max\{V(x) + OPT(P(x)), OPT(x-1)\} & , si\ X > 0 \end{cases}$$

El resultado con el máximo valor posibles será:

OPT(n)

Solución iterativa

Complejidad

Temporal O(n)

Espacial: O(n)

```
OPT[0]=0

Desde i=1 a n

    enOptimo = V[i] + OPT[P(i)]
    noEnOptimo = OPT[i-1]

    si enOptimo >= noEnOptimo
        OPT[i] = enOptimo
    sino
        OPT[i] = noEnOptimo

Retornar OPT[n]
```

Reconstruir las elecciones

Para cada subproblema i

almacenar si la tarea se eligió

Reconstruir para atrás

Partiendo de la tarea n

Iterar pasando por lo subproblemas seleccionados

```
OPT[0]=0
elegido[0]=false
Desde i=1 a n

    enOptimo = V[i] + OPT[P(i)]
    noEnOptimo = OPT[i-1]

    elegido[i]=(enOptimo >= noEnOptimo)
    si enOptimo >= noEnOptimo
        OPT[i] = enOptimo
    sino
        OPT[i] = noEnOptimo

Imprimir OPT[n]

i = n
Mientras i >0
    Si elegido[i]
        Imprimir i
        i = P(i)
    sino
        i--
```

Presentación realizada en Octubre de 2020

Programación dinámica: cambio mínimo

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Cambio mínimo (revisado y revisitado)

Contamos con

Un conjunto de monedas de diferente denominación sin restricción de cantidad

$\$ = (c_1, c_2, c_3, \ldots, C_n)$

Un importe X de cambio a dar

Queremos

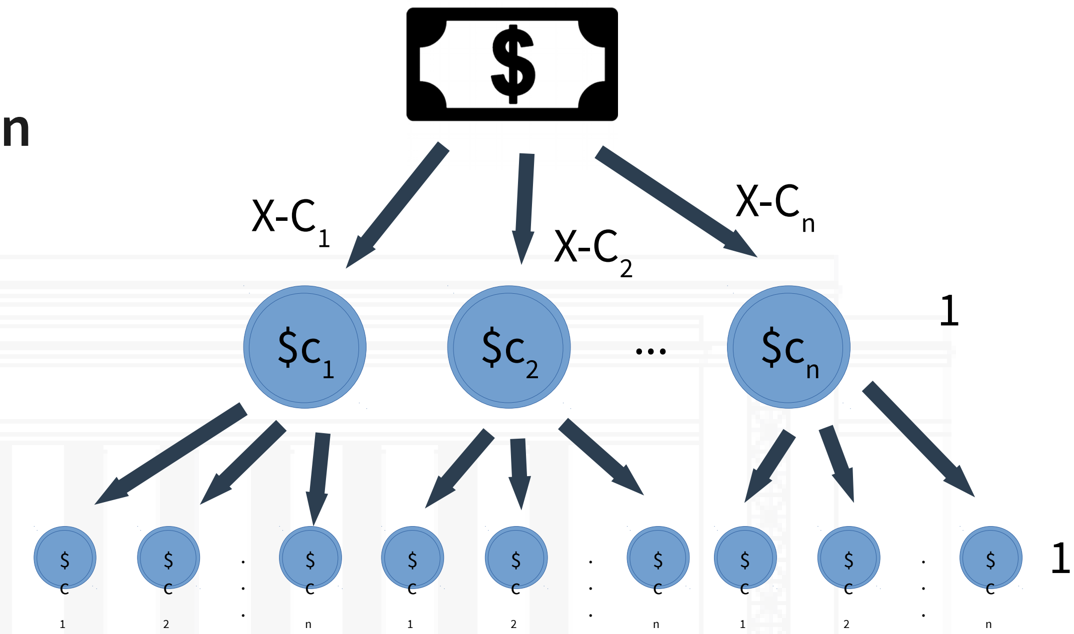
Entregar la menor cantidad posible de monedas como cambio

Solución greedy

No existe

- una solución optima greedy general
- Casos puntuales con valores “canónicos” funcionan

Solución por fuerza bruta



Podemos realizar un árbol de decisión

Iniciamos la raíz en el cambio X a dar

Por cada denominación posible

Dar 1 moneda de C_i

Generar sub-problema de decisión X-C_i

El camino a la hoja con menor profundidad

Es la menor cantidad de monedas a dar

Complejidad

$O(X^n)$

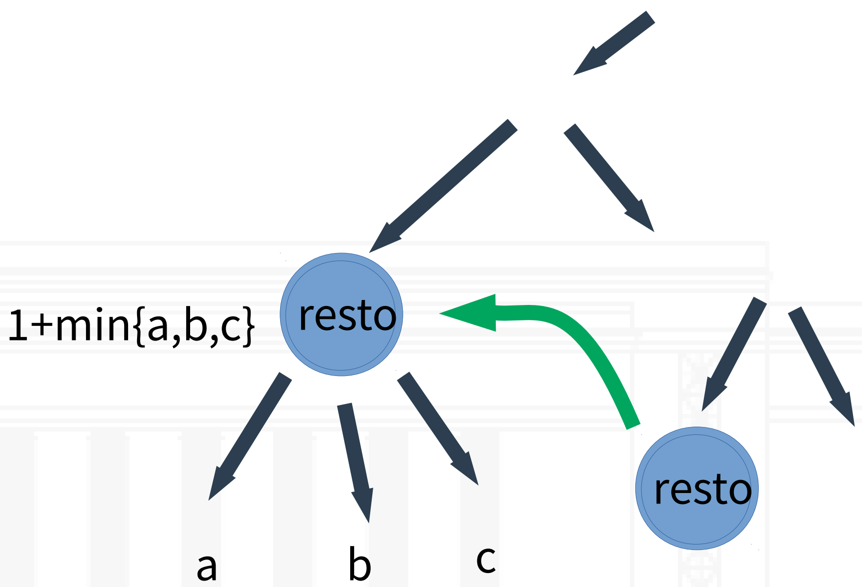
Análisis

Mejoras posibles

Parte de los caminos son iguales

Se los puede calcular solo 1 vez

“es un subproblema”



Subproblema

Calcular el óptimo del cambio X debe usar el mínimo entre los subproblemas X-Cj para j=1...n

Reccurrencia

Podemos expresar el problema como:

$$
\begin{cases}
OPT(x)=0 & , si\ x=0 \\
OPT(x)=1+min_{C_i \in \$}\{OPT(x-C_i)\} & , si\ X>0
\end{cases}
$$

El resultado con el mínimo cambio será:

OPT(x)

Se requieren calcular

Los x-1 OPT() anteriores (no hace falta recalcular un valor previamente obtenido)

En cada subproblema se tiene que realizar n comparaciones.

Solución iterativa

Complejidad

Temporal O(n*X)

Espacial: O(X)

Es un algoritmo pseudo-polinómico

```
OPT[0]=0

Desde i=1 a x

    minimo = +∞
    Desde j=1 a n
        resto = i - C[j]
        si resto >= 0 y minimo > OPT[resto]
            minimo = OPT[resto]

    OPT[i] = 1 + minimo

Retornar OPT[x]
```

Reconstruir las elecciones

Para cada subproblema i

almacenar la C_j que retorne el subproblema de mínimo cambio

Reconstruir para atrás

Partiendo de la denominacion de elección para el OPT[X]

Iterar pasando por lo subproblemas elegidos como mínimos

```
OPT[0]=0
elegida[0] = 0
Desde i=1 a x

    minimo = +∞
    elegida[i] = 0
    Desde j=1 a n
        resto = X - C[j]
        si resto >= 0 y minimo > OPT[resto]
            elegida[i] = j
            minimo = OPT[resto]

    OPT[i] = 1 + minimo


resto = x
Mientras resto >0
    Imprimir C[elegida[resto]]
    resto = resto - C[elegida[resto]]

Imprimir OPT[x]
```

Presentación realizada en Abril de 2020

.UBA fiuba
FACULTAD DE INGENIERÍA

Programación dinámica: Seam carving

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Seam Carving (revisado y revisitado)

Dada una imagen

De h*w pixels

Cada pixel

Tiene un valor de energía asociado e(i,j)

Encontrar la veta

(Horizontal o vertical)

De menor “energía”

Solución greedy

Vimos como resolverlo mediante camino mínimo s-t

Transformamos en un grafo y resolvimos con Dijkstra

Complejidad O([n+m]logn)

con n=h*W (cantidad de pixels y m≈3*n

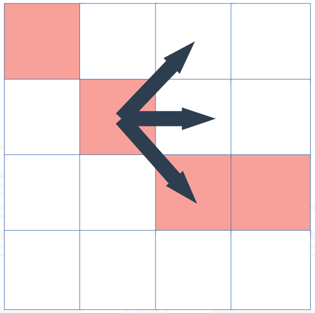
O(h*w*log(h*w))

¿Podemos hacerlo mejor?

Análisis

Vemos la imagen

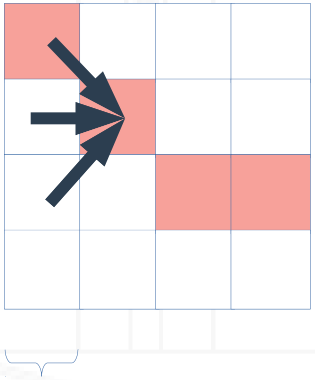
Como una grilla de pixels inter comunicados



Desde un pixel

solo se puede acceder a otros 3 (o 2 en los extremos)

Se puede ser accedido desde otros 3 (o 2 en los extremos)



En la primera columna, por cada pixel

La energía acumulada de pixel solo depende de si mismo

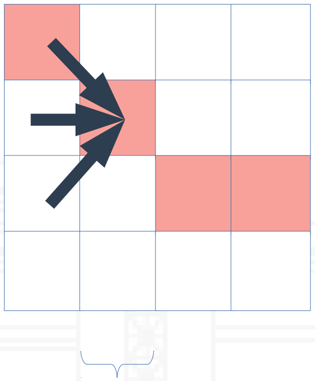
Si fuese una imagen de 1 columna es trivial elegir la veta a remover

Análisis (cont.)

En la segunda columna, por cada pixel

La energía acumulada es la del pixel + como se llego a él

Se puede llegar desde 3 (o 2) pixel de la columna 1.



2da columna

En la columna “n”, por cada pixel

La energía acumulada es la propia + la energía de como se llegó a ella.

Como quiero minimizar: llego por la menor de las 3 (o 2) de la columna n-1

Subproblemas

Podemos partir el problema

Calcular para cada pixel “j” de la columna “i” la energía mínima para llegar a este

Depende unicamente de la columna i-1

Problema base (columna 1)

la energía es del propio del pixel “j”

Podemos expresar el problema como:

$$\begin{cases} OPT(i,j) = e(i,j) & , si\, i = 1 \\ OPT(i,j) = e(i,j) + min\begin{Bmatrix} OPT(i-1, j-1), \\ OPT(i-1, j), \\ OPT(i-1, j+1) \end{Bmatrix} & , si\, i > 1 \end{cases}$$

El resultado con la mínimo energía será:

$$min_{j=1}^{h}\{OPT(w,j)\}$$

Solución iterativa

Complejidad

Temporal: O(w*h)

Espacial: O(w*h)

```
Desde j=1 a h
    OPT[1,j]=e(1,j)

Desde i=2 a w
    Desde j=1 a h
        OPT[i,j] = e(i,j) +
            min {
                OPT(i-1,j-1) ,
                OPT(i-1,j) ,
                OPT(i-1,j+1) ,
            }

menor=+∞

Desde j=1 a h
    if OPT[w,j]<menor
        menor =  OPT[w,j]

Retornar menor
```

Reconstrucción de la veta

Almacenar por cada pixel

- Desde que pixel se llega con mínima energía
- Solo 3 posibles valores j-1 , j. J+1
- Siempre sera de la columna anterior

Desde el pixel de menor energía en la ultima columna

- Reconstruir para atrás el camino de energía menor

Presentación realizada en Abril de 2020

Programación dinámica: Subset Sums y Knapsacks

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Subset Sums

Sea

Un conjunto de “n” elementos $E=\{e_1,e_2,\ldots,e_n\}$

donde cada elemento e_i cuanta con un peso asociado w_i

Queremos

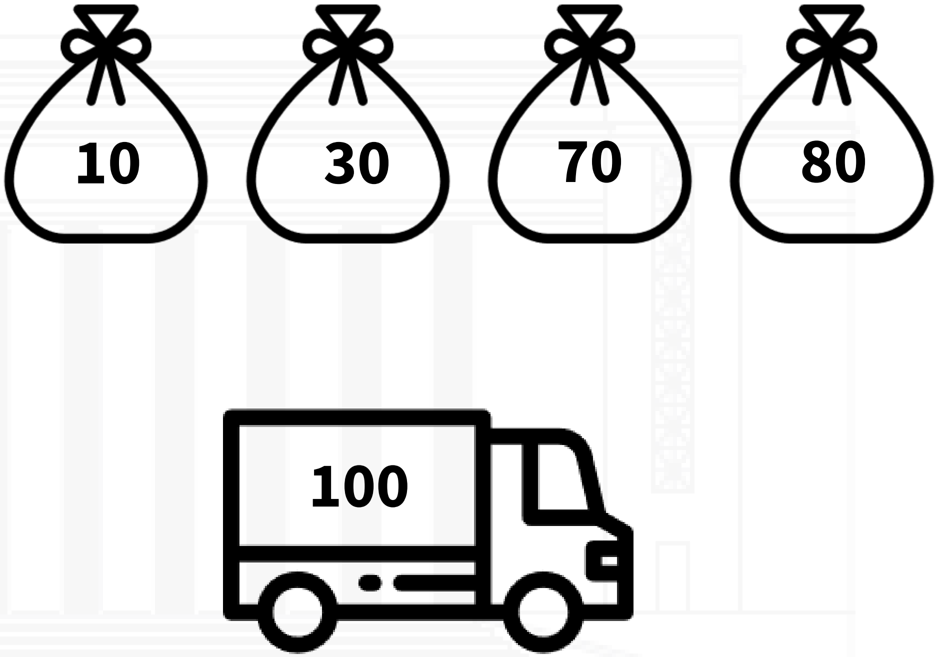
Seleccionar un subset de elementos de E con el mayor peso posible

que no supere un valor W de peso máximo

¿Existe Solución greedy óptima?

Criterios de selección

- Primero más pequeños
- Primero más grandes
- …



No existe solución greedy óptima

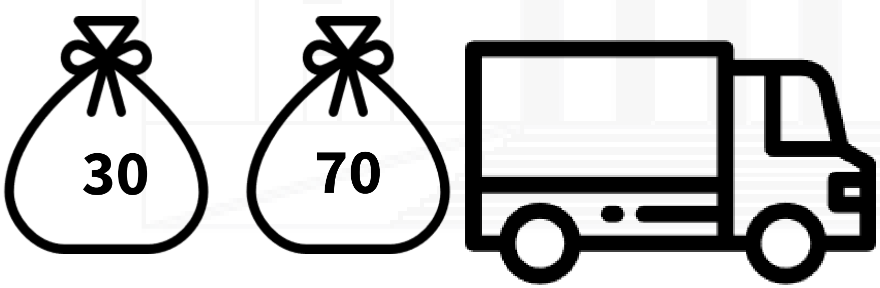
Solución por fuerza bruta

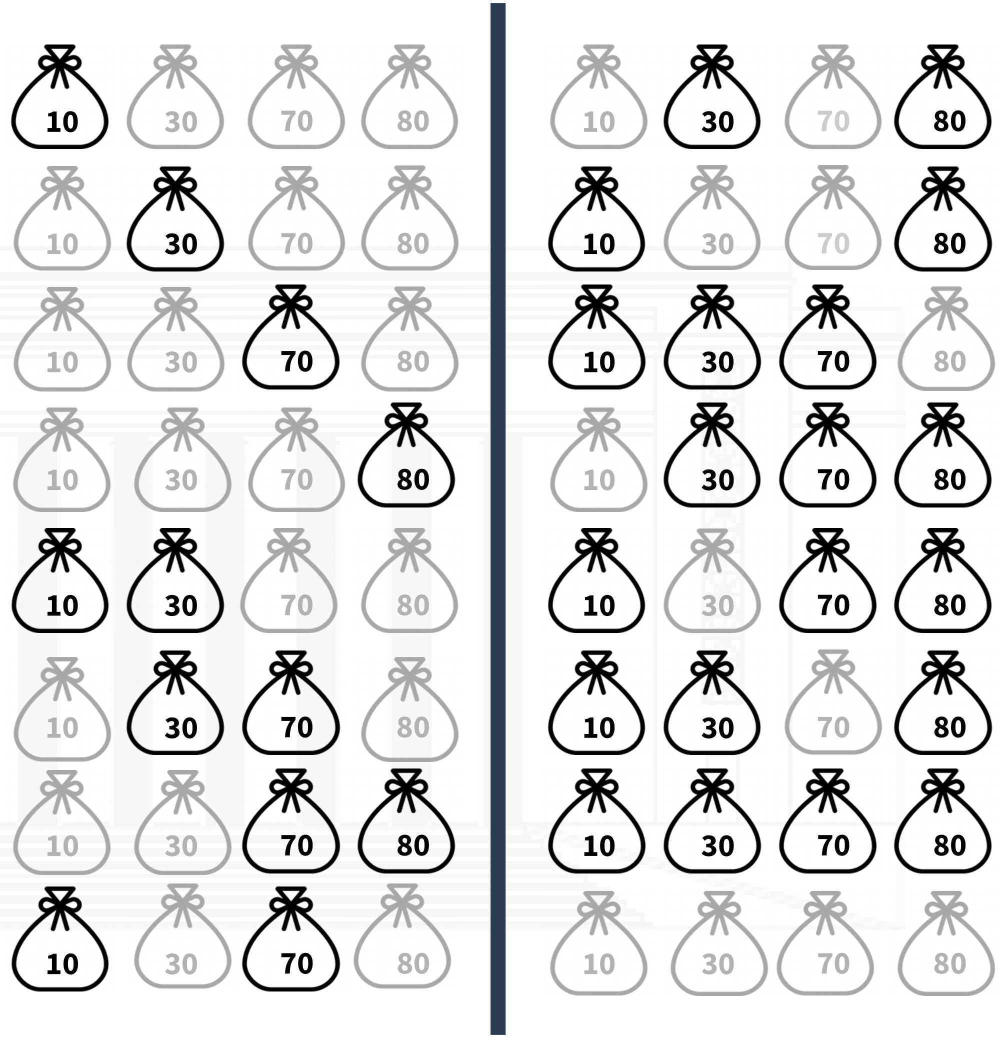
En una solución óptima

Un elemento e_i puede estar o no

Si tengo n elementos

Pueden existir 2^n combinaciones





Buscando mejor solución

Podremos usar programación dinámica?

Existe forma de vincular la selección de un elemento i con los elementos i-1 anteriores?

Es fácil ver que

Si $e_i \notin$ solución → MAX_PESO(e_i) = MAX_PESO(e_i-1)

Pero …

Si $e_i \in$ solución → MAX_PESO(e_i) = w_i + MAX_PESO(???)

Nos falta algo… una variable...

Una cuestión de peso...

Si $e_i \in$ solución

Consume w_i en del W peso disponible

Podemos plantear:

Si $e_i \notin$ solución $\rightarrow$ MAX_PESO(e_i,W) = MAX_PESO(e_i-1, W)

Si $e_i \in$ solución $\rightarrow$ MAX_PESO(e_i,W) = w_i + MAX_PESO(e_i-1, W-w_i)

La mejor solución hasta e_i

Máximo {$e_i \notin$ solución , $e_i \in$ solución }

Subproblemas y recurrencia

Llamaremos

MAX_PESO(i,p)

al problema de determinar el peso máxima que no supere p, utilizando los primeros i elementos del conjunto.

Queremos obtener

MAX_PESO(n,W)

Recurrencia

$$\begin{cases} MAX\_PESO(i,p)=0 & \text{, si } i=0 \text{ o } p=0 \\ MAX\_PESO(i,p)=max\left\{\begin{matrix} w_i+MAX\_PESO(i-1,p-w_i), \\ MAX\_PESO(i-1,p) \end{matrix}\right\} & \text{, si } i>0 \text{ y } p>0 \end{cases}$$

Solución iterativa

Complejidad

Temporal: n*W

Espacial: n*W

Es un algoritmo pseudo polinomial

Si el peso es muy grande nos obliga a realizar muchos cálculos

```
Desde i=0 a n
    OPT[i][0] = 0

Desde p=0 a W
    OPT[0][p] = 0

Desde i=1 a n   // elementos
    Desde p=1 a W  // pesos

        enOptimo = w[i] + OPT[i-1,p-w[i]]
        noEnOptimo = OPT[i-1,p]

        si enOptimo > noEnOptimo
            OPT[l][p] = enOptimo
        sino
            OPT[l][p] = noEnOptimo

Retornar OPT[n,W]
```

Knapsacks

Sea

Un conjunto de “n” elementos $E=\{e_1,e_2,\ldots,e_n\}$

donde cada elemento e_i cuanta con un peso asociado w_i

y una ganancia de v_i

Queremos

Seleccionar un subset de elementos de E con la mayor ganancia posible

que no supere un valor W de peso máximo

Análisis y Subproblema

Es una variante de Subset Sum

Podemos aprovechar el análisis anterior?

Cada elemento e_i que está en la solución

Consume w_i de espacio

Suma v_i de ganancia

Llamaremos

MAX_GANANCIA(i,p)

al problema de determinar la ganancia máxima que no supere p, utilizando los primeros i elementos del conjunto.

Recurrencia

Podemos plantear:

Si $e_i \notin$ solución → MAX_GANANCIA(e_i,W) = MAX_GANANCIA(e_i-1, W)

Si $e_i \in$ solución → MAX_GANANCIA(e_i,W) = v_i + MAX_GANANCIA(e_i-1, W-w_i)

Queremos obtener

MAX_GANANCIA(n,W)

Recurrencia

$$\begin{cases} MAX\_GANANCIA(i,p)=0 & \text{, si } i=0 \text{ o } p=0 \\ MAX\_GANANCIA(i,p)=max\left\{ v_i + MAX\_GANANCIA(i-1,p-w_i), \; MAX\_GANANCIA(i-1,p) \right\} & \text{, si } i>0 \text{ y } p>0 \end{cases}$$

Solución iterativa

Complejidad

Temporal: n*W

Espacial: n*W

Es un algoritmo pseudo polinomial

Si el peso es muy grande nos obliga a realizar muchos cálculos

```
Desde i=0 a n
    OPT[i][0] = 0

Desde p=0 a W
    OPT[0][p] = 0

Desde i=1 a n   // elementos
    Desde p=1 a W  // pesos

        enOptimo = v[i] + OPT[i-1,p-w[i]]
        noEnOptimo = OPT[i-1,p]

        si enOptimo > noEnOptimo
            OPT[l][p] = enOptimo
        sino
            OPT[l][p] = noEnOptimo

Retornar OPT[n,W]
```

Consideraciones

Tanto para Subset Sum como en knaspsak

Si solo se desea calcular el máximo se puede reducir la complejidad espacial (para el subproblema "i" solo se utiliza los resultados de "i-i")

complejidad espacial: O(W)

Si se requiere reconstruir la selección realizada, se puede agregar un indicador binario (si / no) para el subproblema "i","p" sobre conviene elegir o no el elemento en la solución.

Luego se puede desde el subproblema n,W reconstruir para atrás las selecciones.

.UBA fiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2020

Programación dinámica: Bellman-Ford

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Camino mínimo

Dado

- grafo G dirigido y ponderado (con costos positivos)
- de n nodos no aislados y M ejes

Dados dos nodos

- “s” inicial
- “t” final

Encuentra el camino mínimo que los une

Solución

Utilización de algoritmo Dijsktra

Algoritmo greedy

Complejidad O([n+m]logn)

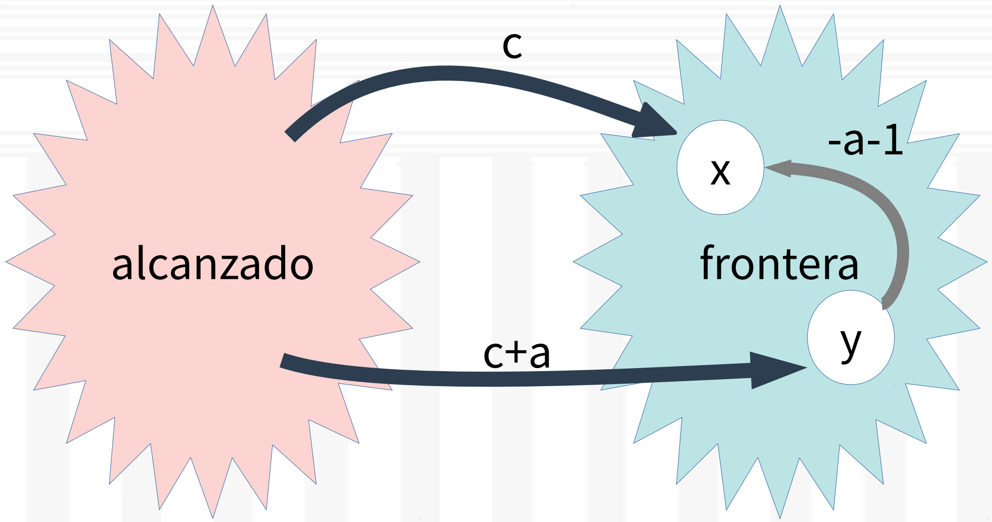
Qué pasa si los existen costos negativos?

Utilizar una arista puede disminuir el costo del camino total

Elección Greedy – con costos negativos

En cada iteración selecciona el menor costo disponible

Puede esto resultar contraproducente?



Elegir el menor costo actual, puede evitar caminos de menor costo general

La solución greedy (Dijkstra) no es optima

Ciclos negativos

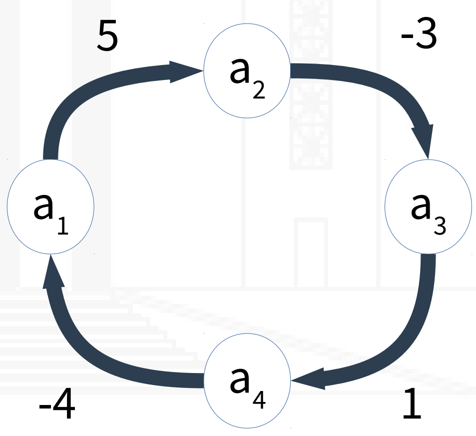
Sea

Nodos $A = \{a_1, a_2, .. a_r\} \in G$

Tal que

Existen las aristas $a_i \rightarrow a_{i+1 \bmod r}$ para todo $a \in A$ conformando un ciclo C

$\sum_{i=0 \text{ to } r} w(a_i \rightarrow a_{i+1 \bmod r}) < 0$



Llamaremos

ciclo negativo "C"

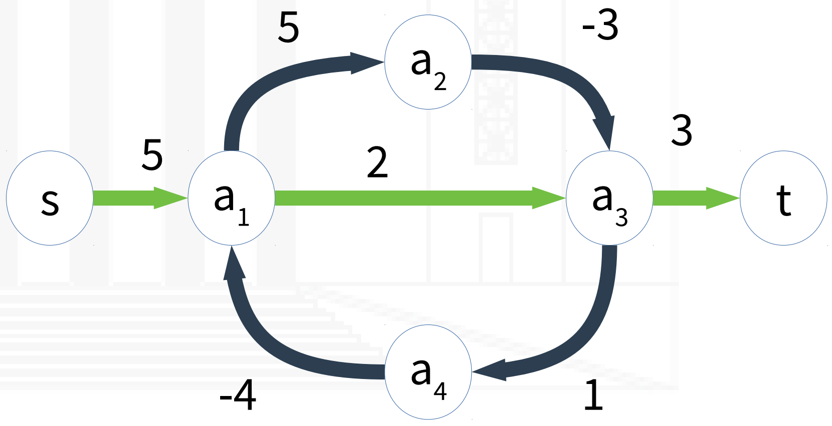
Ciclos negativos y caminos mínimos

La existencia de un ciclo negativo

dentro de un camino s-t

Genera un camino mínimo de -∞

Podemos pensarlo como un loop infinito



Por Fuerza bruta

Para grafo conderado G sin ciclos negativos

Calcular camino minimo s-t

Se deben calcular

Todos costos de los caminos posibles s-t de longitud 1

Todos costos de los caminos posibles s-t de longitud 2

...

Todos costos de los caminos posibles s-t de longitud n-1

El camino mínimo tendrá longitud n-1 como máximo

Bellman-Ford

Algoritmo para camino mínimo

Para grafos con ponderados sin ciclos negativos

Utiliza programación dinámica

Propuesto por

Alfonso Shimbel (1955)

Lester Ford Jr. (1956)

Edward F. Moore (1957)

Richard Bellman (1958)

Análisis

Para llegar desde “s” a un nodo n_i

puede haber utilizado diferentes caminos (nodos y longitudes diferentes)

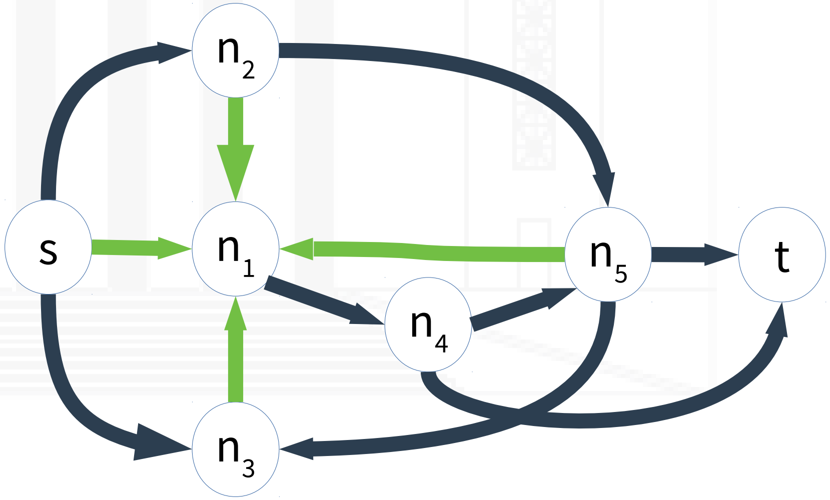
Unicamente puedo llegar desde sus nodos que lo conectan (pred[n_i])

A los que tambien puedo llegar por diferentes caminos

Para llegar desde “s” a n_i en j pasos

Tengo que haber llegado a sus predecesores en j-1 pasos

Y para llegar a estos, se debe haber llegado a sus predecesores en j-2 pasos

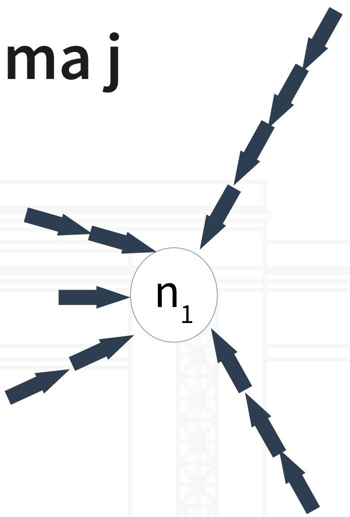


Análisis (cont.)

Al camino mínimo hasta el nodo n_i con longitud máxima j

Lo llamaremos minPath(n_i,j)

Es el mínimo camino que llega a n_i con longitud entre 0 a j



Es el mínimo entre los caminos mínimos que llegan a sus predecesores en máximo j-1 pasos + el peso de llegar a n_i

$$\text{minPath}(n_i, j) = min\left\{ \begin{matrix} \text{minPath}(n_i, j-1) \\ min\left\{\text{minPath}(n_x, j-1) + w(n_x, j-1)\right\} \end{matrix} \right\}$$

con $n_x \in \text{pred}(n_i)$

Subproblemas y recurrencia

El camino mínimo s-t

No contiene ciclos internos. (no hay ciclos negativos en el grafo)

Agregar ciclos solo aumenta el peso del camino

minPath(t,n-1)   *con n la cantidad de nodos en el grafo*

Reccurencia

$$\text{minPath}('s', j) = 0$$

$$\text{minPath}(n_{i,0}) = +\infty \quad \text{con } n_i \neq s$$

$$\text{minPath}(n_i, j) = min\left\{ \begin{array}{l} \text{minPath}(n_i, j-1) \\ min\{\text{minPath}(n_x, j-1) + w(n_x, j-1)\} \end{array} \right\} \quad \text{con } n_x \in \text{pred}(n_i)$$

Solución iterativa

Llamaremos OPT[l][v]

Al camino mínimo de “s” al nodo n_v con máxima longitud l

El nodo “s” se encuentra en v=0

El nodo “t” se encuentra en v=n

Lo almacenaremos en una matriz de nxn

```
Desde l=0 a n-1
     OPT[l][0] = 0

Desde v=0 a n-1
     OPT[0][v] = +∞

Desde l=1 a n-1    // max longitud del camino
     Desde v=1 a n  // nodo
          OPT[l][v] = OPT[l-1][v]
          Por cada p prededesor de v
               si OPT[l][v] > OPT[l-1][p] +
                                        w(p,v)

                    OPT[l][v] = OPT[l-1][p] +
                                        w(p,v)
Retornar OPT[n-1,n]
```

Complejidad temporal

El primer loop esta acotado por n

El acceso de los predecesor se puede hacer en forma eficiente si se invierte el grafo

El segundo loop y el “por cada predecesor” se ejecuta m veces

(m = número de aristas)

O(mxn)

```
Desde l=0 a n-1
    OPT[l][0] = 0

Desde v=0 a n-1
    OPT[0][v] = +∞

Desde l=1 a n-1    // max longitud del camino
    Desde v=1 a n  // nodo
        OPT[l][v] = OPT[l-1][v]
        Por cada p prededesor de v
            si OPT[l][v] > OPT[l-1][p] +
                                    w(p,v)

                OPT[l][v] = OPT[l-1][p] +
                                    w(p,v)
Retornar OPT[n-1,n]
```

Complejidad espacial

La matriz ocupa mxn

El grafo se puede mantener en n+m

(lista de adjacencias)

O(mxn)

Se puede mejorar, solo es necesario los resultado de la última iteración principal.

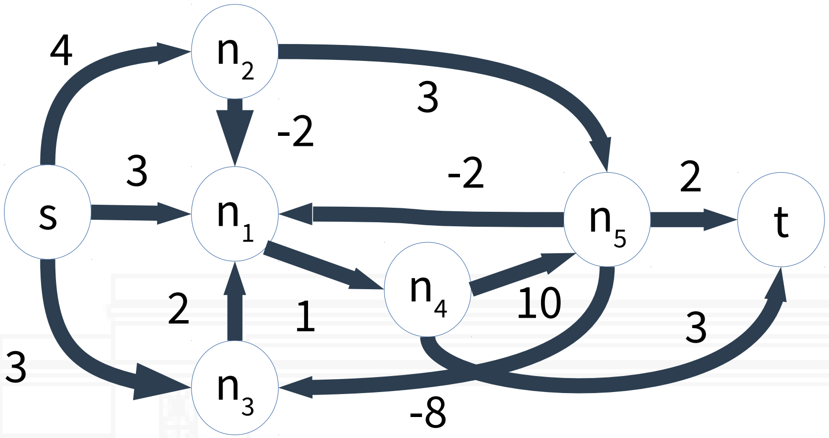
```
Desde l=0 a n-1
    OPT[l][0] = 0

Desde v=0 a n-1
    OPT[0][v] = +∞

Desde l=1 a n-1   // max longitud del camino
    Desde v=1 a n  // nodo
        OPT[l][v] = OPT[l-1][v]
        Por cada p prededesor de v
            si OPT[l][v] > OPT[l-1][p] +
                                w(p,v)

                OPT[l][v] = OPT[l-1][p] +
                                w(p,v)
Retornar OPT[n-1,n]
```

Ejemplo



	s	n_1	n_2	n_3	n_4	n_5	t
0	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	0	3	4	3	$+\infty$	$+\infty$	$+\infty$
2	0	2	4	3	4	7	$+\infty$
3	0	2	4	-1	3	7	7
4	0	1	4	-1	3	7	7
5	0	1	4	-1	2	7	7
6	0	1	4	-1	2	7	5

```
Desde l=0 a n-1
     OPT[l][0] = 0

Desde v=0 a n-1
     OPT[0][v] = +∞

Desde l=1 a n-1    // max longitud del camino
     Desde v=1 a n  // nodo
          OPT[l][v] = OPT[l-1][v]
          Por cada p prededesor de v
               si OPT[l][v] > OPT[l-1][p] +
                                        w(p,v)

                    OPT[l][v] = OPT[l-1][p] +
                                        w(p,v)
Retornar OPT[n-1,n]
```

Camino mínimo:

s-n_2-n_5-n_3-n_1-n_4-t

Costo:

5

Reconstruir las elecciones

Basta agregar un vector

“pred“ De longitud n

Que almacene en la posición i

Desde que predecesor se llega actualmente a ni con el menor costo desde s

Al finalizar la ejecución

Se itera desde pred[n] con n=”t”

pred[pred[n]], pred[pred[pred[n]]]

Hasta llegar a “s”

¿Qué pasa si hay ciclos negativos?

Se puede encontrar un camino desde “s” a un n_i

Con longitud “n” menor al los caminos con máxima longitud n-1

(incluso cuanto mayor la longitud a n más disminuirá el camino mínimo)

Ejecutando Bellman-Ford

una iteración principal más (longitud camino “n”)

si cambia el íinimo de al menos un nodo

Entonces el grafo tiene un ciclo negativo.

Presentación realizada en Abril de 2020

Programación dinámica: Maximum Subarray Problem

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Maximum Subarray Problem

Sea

Una lista de “n” elementos ordenados

Cada elemento e_i tiene un valor numérico v_i

Queremos

Calcular un subconjunto contiguo de elementos S

tal que la suma de los valores sea la máxima posible

Ejemplo

Dados los elementos

-3	5	-3	4	2	1	-10	2	2	-2	1	5
1	2	3	4	5	6	7	8	9	10	11	12

El subvector de suma máxima es

-3	5	-3	4	2	1	-10	2	2	-2	1	5
1	2	3	4	5	6	7	8	9	10	11	12

Suma máxima: 9

Solución por fuerza bruta

El elemento 1

Puede ser el elemento inicial de n soluciones

-3	5	-3	4	2	1	-10	2	2	-2	1	5
1	2	3	4	5	6	7	8	9	10	11	12

El elemento 2

Puede ser el elemento inicial de n-1 soluciones

5	-3	4	2	1	-10	2	2	-2	1	5
2	3	4	5	6	7	8	9	10	11	12

…

El elemento n

Puede ser elemento inicial de 1 solución

5
12

Total de subvectores posibles

$n + n\text{-}1 + n\text{-}2 + \ldots + 1 = n^*(n\text{-}1)/2 = (n^2 - n)/2$

Tiempo total de cálculo

Por cada subvector posible, hacer la suma: $O(n^3)$

Podemos hacerlo mejor?

Podemos pensarlo como subproblemas:

MAX(i): El máximo subvector que termina en el elemento i

Para el elemento i=3 tenemos que probar:

(e3)=-3 y (e2,e3)=2 y (e1,e2,e3)=-1

Y quedarnos con el máximo. (e2,e3)

Podemos calcular los n subproblemas

-3	5	-3	4	2	1	-10	2	2	-2	1	5
1	2	3	4	5	6	7	8	9	10	11	12

Y quedarnos con el máximo total

… Lo podemos hacer en $O(n_2)$

… podemos hacerlo aun mejor? SI! Existe una solución usando división y conquista O(nlogn)

… y se puede aun mejor

Algoritmo Kadane

Propuesto por Joseph "Jay" Born Kadane

En 1977 (y resuelto en pocos minutos)

Resuelve el problema de Maximum Subarray Problem

O(n)

La historia de la invención puede leerse en

“programming pearls” por Jon Bentley

https://dl.acm.org/doi/pdf/10.1145/358234.381162

Relación entre subproblemas

El máximo subvector que termina en el elemento i

Esta relacionado con el máximo subvector que termina en el elemento i-1

Si el máximo subvector que termina en i-1

$MAX(i-1) > 0 \rightarrow MAX(i) = MAX(i-1) + v_i$

$MAX(i-1) < 0 \rightarrow MAX(i) = v_i$

Recurrencia

Definimos:

$$\begin{cases} \text{MAX}(1) = v[1] \\ \text{MAX}(i) = max\{\text{MAX}(i-1), 0\} + v[1] \end{cases}$$

Queremos obtener:

MAX(n)

Pseudocódigo

Complejidad

Temporal: O(n)

Espacial: O(1)

```
…
MaximoGlobal = v[1]
MaximoLocal = v[1]
IdxFinMaximo = 1

Desde i=2 a n   // elementos
    MaximoLocal = max(MaximoLocal,0) + v[i]

    if MaximoLocal > MaximoGlobal
        MaximoGlobal = MaximoLocal
        IdxFinMaximo = i


Retornar MaximoGlobal
```

Comparación de tiempos de ejecución

TABLE I. Summary of the Algorithms

Algorithm		1	2	3	4
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	$46N \log N$	$33N$
Time to solve problem of size	10^2	3.4 secs	130 msecs	30 msecs	3.3 msecs
	10^3	.94 hrs	13 secs	.45 secs	33 msecs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 min	3.3 secs
	10^6	108 mill	5 mos	15 min	33 secs
Max problem solved in one	sec	67	280	2000	30,000
	min	260	2200	82,000	2,000,000
	hr	1000	17,000	3,500,000	120,000,000
	day	3000	81,000	73,000,000	2,800,000,000
If N multiplies by 10, time multiplies by		1000	100	10+	10
If time multiplies by 10, N multiplies by		2.15	3.16	10−	10

Algoritmos: 1) Fuerza bruta. 2) mejoras de fuerza bruta. Tiempo cuadrático. 3) Utilizando división y conquista. 4) Con programación dinámica

Presentación realizada en Abril de 2020

Programación dinámica: Mínimos cuadrados segmentados

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

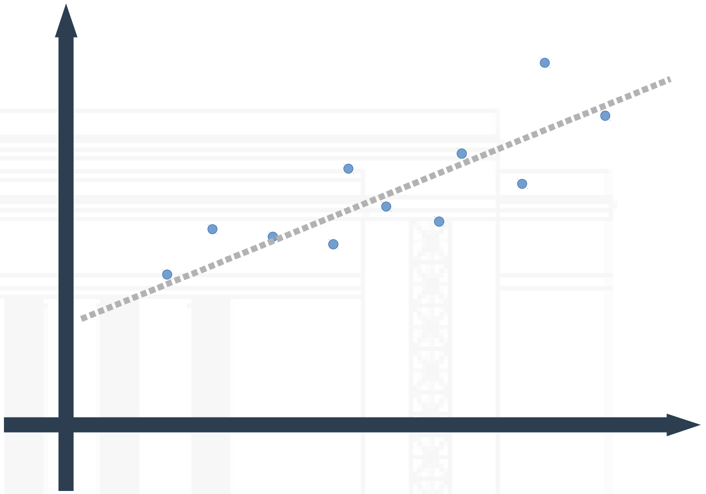
vpodberezski@fi.uba.ar

Mínimos cuadrados segmentados

Sea

Un set de "n" puntos P=(x,y) en el plano

p_i = (xi,yi) ∈ P y $x_i > x_a$ para todo a>i



Queremos

aproximar mediante segmentos los puntos de P

minimizando el error cometido

con la menor cantidad posibles de segmentos

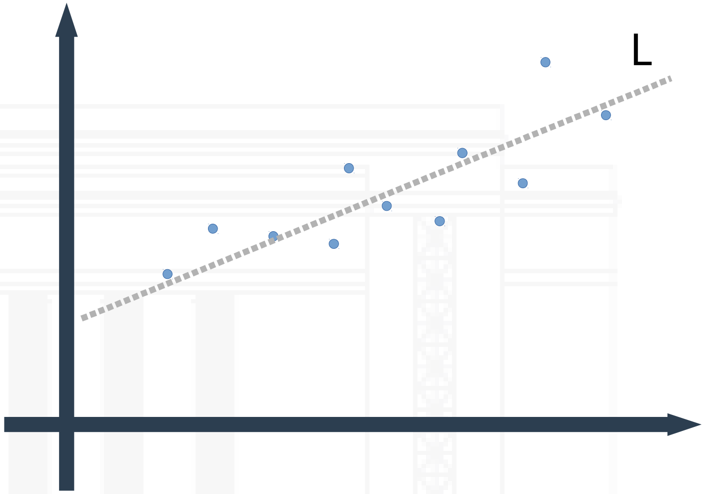
Segmentos

Recta de aproximación

$$L = a\,x + b$$

$$a = \frac{n \sum_i x_i\, y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2}$$

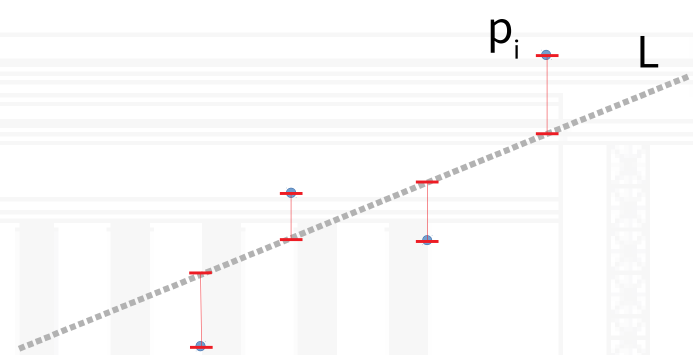
$$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$



Cálculo de error

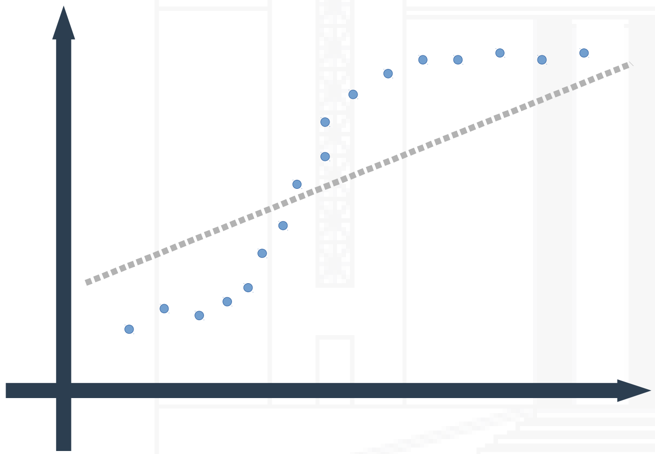
Error cometido

$$E(L,P)=\sum_{i}^{n}(y_i-ax_i-b)^2$$

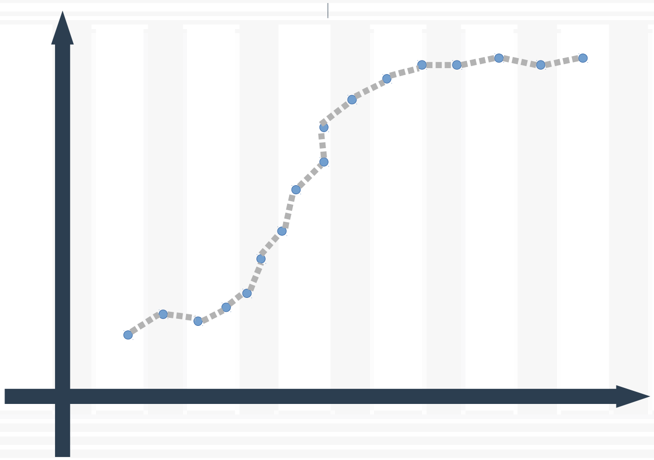


Ejemplo

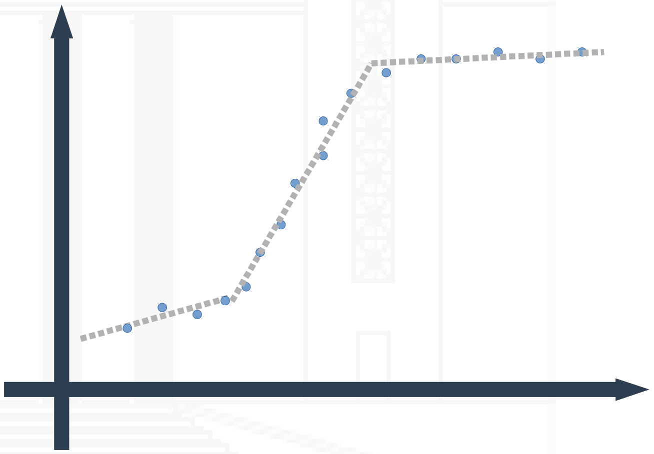
Diferentes soluciones a un mismo set de puntos



1 segmento. Error grande



n-1 segmentos. Cero error



3 segmentos. Error aceptable

Penalización por nuevo segmento

Proponemos un parámetro “C” > 0

Penalidad por cada segmento añadido

El ajuste del mismo establecerá un equilibrio entre segmentos y error cometido

A mayor “C” → menos segmentos

A menor “C” → menos error

Solución por fuerza bruta

Sobre n puntos

Cada punto i>1 puede ser el final de un segmento

Podemos representarlo como un vector de tamaño n

El valor 0 o 1 (es el final del segmento o no)

Existen 2^n combinaciones posibles

Para cada una, calcular los segmentos O(n) y el error cometido O(n)

Complejidad $O(2^n * n)$

Análisis

El punto p_n pertenece al último segmento

- Este segmento inicia en un p_x anterior

Si en la solución optima conocemos el último segmento

- Conocemos el error $e_{x,n}$
- El problema se reduce al subproblema entre los puntos 1 y x-1

$OPT(n) = e_{x,n} + C + OPT(x-1)$

Análisis (cont.)

Pero no conocemos el óptimo…

Pero queremos elegir como último segmento aquel que minimice el error general

$$OPT(n) = min_{1 \le x \le n}(e_{x,n} + C + OPT(x-1))$$

Podemos generalizar el problema como

$$\begin{cases} OPT(i) = min_{1 \le x \le i}(e_{x,i} + C + OPT(x-1)) \\ OPT(0) = 0 \end{cases}$$

Solución iterativa

Complejidad Temporal

El cálculo de cada optimo es O(n)

Se calculan n óptimos

→ $O(n^2)$

El calculo de los errores se realizara para todos los pares posibles. Es O(n2)

El calculo del error es O(n)

→ $O(n^3)$ que es la complejidad total

```
OPT[0] = 0

Para todo par i,j con i<=j
        Calcular e[i][j]

Desde j=1 a n

    OPTIMO[j] = +∞

    Desde i=1 a n
        segmento = e[i][j] + C + OPT[i-1]

        si OPTIMO[j] > segmento
            OPTIMO[j] = segmento

Retornar OPT[n]
```

Solución iterativa (cont.)

Complejidad espacial

Almacenamiento de errores: $O(n^2)$

Almacenamiento del optimo $O(n)$

Total: $O(n^2)$

```
OPT[0] = 0

Para todo par i,j con i<=j
        Calcular e[i][j]

Desde j=1 a n

    OPTIMO[j] = +∞

    Desde i=1 a n
        segmento = e[i][j] + C + OPT[i-1]

        si OPTIMO[j] > segmento
            OPTIMO[j] = segmento

Retornar OPT[n]
```

.UBA fiuba
FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2020

Programación dinámica: Problema del viajante

Teoría de Algoritmos I (75.29 / 95.06)
Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Problema del viajante de comercio (TSP)

Sea

Un conjunto de n ciudades “C”

Un conjunto de rutas con costo de tránsito

Existe una ruta que une cada par de ciudades

El costo de cada ruta puede ser simétrico o asimétrico (diferente valor ida y vuelta)

Queremos

Obtener el circuito de menor costo

que inicia y finalice en una ciudad inicial

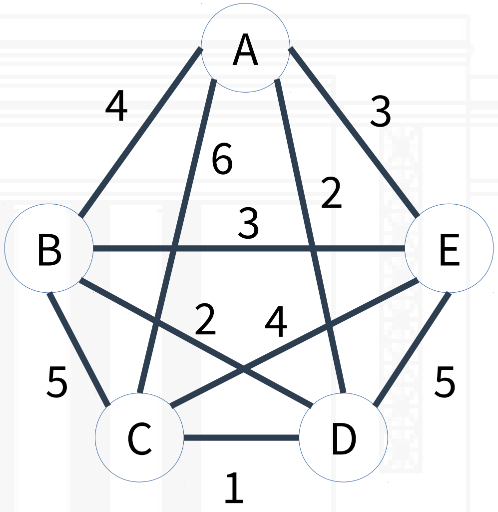
y pase por el resto de las ciudades 1 y solo 1 vez

Ejemplo

Contamos con 5 ciudades

El costo por camino es simétrico

Partimos de A



Posibles ciclos:

A – B – C – D – E – A → Costo: 18

A – C – B – D – E – A → Costo: 21

A – D – B – C – E – A → Costo: 16

. . .

Ciclo de menor costo:

A – D – C – B – E – A → Costo: 14

A – D – C – E – B – A → Costo: 14

Solución por fuerza bruta

Tenemos que calcular todos los ciclos posibles.

Con n ciudades todas interconectadas (grafo completo)

Existen (n-1)! Ciclos de longitud n-1

Para cada ciclos calcular su costo

O(n)

Y quedarnos con el mínimo

Complejidad

$O(n*(n-1)!) \rightarrow O(n!)$

Algoritmo Bellman–Held–Karp

Propuesto forma independiente en 1962

"Dynamic programming treatment of the travelling salesman problem" por Richard Bellman

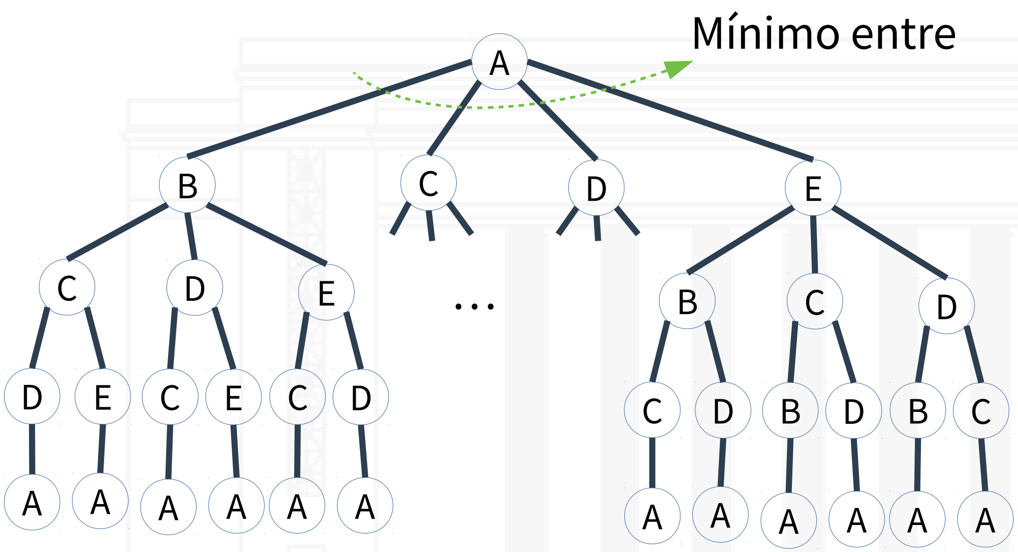
http://akira.ruc.dk/~keld/teaching/algoritmedesign_f08/Artikler/05/Bellman61.pdf

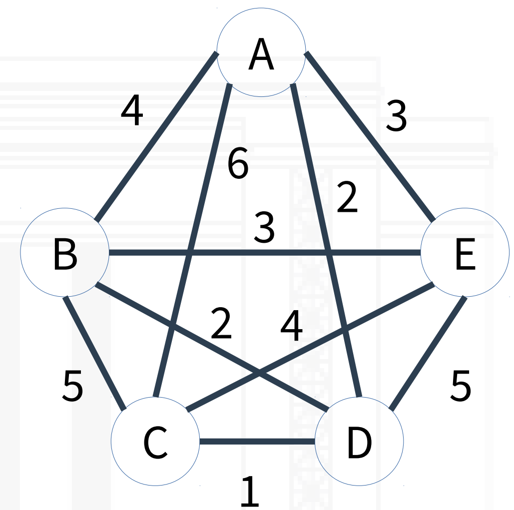
"A dynamic programming approach to sequencing problems" por Michael Held y Richard M. Karp"

Utiliza programación dinámica

Análisis

Podemos descomponer el problema como





Son (n-1)! hojas

… pero algunas ramas se ven repetidas

Análisis

El óptimo de problema

Se puede descomponer como el mínimo entre los subproblemas menores.

$$OPT(i,\{S\}) = min_{j\in\{S\}} w(i,j) + OPT(j,\{S-j\})$$

$$OPT(i,\emptyset) = w(i,\text{start}), \qquad \text{start: ciudad inicial}$$

Con

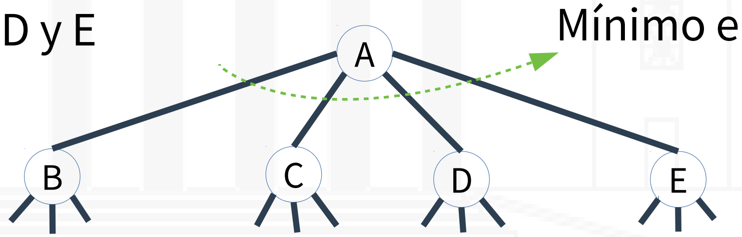
{S} subset de ciudades

i ciudad de partida

En el ejemplo:

Inicialmente parto de "A"

Si voy por "B", luego tengo que pasar por C,D y E



Mínimo entre

OPT(A,{B,C,D,E}) = min [w(A,B) + OPT(B,{C,D,E}) , w(A,C) + OPT(C,{B,D,E}) , w(A,D) + OPT(D,{B,C,E}) , w(A,E) + OPT(E,{B,C,D})]

Análisis (cont.)

Todos los subproblemas que tenemos que calcular:

OPT('B',{ C,D ,E}) , OPT('C',{ B, E, D }) , OPT('D',{ B, C, E }, OPT('E',{ B, C, D }),

OPT('B',{ C,D }) , OPT('B',{ C,E }) , OPT('B',{ D, E }),
OPT('C',{ B,D }) , OPT('C',{ B,E }) , OPT('C', { D, E }),
OPT('D',{ B,C }) , OPT('D',{ B,E }) , OPT('D',{ C, E }),
OPT('E',{ B,C }) , OPT('E',{ B,D }) , OPT('E',{ C, D }),

OPT('B',{ C }) , OPT('B',{ D }) , OPT('B',{ E }),
OPT('C',{ B }) , OPT('C',{ B }) , OPT('C',{ E }),
OPT('D',{ B }) , OPT('D',{ C }) , OPT('D',{ E }),
OPT('E',{ B }) , OPT('E',{ C }) , OPT('E',{ D }),

OPT('B', ∅) , OPT('C', ∅), OPT('D', ∅),OPT('E', ∅)

Para las n-1 ciudades excepto la ciudad de inicio (y fin)

Combinaciones de ciudades tomadas de diferente cantidad (según nivel subproblema)

Análisis (cont.)

Todos los subproblemas que tenemos que calcular:

Combinación de 3 ciudades

OPT('B',{ C,D ,E}) , OPT('C',{ B, E, D }) , OPT('D',{ B, C, E }, OPT('E',{ B, C, D }), $\binom{3}{3} * 4$

Tomadas de a 3

$+$

OPT('B',{ C,D }) , OPT('B',{ C,E }) , OPT('B',{ D, E }),
OPT('C',{ B,D }) , OPT('C',{ B,E }) , OPT('C', { D, E }),
OPT('D',{ B,C }) , OPT('D',{ B,E }) , OPT('D',{ C, E }),
OPT('E',{ B,C }) , OPT('E',{ B,D }) , OPT('E',{ C, D }), $\binom{3}{2} * 4$

$+$

OPT('B',{ C }) , OPT('B',{ D }) , OPT('B',{ E }),
OPT('C',{ B }) , OPT('C',{ B }) , OPT('C',{ E }),
OPT('D',{ B }) , OPT('D',{ C }) , OPT('D',{ E }),
OPT('E',{ B }) , OPT('E',{ C }) , OPT('E',{ D }), $\binom{3}{1} * 4$

$+$

OPT('B', ∅) , OPT('C', ∅), OPT('D', ∅),OPT('E', ∅) $\binom{3}{0} * 4$

Análisis (cont.)

Cantidad total de subproblemas:

Para n ciudades

$$\sum_{k=0}^{n-2} \binom{n-2}{k} * (n-1)$$

Que podemos expresar como:

$$(n-1) * \sum_{k=0}^{n-2} \binom{n-2}{k}$$

Y utilizando teorema del binomio con x=y=1

$$(n-1) * \sum_{k=0}^{n-2} \binom{n-2}{k} 1^{n-2-k} 1^{k} \quad = (n-1) * (1+1)^{(n-2)}$$

Teorema del binomio:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

Análisis (cont.)

Cantidad total de subproblemas:

$$=(n-1)*(2)^{(n-2)}$$

No es polinomial

… pero es mejor que exponencial para n grandes

Cada subproblema debe buscar el mínimo

Utilizando sus subproblemas

La comparación se ejecuta en tiempo lineal

Solución iterativa

Llamamos

C al conjunto de todas las ciudades

1 a la ciudad inicial

El resto de las ciudades están numeradas de 2 a n

Complejidad total:

Cantidad de subproblemas * calculo del costo subproblemas (lineal)

$O(n^2 2^n)$

```
Desde i=2 a n      //ciudad 1 es la inicial
    OPT[i][∅] = W[i][1]

Desde k=1 a n-2
    Para todo subset S de C-{1} de tamaño k

        Para cada elemento i de S

            OPT[‘i’,S-{i}] = +∞

            Por cada elemento j de S-{i}
                r=OPT[j,S-{i,j}] + w[j][i]

                Si (r<OPT[‘i’,S-{i}])
                    OPT[‘i’,S-{i}]=r

CaminoMinimo = +∞
Desde j=2 a n
    ciclo = OPT[i,S-{1,i}] + w[1][i]
    Si (CaminoMinimo>ciclo)
            CaminoMinimo = ciclo
Retornar caminoMinimo
```

Conclusión

El problema del viajante de comercio

Utilizando fuerza bruta $O(n!)$

Utilizando programación dinámica $O(n^2 2^n)$

No es polinómico

No se conoce una solución polinómica.

(si alguien la encuentra… será el hombre mas famoso (o infame) del mundo.

n	Fuerza bruta	Prog dínamica
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800
10	3628800	102400
15	1307674368000	7372800
20	2,43290200817664E+018	419430400
50	3,04140932017134E+064	2,81474976710656E+018
100	9,33262154439442E+157	1,26765060022823E+034

Presentación realizada en Abril de 2020