

Algoritmos por fuerza Bruta: Generar y probar

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

¿Qué entendemos por fuerza Bruta?

- **Fuerza bruta:**

- Proceso arduo "manual" o "físico"
- Estrategia de resolución construida mediante un pobre o mínimo trabajo "mental"
- También conocido como: búsqueda exhaustiva

- **Algoritmos por fuerza bruta:**

- En el peor de los casos prueban todos las posibles soluciones para hallar la respuesta buscada

- **Metodologías que usan Fuerza bruta:**

- Generar y probar
- Vuelta atrás (backtracking)
- Ramificar y probar (branch and bound)

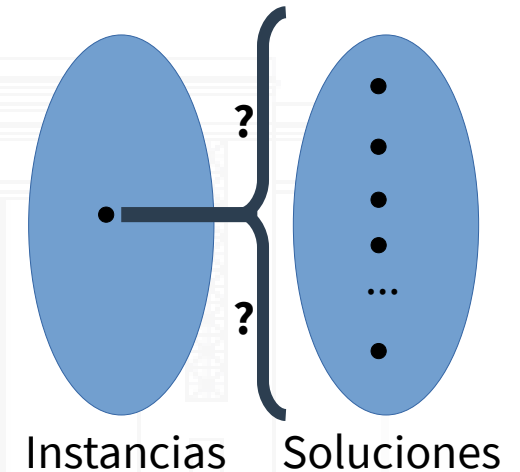
Generar y probar

- **Función generativa:**

- Permite recorrer todo el espacio de soluciones posibles.
- Estará restringido por la naturaleza del problema (restricciones explícitas)

- **Función de prueba:**

- Verifica cada solución candidata (restricciones implícitas).
- Una solución es factible si cumple con los criterios de solución del problema



Generar y probar: Esquema general

La función generativa provee :

una estructura para representar una posible solución del espacio de soluciones.

una manera de obtener la próxima solución

una manera de determinar la ausencia de próximas soluciones

La función de prueba provee:

la lógica para verificar la factibilidad de la solución

la lógica para comparar una solución contra otra.

Sea S el dominio de soluciones posibles al problema P

Mientras queden soluciones candidatas por verificar y no se haya encontrado la solución

Sea s la siguiente solución candidata a verificar

Si s corresponde a una solución válida
retornar s

Espacios de soluciones

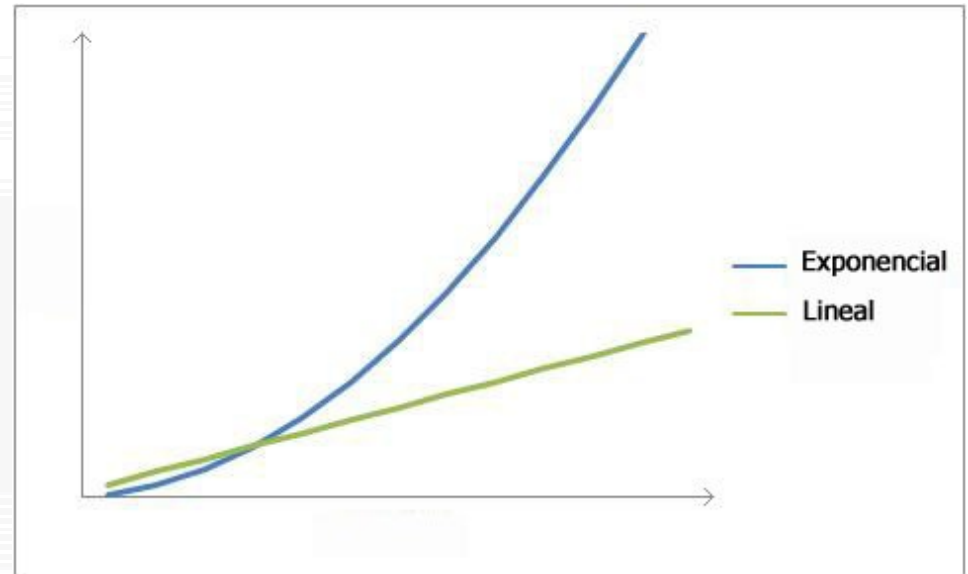
- **Espacios de soluciones exponenciales**

- n-tuplas.
- Permutaciones,
- Combinaciones
- Particiones de conjunto
- ...



Explosión combinatoria

- **A medida que el tamaño de la instancia crece linealmente**
 - El espacio de soluciones crece exponencialmente (o peor)
- **Generar y probar**
 - Para instancias relativamente grandes el tiempo de ejecución se vuelve inmanejable
 - Solo utilizar antes la ausencia de un método de resolución alternativo mas eficiente.



Generar y probar: n-tuplas y el problema de la mochila

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Generar y probar: n-tuplas

- **Espacio de soluciones**

- Trabajaremos sobre la elección de un subconjunto entre n elementos
- Los elementos son únicos e indivisibles.
- Un elemento puede estar seleccionado o no.
- No importa el orden de los elementos seleccionados

- **Función generativa**

- Generación de las n -tuplas
- Corresponderá a las restricciones explícitas del problema

Ejemplo práctico: Problema de la mochila

- **Contamos con:**
 - una mochila con una capacidad de K kilos
 - un subconjunto del conjunto E de “ n ” elementos
- **Cada elemento i tiene:**
 - un peso de k_i kilos
 - un valor de v_i .
- **Queremos seleccionar un subconjunto de E**
 - con el objetivo de maximizar la ganancia.
 - el peso total seleccionado no puede superar la capacidad de la mochila.



Ejemplo práctico: Problema de la mochila

- Podemos asignarle a cada elemento identificador único (un valor entero)
- Asignar un orden a los elementos según su identificador

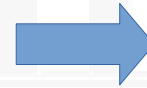
1	2	3	4	5	6	7	8	9	10	11	12	13
k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}
V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}



Ejemplo práctico: Problema de la mochila

- Cualquier subconjunto de los identificadores corresponde a una posible solución al problema
- Una solución es factible si su peso combinado es menor a la capacidad de la mochila
- La solución óptima es aquella solución factible con mayor suma de valor entre sus elementos

1	2	3	4	5	6	7	8	9	10	11	12	13
		X				X			X			
k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}



$$¿ k_3 + k_7 + k_{10} \leq K ?$$

$$\text{Valor} = v_3 + v_7 + v_{10}$$

Ejemplo práctico: Problema de la mochila

- Podemos expresar una posible solución como un vector de “n” posiciones
- Espacio de soluciones: Podemos tener 2^n posibles soluciones.
- Llamamos al proceso de construir estas posibles soluciones como la generación de todas las n-tuplas

1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	0	0	0	1	0	0	1	0	0	0



$$¿ k_3 + k_7 + k_{10} \leq K ?$$

$$\text{Valor} = v_3 + v_7 + v_{10}$$

Generación de n-tuplas

- Se generarán cada uno de los elementos posibles en orden lexicográfico.
 - Cada vector que representa una posible solución también representa en binario el orden en el que es encontrado y evaluado esa posible solución.
- Para la generación de estas soluciones podemos utilizar un contador binario.

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Generación de n-tuplas: Contador binario

Para inicializar:

comenzar con una mochila vacía Ningún elemento seleccionado

Complejidad temporal: $O(n)$

Obtener la próxima posible solución:

Comenzando por la derecha del vector, mientras que el contenido sea un 1 reemplazar por un 0

Establecer el siguiente elemento del vector en 1.

Complejidad amortizada temporal: $O(1)$

Sea C un vector de n posiciones (soluciones)

Inicializar C:

Desde $x=0$ a n
 $C[x]=0$

Incrementar C:

Sea $pos=n-1$
Mientras $C[pos]==1$ y $pos>0$
 $C[pos]=0$
 Decrementar pos
Si $pos==0$
 Retornar 'fin' //Overflow!
 $C[pos]=1$
retornar C

Generación de n-tuplas: Contador binario

Para inicializar:

Obtener la próxima posible solución
(Ejemplo):

0 0 0

0 1 1



1 0 0

Sea C un vector de n posiciones (soluciones)

Inicializar C:

Desde $x=0$ a n

$C[x]=0$

Incrementar C:

Sea $pos=n-1$

Mientras $C[pos]==1$ y $pos>0$

$C[pos]=0$

Decrementar pos

Si $pos== -1$

Retornar 'fin' //Overflow!

$C[pos]=1$

retornar C

Verificación de la solución

Factibilidad:

Sumar los pesos de los elementos seleccionados (en el vector con un "1") y comparar con la capacidad de la mochila

Complejidad temporal: $O(n)$

Complejidad espacial: $O(1)$

Ganancia:

Sumar los valores de los elementos seleccionados.

Complejidad temporal: $O(n)$

Complejidad espacial: $O(1)$

Es Factible C:

```
Sea pesoNecesario = 0
Desde i=0 hasta n-1
    pesoNecesario += C[i] * k[i]
retornar ( pesoNecesario <=K)
```

Ganancia C:

```
Sea valorTotal = 0
Desde i=0 hasta n-1
    valorTotal += C[i] * v[i]
retornar valorTotal
```


Generar y probar: Problema de la mochila

Unificando:

Se recorren todas las combinaciones posibles

Se verifica por cada uno si es una solución factible

Se verifica por cada uno factible es mejor que la mayor solución previamente encontrada

Complejidad temporal total: $O(2^n)$

Complejidad espacial total: $O(n)$

```
Sea C un vector representando un subconjunto
de elementos
Inicializar C
Sea maximaGanancia = 0
Sea soluciónMáxima = C

Mientras Incrementar C <> 'fin'
    Si Es Factible C y Ganancia C >
        maximaGanancia
        maximaGanancia = Ganancia C
        soluciónMáxima = C

retornar soluciónMáxima y maximaGanancia
```

Generar y probar: Permutaciones y el problema del viajante

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Generar y probar: Permutaciones

- **Espacio de soluciones**

- Trabajaremos sobre la determinación de un ordenamiento de n elementos
- Los elementos son únicos e indivisibles.
- Todos los elementos deben ser seleccionados.

- **Función generativa**

- Generación de las permutaciones de los elementos
- Corresponderá a las restricciones explícitas del problema

Ejemplo práctico: Problema del viajante de comercio

- Contamos con un conjunto de “n” ciudades a visitar.
 - Existen caminos que unen pares de ciudades.
- Cada camino
 - inicia en una ciudad “x” y finaliza en la ciudad “y”
 - tiene asociado un costo de tránsito de $w_{x,y}$.
- Partiendo desde una ciudad inicial y finalizando en la misma se quiere
 - construir un circuito que visite cada ciudad una y solo una vez minimizando el costo total.



Ejemplo práctico: Problema del viajante de comercio

El circuito parte de una ciudad inicial

la nombraremos “0”

Y debe finalizar en la misma ciudad

Cada ciudad

Recibe un identificador numérico entre 1 y n.

Para representar el orden de visita a las ciudades

Usaremos un vector C de n posición

En la primer posición la ciudad que iremos desde “0”

En la última posición la ciudad desde la que volveremos a “0”

2	3	1	5	6	7	4
---	---	---	---	---	---	---



0 → 2 → 3 → 1 → 5 → 6 → 7 → 4 → 0

Ejemplo práctico: Problema del viajante de comercio

Para un circuito “C”

Podemos determinar su factibilidad

Podemos calcular su costo

Un circuito es factible

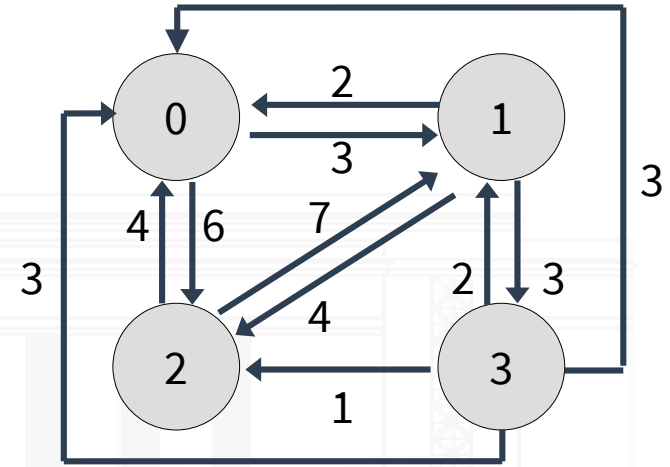
Si existe un camino que une a cada par de ciudades en él

El costo de circuito

Es la suma de los costo de los caminos entre ciudades en él

Existen $n!$ posibles circuitos

Correspondientes a generar las n permutaciones de las ciudades



2	3	1	✗
2	1	3	✓

➡ Costo: $6 + 7 + 3 + 3 = 19$

Generación de las permutaciones

- **Se generarán en orden lexicográfico.**
 - Se comenzará con un vector con los elementos ordenados de forma creciente
- **Generar la próxima permutación**
 - Se realiza en un proceso de 3 pasos
- **Finaliza el proceso**
 - cuando el vector esta ordenado de forma decreciente

Con $n=4 \rightarrow 4! = 24$

1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
...			
4	3	2	1

Generación de permutaciones: Inicialización

Para inicializar:

Establecer en cada posición $C[i]$ el elemento i ordenado lexicográficamente

Corresponde a realizar el circuito

Complejidad temporal: $O(n)$

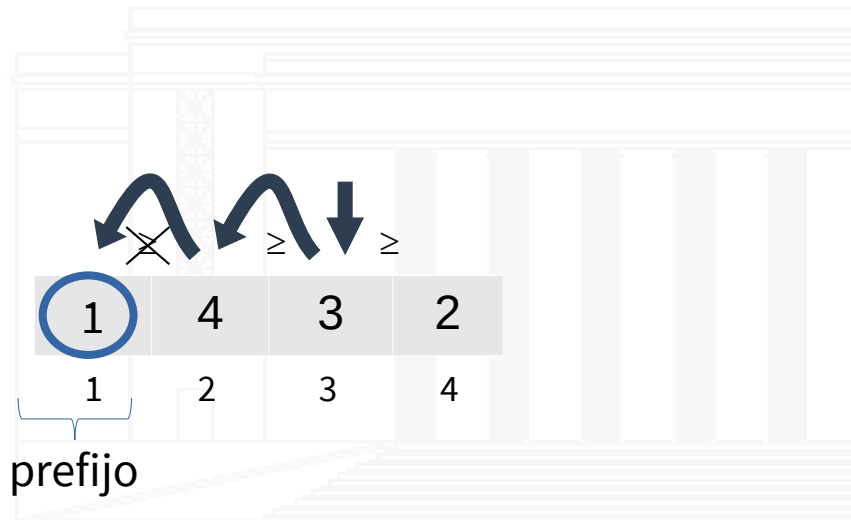
Sea C un vector representando un orden de visita de las m ciudades internas

Inicializar C :

Desde $x=1$ a n

$C[x]=x$

Generación de permutaciones: Obtener siguiente - Paso 1



Sea C un vector representando un orden de visita de las m ciudades internas

permutar C :

Sea $indice1 = n - 1$

Mientras $C[indice1] \geq C[indice1 + 1]$

$indice1 -= 1$

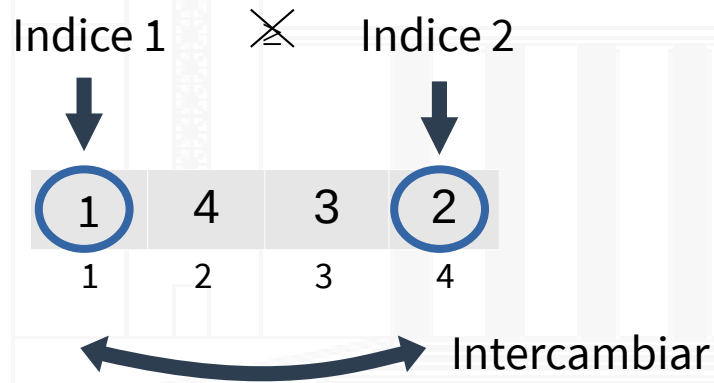
Si $indice1 == 0$

retornar 'fin'

...

Generación de permutaciones: Obtener siguiente - Paso 2

Reemplazar el último elemento del prefijo con el elemento más pequeño a su izquierda que sea superior a este



Complejidad temporal (en el peor de los casos): $O(n)$

...

```
Sea indice2=n  
Mientras C[indice1]>=C[indice2]  
    indice2 -= 1
```

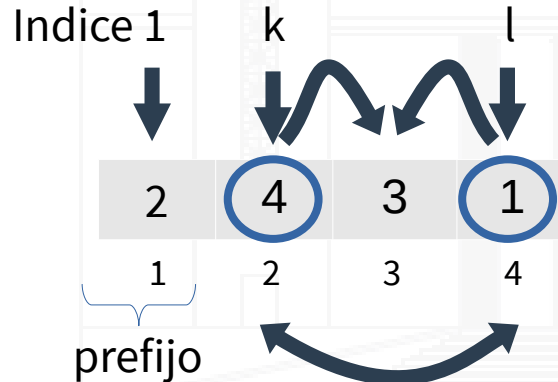
Intercambiar C[indice1] y C[indice2]

...

Generación de permutaciones: Obtener siguiente - Paso 3

Ordenar de menor a mayor los elementos por fuera del prefijo

Invertir el vector fuera del prefijo



Complejidad temporal (en el peor de los casos):
 $O(n)$

...

Sea $k = \text{indice1} + 1$

Sea $l = n$

Mientras $k < l$

Intercambiar $C[k]$ y $C[l]$

$k += 1$

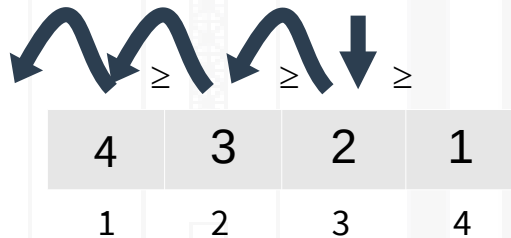
$l -= 1$

retornar C

Generación de permutaciones: Obtener siguiente - Final

La generación finaliza cuando los elementos quedan ordenados de mayor a menor

(equivalente a la ausencia de prefijo)



Complejidad temporal (en el peor de los casos): $O(n)$

Sea C un vector representando un orden de visita de las m ciudades internas

permutar C :

Sea $indice1 = n - 1$

Mientras $C[indice1] \geq C[indice1 + 1]$

$indice1 -= 1$

Si $indice1 == 0$

retornar 'fin'

...

Verificación de la solución

Costo:

Evaluar la suma de los costos de los caminos que lo integran

Opción: Considerar costo infinito si no existe un camino que une a dos ciudades en el circuito

Complejidad temporal: $O(n)$

Sea o la ciudad de inicio y finalización
Sea $w[x,y]$ el costo de transitar por el camino que une la ciudad x a la y .

Costo C :

```
Sea costoTotal = w[o,C[1]]
Desde i=1 hasta m
    costoTotal += w[C[i],C[i+1]]
costoTotal += w[C[m],o]

retornar costoTotal
```

Generar y probar: Problema del viajante de comercio

Unificando:

Comenzar con el primer circuito posible

Mientras quedan circuitos posibles obtener el próximo y evaluar si es superior al mejor previamente encontrado.

Complejidad temporal: $O((n+1)!)$

```
Inicializar C
```

```
Sea minimoCosto = Costo C
```

```
Sea solucióninima = C
```

```
Mientras permutar C <> 'fin'
```

```
    Si minimoCosto > Costo C
```

```
        minimoCosto = Costo C
```

```
        solucióninima = C
```

```
retornar solucióninima y minimoCosto
```

Generar y probar: Combinaciones y el problema del clique

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Generar y probar: Combinaciones

- **Espacio de soluciones**

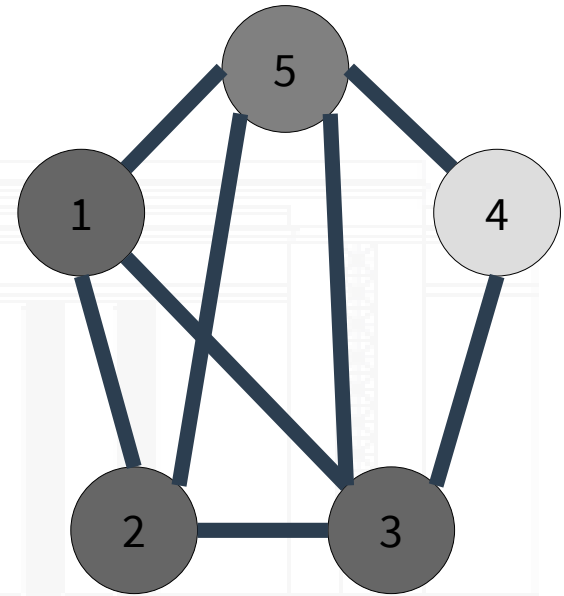
- Trabajaremos sobre la selección de un subconjunto de m elementos entre n .
- Los elementos son únicos e indivisibles.
- No importa el orden en el que son seleccionados

- **Función generativa**

- Generación de las combinaciones de los elementos
- Corresponderá a las restricciones explícitas del problema

Ejemplo práctico: Problema del clique

- **Dado un grafo no direccionado $G=(V,E)$**
 - con V su conjunto de vértices y E el de sus aristas.
 - Queremos obtener, si existe, un clique de tamaño m dentro de él.
- **Un clique es un subconjunto de vértices**
 - en el que para cualquier par de ellos existe un eje que los une
 - Equivale a decir: todos los vértices son adyacentes entre sí
 - Equivale a decir: conforman entre sí un subgrafo completo



Clique de tamaño 4: {1,2,3,5}

Ejemplo práctico: Problema del clique

Para un subconjunto M vértices de tamaño m del grafo $G=(V,E)$

Podemos verificar si cada vértices este conectado entre sí

Es un proceso en $O(m^2)$.

Si todos los vértices de M están conectados entre si

Corresponde a un clique de tamaño m

La cantidad de subconjuntos M de tamaño m en G

Se puede calcular como $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

Ejemplo práctico: Problema del clique

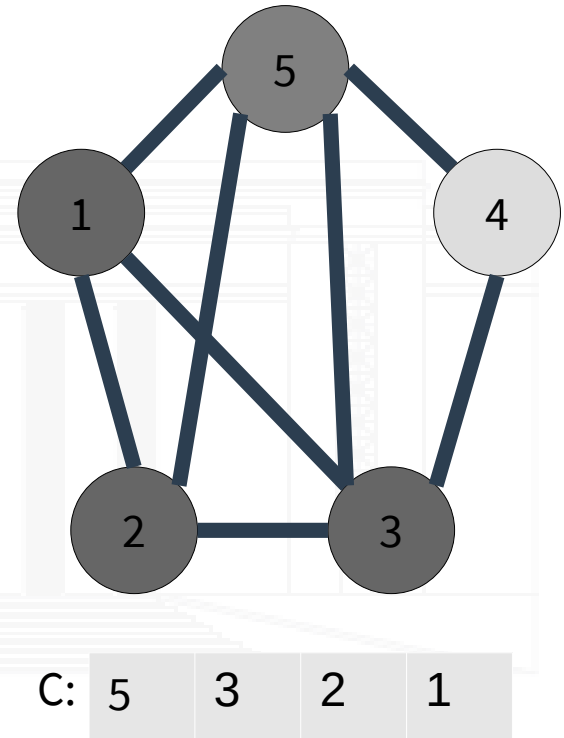
Cada vértice

Recibe un identificador numérico entre 1 y n .

Para un subconjunto M de vértices de G

Utilizaremos un vector C de tamaño m

Los vértices en el vector estarán ordenados de forma decreciente



Generación de las combinaciones de n elementos tomados de a m

- **Se generarán en orden lexicográfico.**

- Se comenzará los m elementos con menor identificador

- **Generar la próxima combinación**

Utilizará dos variables centinelas en el vector

Se buscare desde la derecha el primer elemento no contiguo y se lo incrementa

- **Finaliza el proceso**

- cuando el vector contenga los m elementos con mayor identificador
- El primer elemento no contiguo es el primer centinela

centinelas Con $n=7$ y $m=4$

0	8	4	3	2	1
0	8	5	3	2	1
0	8	5	4	2	1
0	8	5	4	3	1
0	8	5	4	3	2
0	8	6	3	2	1
0	8	6	4	2	1
...		...			
0	8	7	6	5	4

$n+1$

Generación de las combinaciones: Inicialización

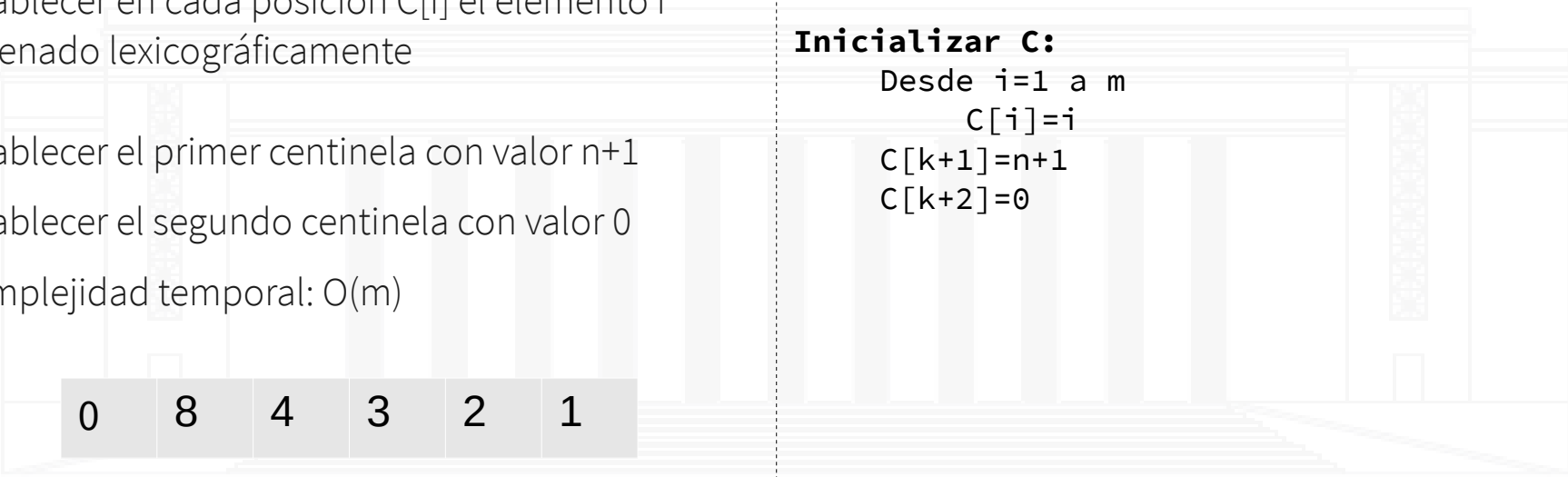
Para inicializar:

Establecer en cada posición $C[i]$ el elemento i ordenado lexicográficamente

Establecer el primer centinela con valor $n+1$

Establecer el segundo centinela con valor 0

Complejidad temporal: $O(m)$



0	8	4	3	2	1
6	5	4	3	2	1

Sea C un vector representando el conjunto de vértices a evaluar con dos posiciones adicionales.

Inicializar C :

Desde $i=1$ a m

$C[i]=i$

$C[k+1]=n+1$

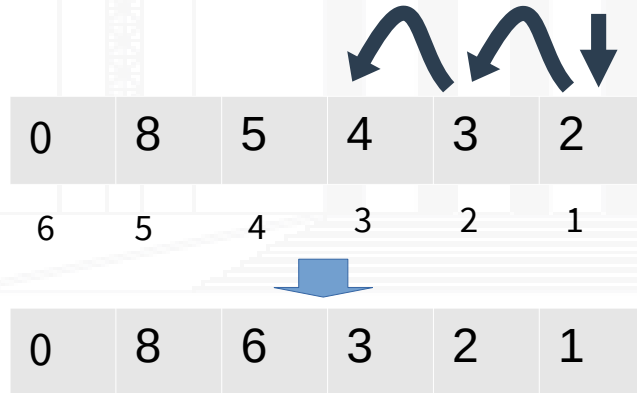
$C[k+2]=0$

Generación de las combinaciones: Obtener siguiente

Buscar el primer elemento desde la derecha no contiguo

Incrementar en 1 ese elemento

Establecer los elementos anteriores con valor contiguos comenzando por el uno.



Incrementar C:

Sea $j=1$

Mientras $C[j]+1 \leq C[j+1]$

$C[j] = j$

$j+=1$

Si $j > m$

retornar 'fin'

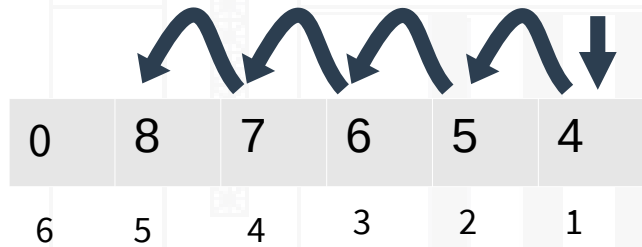
$C[j]+1$

retornar C

Generación de las combinaciones: Finalización

El proceso finaliza

Cuando todos los elementos son contiguos excepto el último en el vector



Incrementar C:

```
Sea j=1
Mientras C[j]+1==C[j+1]
    C[j] = j
    j+=1
Si j>m
    retornar 'fin'

C[j]+=1
retornar C
```

Verificación de la solución

Verificación:

Evaluar si los m elementos corresponden a vértices unidos entre sí.

Complejidad temporal: $O(m^2)$

Sea C un vector con los nodos a evaluar
Sea M la matriz de adyacencia del grafo G

Es_clique C:

Desde elemento1=1 a m

Desde elemento2=elemento1+1 a m

Si $M[C[\text{elemento1}], C[\text{elemento2}]] \neq 0$

Retornar 'No'

Retornar 'Si'

Generar y probar: Pseudocódigo

Unificando:

Por cada posible subconjunto de m vértices verificamos si corresponde a un clique.

Si lo es, lo retornamos.

Sino, se continua buscando hasta verificar todos los posibles subconjuntos.

Complejidad temporal: $O(m^2 n! / m!(n-m)!)$

Sea C un vector con los nodos a evaluar
Sea M la matriz de adyacencia del grafo G

Inicializar C

Repetir

 Si Es_clique C
 retornar C

Hasta que Incrementar $C == \text{'fin'}$

retornar 'No hay solución'

Generar y probar: Particiones de conjunto y clustering

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Generar y probar: Particiones de conjunto

- **Espacio de soluciones**

- Trabajaremos sobre la separación de n elementos en diferentes subconjuntos.
- Los elementos son únicos, indivisibles y deben pertenecer a un conjunto.
- No interesa ni el orden de los elementos dentro de cada subconjunto, ni el orden de los subconjuntos.

- **Función generativa**

- Generación de las partición es de los elementos
- Corresponderá a las restricciones explícitas del problema

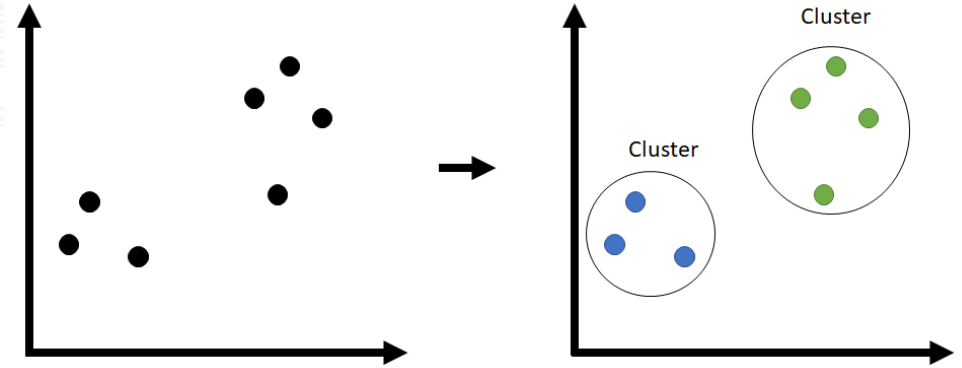
Ejemplo práctico: Clustering

Dados “n” elementos

con ciertas características mensurables y comparables

Queremos agruparlos en clusters

de forma de lograr un balance que minimice la distancia entre los puntos dentro de un mismo cluster
y maximizar la distancia entre los puntos entre clusters diferentes.



Ejemplo práctico: Clustering

No se establece una cantidad requerida de clusters

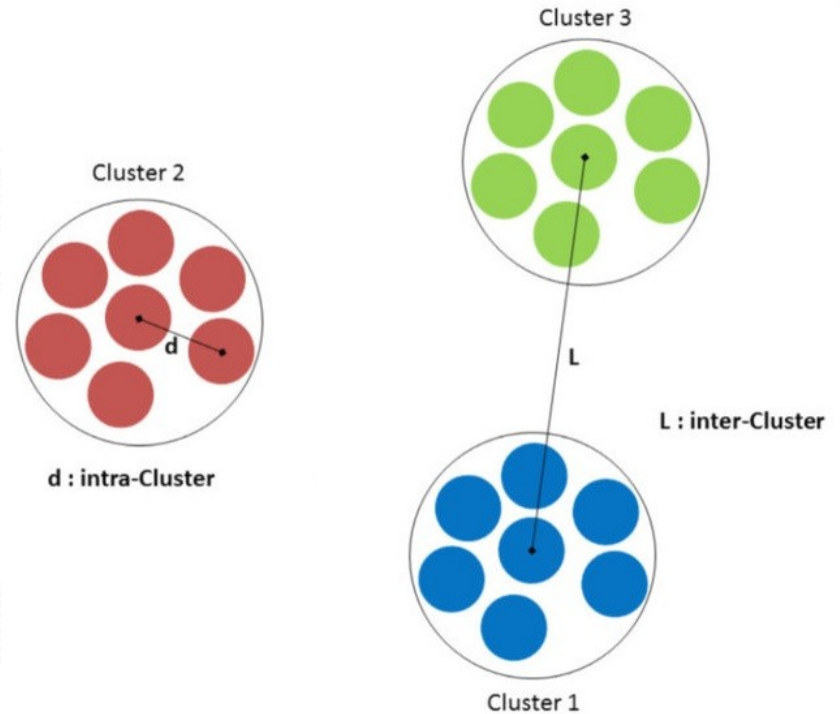
Se deben evaluar todos los clusters posibles para buscar el óptimo

Todos los elementos deben estar en un cluster

Para evaluar una posible solución

se establece una medida de similitud entre elementos

se define una medida inter cluster



Ejemplo práctico: Clustering

Podemos expresar una posible solución con un vector P de “n” posiciones.

La posición i corresponde al elemento i

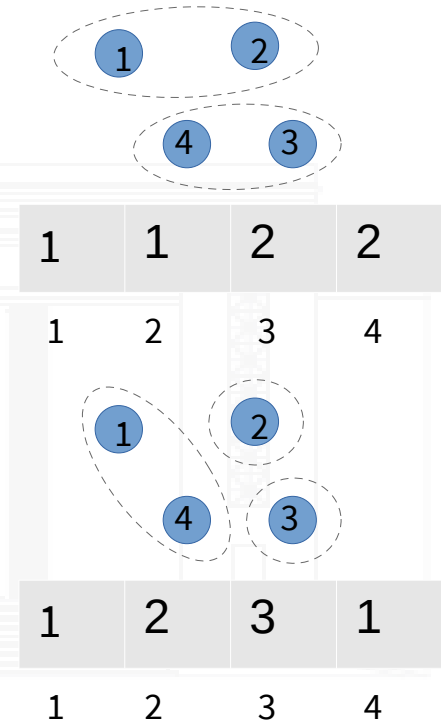
Su valor numérico corresponde al cluster (partición) donde ese asigna.

Al primer elemento siempre se asigna a la partición 1.

El valor en la posición j es menor o igual al máximo valor de los elementos anteriores más uno (cadena de crecimiento restringido o “restricted growth string”)

La cantidad de posibles particiones de “n” elementos

Se puede acotar a $\Theta\left(\frac{n}{\ln n}\right)^n$



Generación de las particiones

- **Se generarán en orden lexicográfico.**
 - Se comenzará con todos los elementos en un mismo conjunto (el 1)
- **Utilizará un vector auxiliar.**
 - En la posición i tendrá el valor máximo que aparece en alguna posición anterior
- **Generar la siguiente partición**
 - Buscará el primer elemento incrementable comenzando por la derecha
 - Lo llevará al siguiente conjunto y pondrá los recorridos en el conjunto 1
- **Finaliza el proceso**
 - Cuando cada elemento se encuentra en un conjunto diferente

1	1	1	1
1	1	1	2
1	1	2	1
1	1	2	3
1	2	1	1
1	2	1	2
1	2	1	3
...			
1	2	3	4
1	2	3	4

Generación de las particiones: Inicialización

- **Para inicializar:**

- Comenzar con el vector P y Auxiliar A con todos sus valores en 1
- Corresponde al tener todos los elementos en el mismo conjunto

P:	1	1	1	1
A:	1	1	1	1
	1	2	3	4

Sea P un vector con la partición
Sea A un vector auxiliar

Inicializar P:

Desde $i=1$ a n

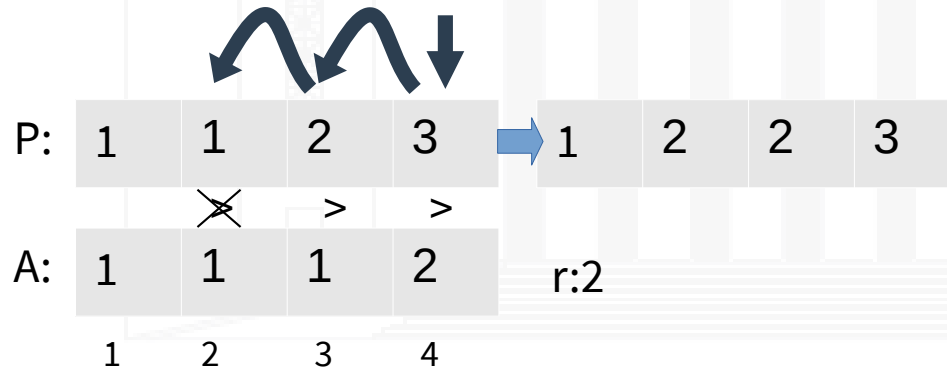
Establecer $P[i]=1$

Establecer $A[i]=1$

Generación de las particiones: Obtener siguiente – Paso 1

- **Buscar el primer elemento incrementable**

- Corresponde al primer elemento tal que no existe un anterior en un conjunto con identificador mayor o igual



Incrementar P:

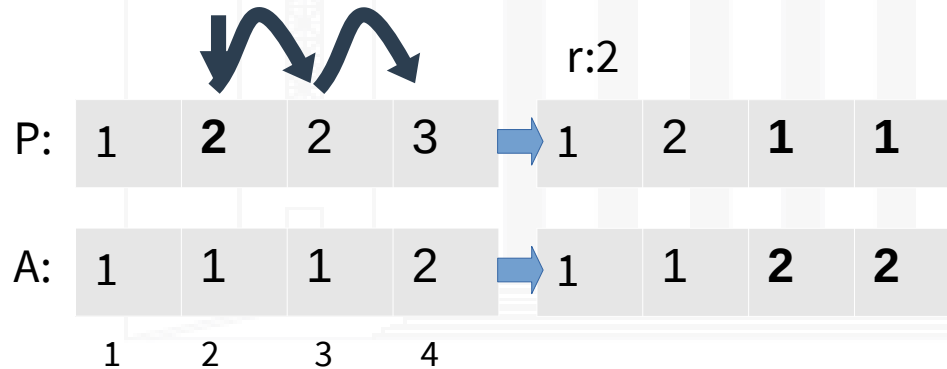
```
Sea  $j=n$   
Mientras  $P[j]>A[j]$   
     $j--$   
Si  $j==1$   
    retornar 'fin'
```

```
 $P[j] += 1$   
 $r = \max(P[j], A[j])$ 
```

...

Generación de las particiones: Obtener siguiente – Paso 2

- Pasar los elementos posteriores al incrementado a 1
 - Actualizar el vector con el mayor entre el valor del incrementado o el máximo anterior



```
...  
j += 1
```

```
Mientras j <= n
```

```
    Establecer P[j] = 1
```

```
    Establecer A[j] = r
```

```
    j += 1
```

```
retornar P
```

Generación de las particiones: Final

- El procedimiento finaliza cuando no quedan elementos incrementables
 - En ese caso se llega al elemento en la posición 1

P:	1	2	3	4
	X	>	>	>
A:	1	1	2	3
	1	2	3	4

Incrementar P:

```
Sea j=n
Mientras P[j]>A[j]
    j-=1
Si j==1
    retornar 'fin'
```

```
P[j]+=1
r = max(P[j],A[j])
```

...

Verificación de la solución

Distancia intra cluster y inter cluster

Para cada cluster se debe calcular su distancia intra cluster

Entre cada cluster se debe calcular su distancia inter cluster.

Existen diferentes alternativas para estos indicadores (excede el alcance de nuestra explicación)

Calculo de calidad de la partición

En base a los indicadores de la solución factible se debe calcular un valor de calidad de la misma.

Nuevamente existen diferentes alternativas

Se evaluará su valor y se utilizará para comparar entre particiones

Se seleccionará a aquella partición que maximice la calidad

Generar y probar: Particiones de conjunto

Unificando:

Comenzar con la primera partición posible

Mientras quedan particiones posibles obtener el próximo y evaluar si su calidad es superior al mejor previamente encontrado.

Complejidad temporal: $\Theta\left(\frac{n}{\ln n}\right)^n * \Theta(\text{CalculoCalidad}())$

```
Inicializar P
```

```
Sea valorSolucion = Evaluar P
```

```
Sea mejorSolucion = P
```

```
Mientras permutar C <> 'fin'
```

```
    Si valorSolucion > Evaluar P
```

```
        minimoCosto = Evaluar P
```

```
        mejorSolucion = P
```

```
retornar mejorSolucion y valorSolucion
```

Fuerza Bruta: Espacio de estados finitos y recorrido exhaustivo

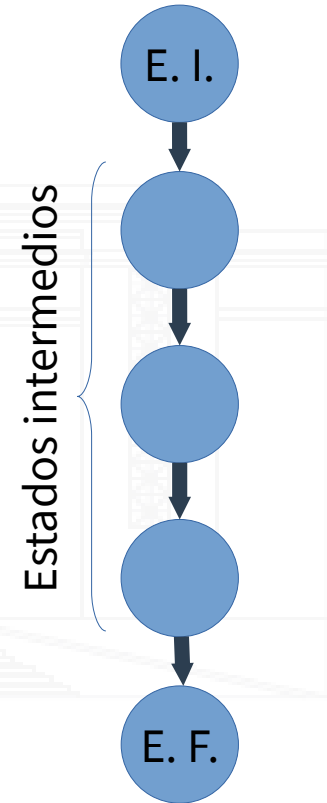
Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Procedimientos de resolución de problemas

- **Muchos procedimientos para la resolución de problemas**
 - se pueden pensar como una serie de decisiones sobre la instancia del problema que modifica su estado.
- **Llamamos:**
 - estado inicial al punto de partida
 - estados intermedios: A las diferentes elecciones/decisiones que se van realizando que transforman a la instancia
 - Estado final: A la situación a la que llegamos luego del procedimiento que llamaremos también solución.



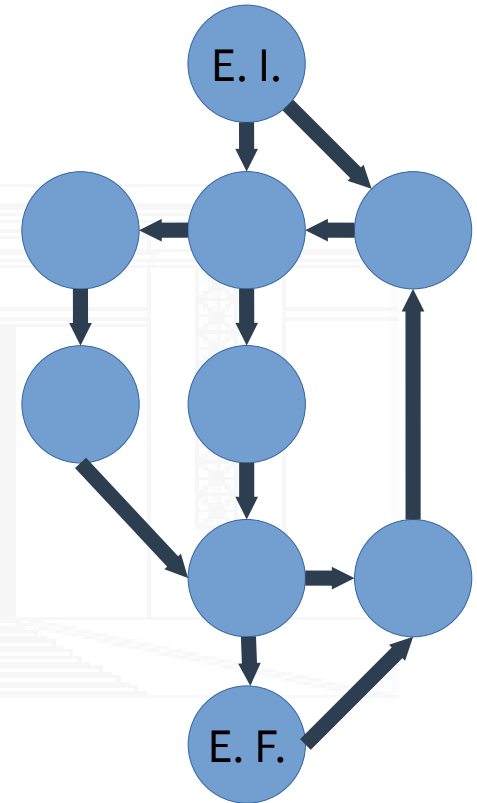
Grafo de espacio de estados

- **Un estado**

- Puede permitir diferentes decisiones
- Diferentes decisiones llevan a estados diferentes
- Es posible que pueda ser accedido desde diferentes estados

- **Llamamos:**

- Grafo de espacio de estados a esta representación del problema



Ejemplo 1: Problema de la mochila

Contamos con:

una mochila de K kilos y un conjunto E de “ n ” elementos con peso y valor

Queremos introducir un subconjunto de E

Para maximizar la ganancia y no superar la capacidad de la mochila.

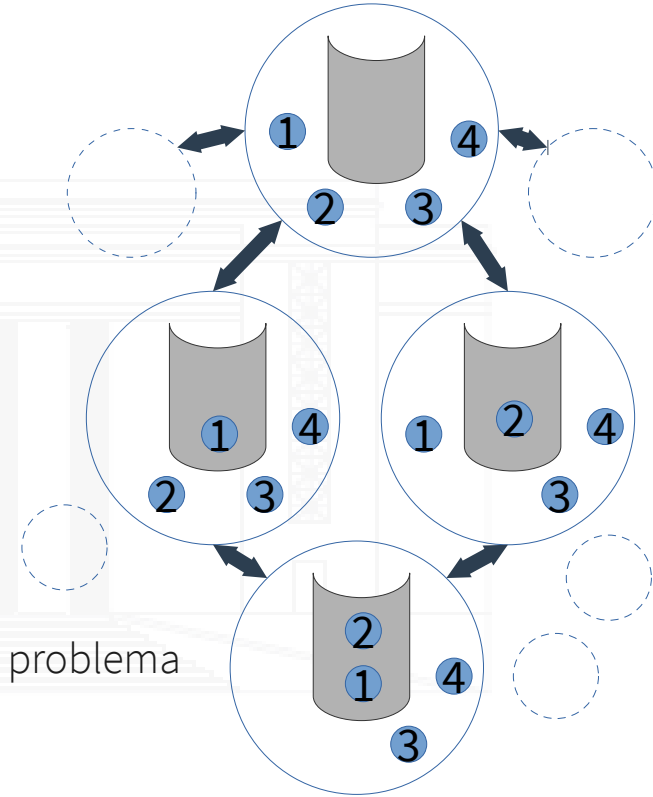
Podemos:

Definir el estado inicial como todos los elementos fuera de la mochila

Transicionar a otro estado decidiendo agregar o quitar cierto elemento

Hallar el estado final como aquel estado que cumpla el requerimiento del problema

Existen 2^n posibles estados.



Ejemplo 2: 8-puzzle

Contamos:

con un tablero de 3 filas por 3 columnas

8 piezas deslizantes numeradas del 1 al 8. (ordenadas en cualquier posición inicial en el tablero)

Cada pieza:

se puede mover de forma horizontal o vertical a la única posición vacía del tablero.

El objetivo:

Ordenar las piezas de forma que se puedan leer de corrido de forma ascendente sus números y que el espacio vacío quede en el extremo inferior derecho

Posible estado inicial:

5		3
2	8	1
4	7	6

Estado final buscado:

1	2	3
4	5	6
7	8	

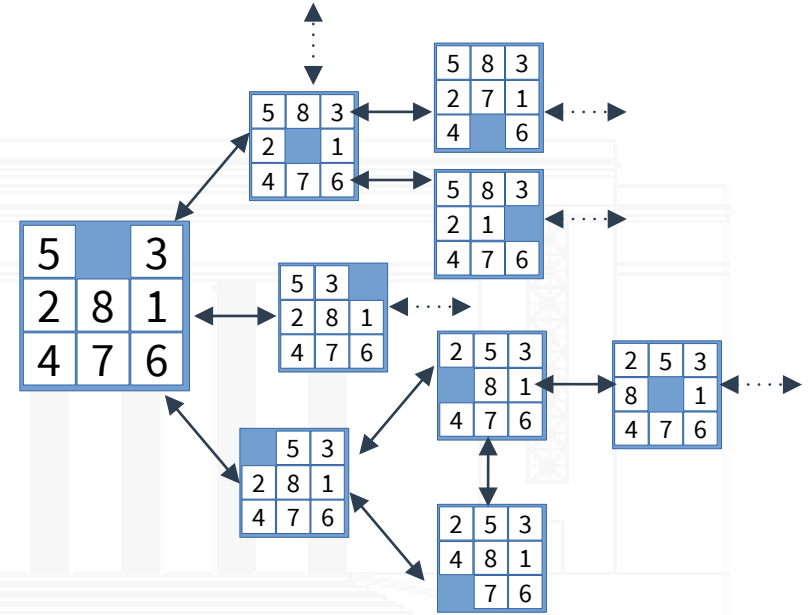
Ejemplo 2: 8-puzzle grafo de estados

Dada una instancia del problema:

Existen 9! Estados posibles

Dependiendo del estado inicial se puede acceder a la mitad de estos

La solución buscada puede no ser accesible en el grafo



Grafo de espacio de estados y problemas

Un mismo problema

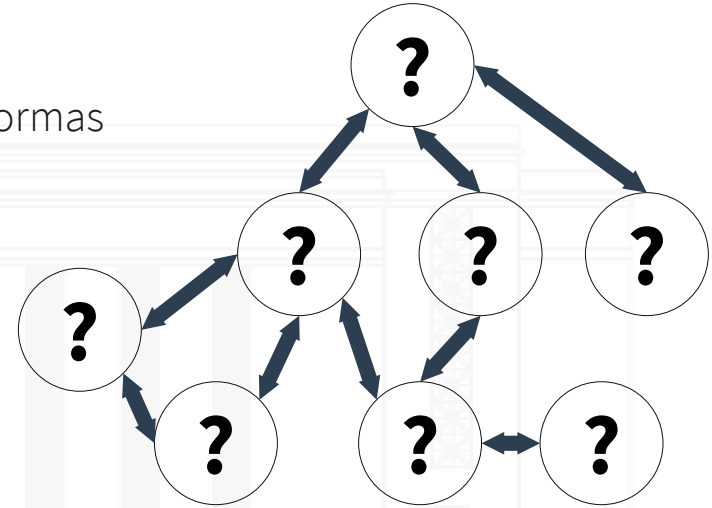
Permite (la gran mayoría de las veces) ser resuelto de diferentes formas

Cada estrategia de resolución, determina:

La estructura y tamaño del grafo de espacio de estados

La forma de construir el grafo

La forma de recorrer el grafo



Algunos métodos de búsqueda (y/o creación del grafo de estados)

Forma exhaustiva:

- Búsqueda por anchura
- Búsqueda en profundidad
- Primero el mejor (best first)
- Generar y probar
- Programación dinámica (*)
- ...

Forma heurística

- Búsqueda local
- Voraz
- ...

Backtracking: Introducción

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Jerarquía de espacio de soluciones

En un problema combinatorio

tenemos “n” elementos,

Podemos conformar diferentes posibles soluciones “combinando” los elementos.

Podemos expresar una posible solución

Como una tupla de como mucho $t \leq n$ elementos $(x_1, x_2, \dots, x_{t-1}, x_t)$

Existen un subconjunto de posibles soluciones

que comienzan con los mismos “t-1” elementos iniciales

A su vez estos forman parte de un conjunto de soluciones que inician con “t-2” mismo elementos.

Esto nos permite establecer una jerarquía en el espacio de soluciones.

Que podemos representar mediante un árbol de decisiones

Ejemplo: 8-reinas

Contamos con

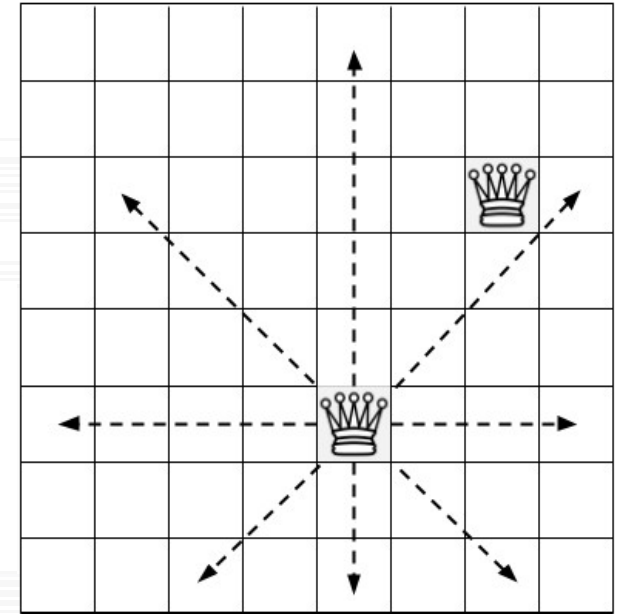
un tablero de ajedrez de 8 filas por 8 columnas.

Sabemos que

la pieza conocida como “reina” ubicada en un celda del tablero ataca a toda pieza que se encuentra en su misma fila, columna o diagonal.

Queremos

ubicar 8 reinas en el tablero de forma tal que ninguna de ellas ataque a otra.



Ejemplo: 8-reinas – Espacio de soluciones

Podemos considerar

a cada celda como un elemento

a la elección de 8 celdas como una posible solución

buscamos combinaciones de 64 elementos tomados de a 8.
(4.426.165.368 posibles soluciones)









Lo representamos como un vector donde cada elemento es mayor a los anteriores.

La posible solución (2,12,24,27,37,47,49,62)

Cumple las restricciones explícitas (8 celdas diferentes)

No cumple las restricciones implícitas (La reina en la celda 2 ataca a la reina en la celda 47)

(2,12,25,27,37,47,49,62) comparte con la solución anterior las primeras dos elecciones de celdas

1		3	4	5	6	7	8
9	10	11		13	14	15	16
17	18	19	20	21	22	23	
25	26		28	29	30	31	32
33	34	35	36		38	39	40
41	42	43	44	45	46		48
	50	51	52	53	54	55	56
57	58	59	60	61		63	64

Ejemplo: 8-reinas – Jerarquía en el espacio de soluciones

Si la primera elección corresponde a la celda 1

Existen $\binom{63}{7}$ cantidad de posibles soluciones que se pueden construir

Si la primera elección corresponde a la celda 5

Existen $\binom{59}{7}$ cantidad de posibles soluciones que se pueden construir

Si las primeras dos elecciones corresponden a la celda a las celdas 2 y 5

Existen $\binom{59}{6}$ cantidad de posibles soluciones que se pueden construir

Si las primeras dos elecciones corresponden a la celda a las celdas 4 y 20

Existen $\binom{44}{6}$ cantidad de posibles soluciones que se pueden construir

Ejemplo: 8-reinas – Jerarquía en el espacio de soluciones

Podemos representar las elecciones (y los estados del problema) como un árbol

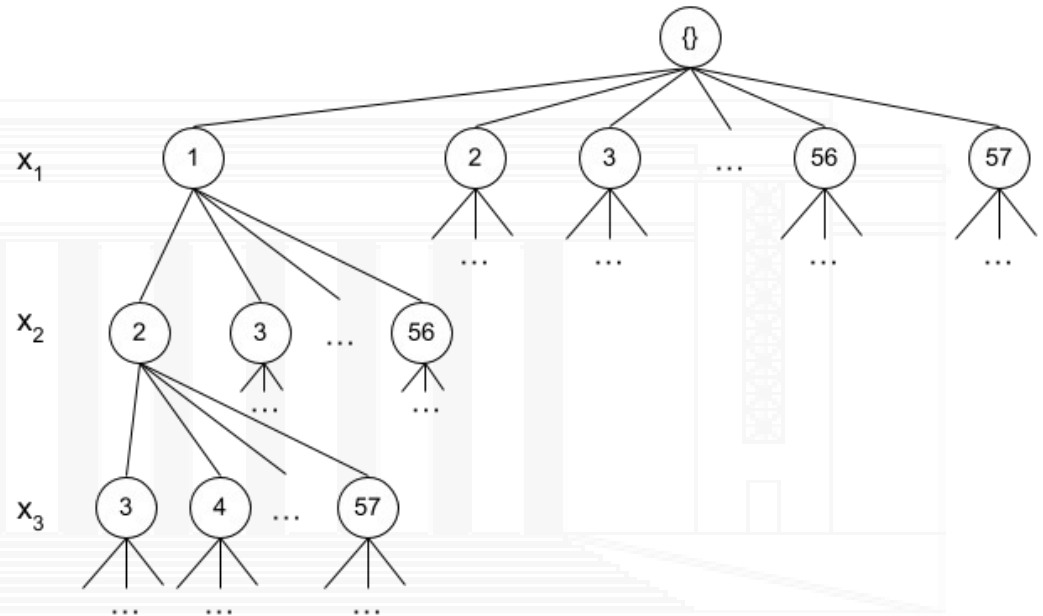
La raíz corresponde al tablero vacío

Los nodos en el nivel “i” en el árbol a seleccionar una celda en la i-esima posición del vector de soluciones

La profundidad máxima del árbol será 8

Existirán $\binom{64}{8}$ hojas en el árbol

Recorrer el árbol de forma exhaustiva nos permite encontrar todas las posibles soluciones



Clasificación de estados en árbol de estados

Estados del problema

Corresponden a todos los nodos del árbol de estados

Estados solución

Subconjunto de estados del problema

Corresponde a aquellos que cumplen con las restricciones explícitas del problema

Estados respuesta

Subconjunto de estados solución

Corresponde a aquellos que cumplen con las restricciones implícitas del problema

Propiedad de corte

Tamaño del árbol de estados

La cantidad de estados del problema es igual o mayor que la cantidad de estados solución

Muchas veces no es necesario recorrer todo el árbol

En ocasiones al realizar las primeras j decisiones podemos saber si existe una posible respuesta al problema que las incluya

Llamaremos propiedad de corte

A la posibilidad de evaluar en un estado del problema la ausencia de algún estado respuesta descendiente del mismo

Aplicaremos una función límite a un estado del problema para evaluar la propiedad de corte

En caso de aplicarse la propiedad de corte, desistiremos la exploración de los descendientes de ese nodo del árbol

Diremos que “PODAMOS” el árbol.

Ejemplo: 8-reinas – Propiedad de corte

Considerar el estado del problema (1,2)



Corresponde a incluir una reina en la celda 1 y otra en la 2

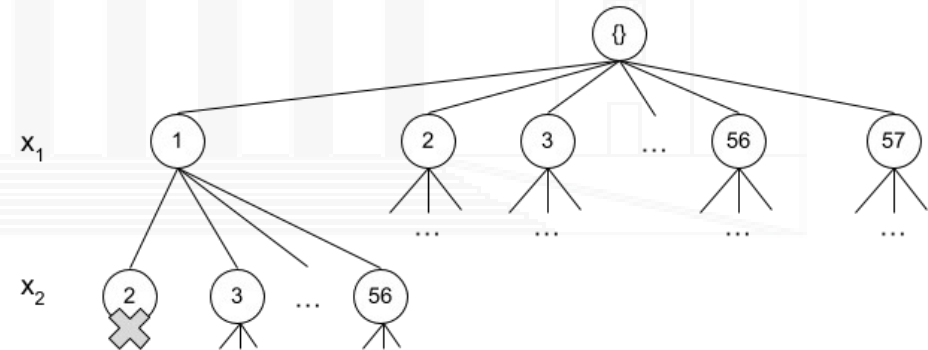
Estas se atacan entre sí

Cualquier estado solución (1,2,x₃,x₄,...,x₈)

No sera un estado respuesta (no cumple con las restricciones implícitas)

Por lo tanto, por propiedad de corte, podemos evitar la exploración de los estados del problema descendientes de (1,2)

		3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



Recorrido y creación del árbol de estados del problema

Generaremos el árbol de estados de forma dinámica

Partiremos de la raíz

Por cada posible estado de problema descendiente se generarán sus descendientes

Existen diferentes maneras de recorrer el árbol

Utilizaremos Depth-First Search

Se adentrará lo más que puede en la profundidad del árbol

Se evalúa si se hay encontrado una solución.

Se retrocede (Backtrack) cuando no quedan caminos por recorrer en la rama actual

Este retroceso ocurre cuando todos los descendientes ya fueron explorados o podados

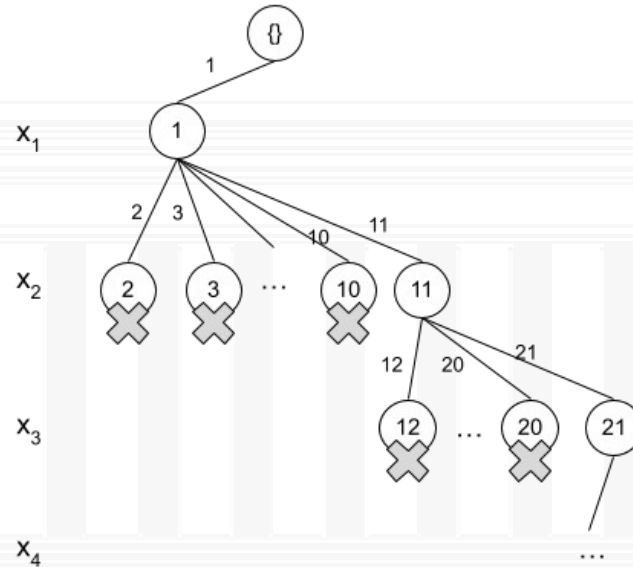
Ejemplo: 8-reinas – Recorrido del árbol




Se presentan primeros pasos en el recorrido del árbol

Al incluir la reina en la celda 1 se podan todos los nodos del problema descendentes que tienen a la segunda en las celdas 2 a 10.

Al incluir la segunda reina en la celda 11 se podan aquellos descendentes que tiene a la tercera en las celdas todas las soluciones que tienen

El orden de exploración se muestra como una etiqueta en el eje del árbol



	2	3	4	5	6	7	8
9	10		12	13	14	15	16
17	18	19	20		22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Backtracking

Se conoce como Backtracking

Al mecanismo de crear dinámicamente el árbol de estados del problema

Recorrerlo mediante Depth-First Search

Podar la exploración utilizando la propiedad de corte

El algoritmo fue trabajado y explorado inicialmente entre la década del 50 y 60

Se reconoce a D. H. Lehmer en los 50 como quien acuñó esta denominación.

A R. J. Walker en los 60 por darle forma algorítmica.

A S. Golomb y L. Baumert por demostrar su aplicabilidad en una gran variedad de situaciones

Backtracking – Pseudocódigo recursivo

Backtrack (estadoActual):

Si estadoActual es un estado resultado
retornar estadoActual

Si estadoActual supera la propiedad de corte
Por cada posible estadoSucesor de estadoActual
resultado = Backtrack(estadoSucesor)

si resultado es un estado resultado
retornar resultado

retornar vacio

Sea estadoInicial la raiz del arbol de estados
Backtrack(estadoInicial)

Backtracking: Problema de la mochila

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Problema de la mochila

- **Contamos con:**
 - una mochila con una capacidad de K kilos
 - un subconjunto del conjunto E de “ n ” elementos
- **Cada elemento i tiene:**
 - un peso de k_i kilos
 - un valor de v_i .
- **Queremos seleccionar un subconjunto de E**
 - con el objetivo de maximizar la ganancia.
 - el peso total seleccionado no puede superar la capacidad de la mochila.



Problema de la mochila – identificación de los elementos

- Podemos asignarle a cada elemento identificado un único (un valor entero)
- Asignar un orden a los elementos según su identificador

1	2	3	4	5	6	7	8	9	10	11	12	13
k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}
V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}



Problema de la mochila – Representación de una solución

- Podemos expresar una posible solución como una lista de como mucho “n” elementos
- La lista estará ordenada de menor a mayor según el identificador del elemento
- La lista vacía corresponde a la mochila vacía



A diagram of a backpack is shown in the background. Inside the backpack, there is a 2x3 grid of items. The top row contains items with identifiers 1, 2, and 3. The bottom row contains items with values 3, 7, and 10. A blue arrow points from this grid towards the right, indicating a selection process.

1	2	3
3	7	10

$$¿ k_3 + k_7 + k_{10} \leq K ?$$

$$\text{Valor} = v_3 + v_7 + v_{10}$$

Árbol de estados del problema

- La raíz representa la mochila vacía
- Un nodo corresponde a ingresar un determinado elemento a la mochila
 - Se agrega a los anteriores incluidos
- Los descendientes de cada nodo
 - corresponden a los posibles elementos a elegir posteriores (según su identificador) al agregado en el mismo
 - Al incluir un elemento ocupa su peso en la mochila y brinda su valor
- Llamaremos al árbol resultando como árbol combinatorio.
 - Todos los estados del problema corresponden a estados solución
 - El total de estado del problema corresponde a 2^n

Ejemplo: Árbol de estados del problema

Supongamos la siguiente instancia



1

2

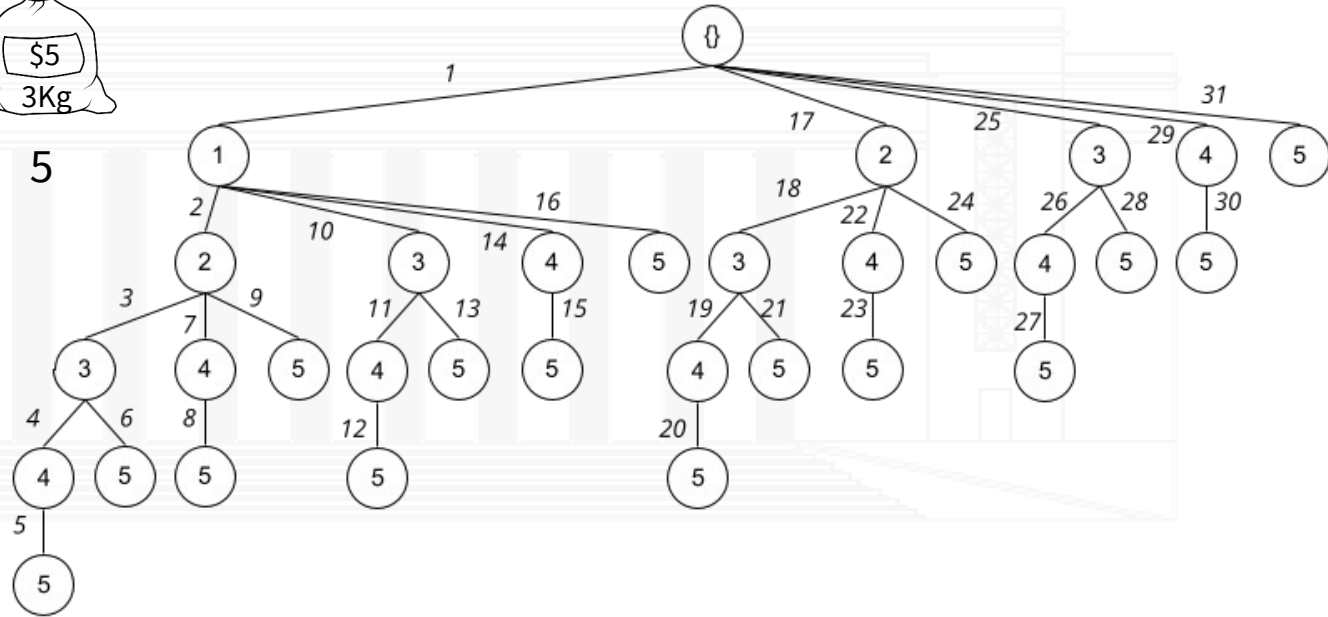
3

4

5

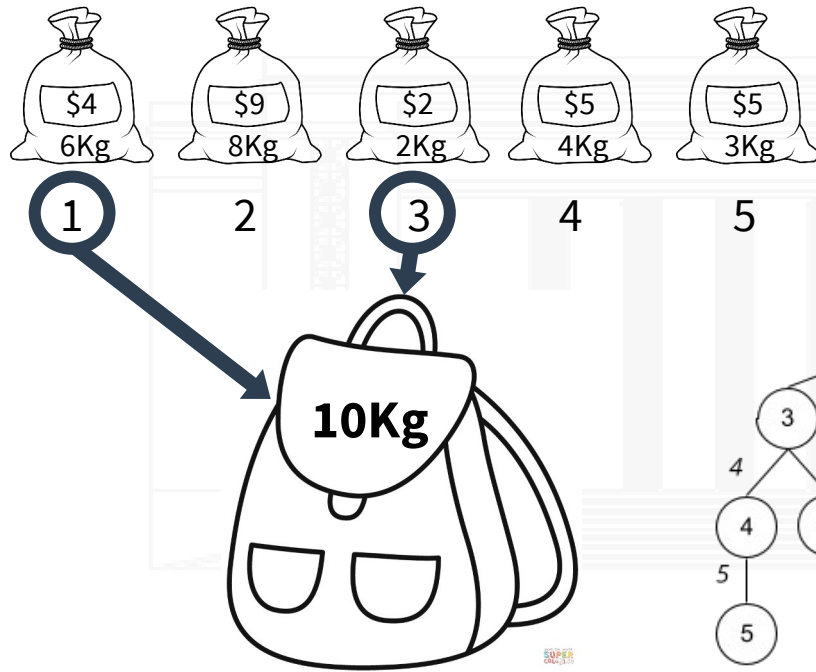


Los estados se pueden representar como un árbol combinatorio:

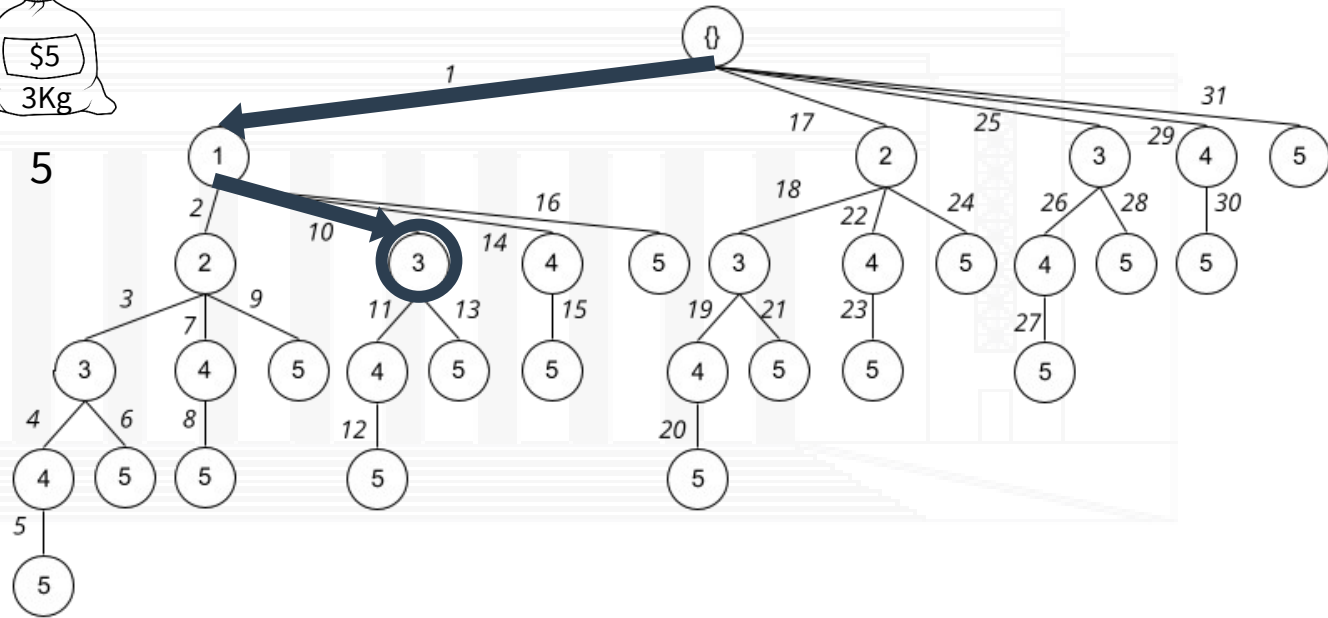


Ejemplo: Árbol de estados del problema

Posible solución:



Se corresponde al recorrido en el árbol:



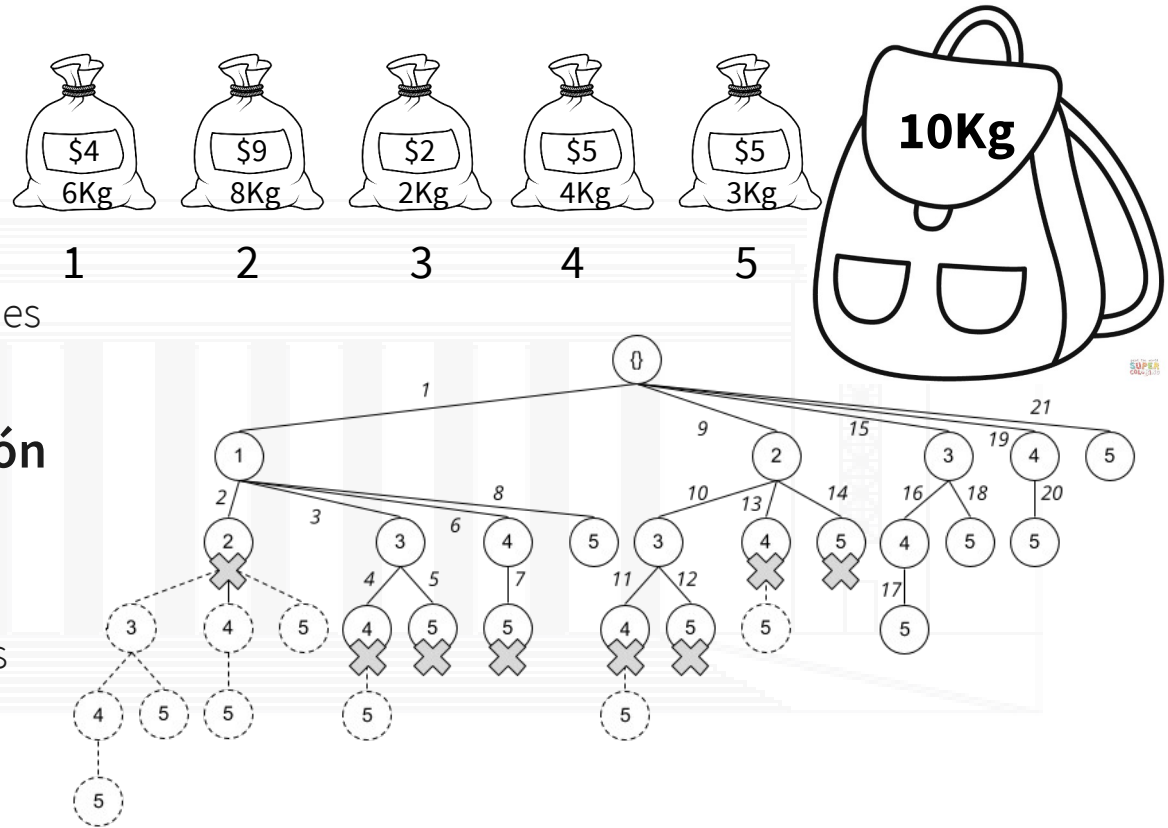
Poda del árbol

No todas las combinaciones de elementos son posibles

Si un conjunto de elementos supera la capacidad de la mochila, agregar adicionales también lo hará

La función límite verifica la superación de la capacidad

En caso de ausencia, poda el nodo y desestima la inspección de todas las ramas del árbol que las contiene



Backtracking – Pseudocódigo

Sea elementos un vector de los elementos disponibles
Sea mochila una lista inicialmente vacía con los elementos seleccionados.
Sea maximaGanancia la cantidad máxima obtenida en la mochila
Sea maximaCombinacion los elementos seleccionados para obtener la máxima ganancia

```
mochila={}.  
maximaGanancia=0  
maximaCombinacion=mochila
```

```
Backtrack(mochila)
```

Backtracking – Pseudocódigo (II)

Backtrack (mochila):

```
Si mochila no supera la capacidad disponible
  Sea ganancia la suma de los elementos en la mochila
  si ganancia > maximaGanancia
    maximaGanancia = ganancia
    maximaCombinacion = mochila

Si mochila tiene capacidad disponible
  Sea ultimoElemento el ultimo elemento añadido en la mochila
  Por cada elemento posterior a ultimoElemento en elementos
    Agregar elemento a mochila
    Backtrack(mochila)

  Quitar elemento de mochila
```

Complejidad Temporal

En el peor de los casos no es posible podar nodos del arbol

Debemos recorrer cada uno de los nodos del árbol con una complejidad $O(2^n)$

En los nodos en el peor de los casos debemos hacer un trabajo $O(n)$

Para resguardar una mejor solución encontrada

Para calcular el peso y ganancia acumulado (se podría realizar en $O(1)$)

La multiplicación de ambas complejidades nos brinda la complejidad temporal del algoritmo.

Complejidad Espacial

Por la implementación recursiva

Por cada llamado en profundidad en el árbol incluimos el consumo de memoria adicional

La memoria utilizada

Es proporcional a la profundidad máxima de la recursión generada

Para el árbol combinatorio la profundidad máxima es “n”.

En cada nivel de profundidad

Se agrega un elemento a la mochila

Se realizan cálculos que requieren $O(1)$ de almacenamiento

Por lo que la complejidad espacial es $O(n)$

Backtracking: Problema del viajante de comercio

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Problema del viajante de comercio

- Contamos con un conjunto de “n” ciudades a visitar.
 - Existen caminos que unen pares de ciudades.
- Cada camino
 - inicia en una ciudad “x” y finaliza en la ciudad “y”
 - tiene asociado un costo de tránsito de $w_{x,y}$.
- Partiendo desde una ciudad inicial y finalizando en la misma se quiere
 - construir un circuito que visite cada ciudad una y solo una vez minimizando el costo total.



Representación del circuito

El circuito parte de una ciudad inicial

la nombraremos “0”

Y debe finalizar en la misma ciudad

Cada ciudad

Recibe un identificador numérico entre 1 y n.

Para representar el orden de visita a las ciudades

Usaremos un vector C de n posición

En la primer posición la ciudad que iremos desde “0”

En la última posición la ciudad desde la que volveremos a “0”

2	3	1	5	6	7	4
---	---	---	---	---	---	---



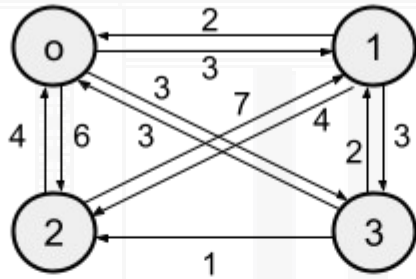
0 → 2 → 3 → 1 → 5 → 6 → 7 → 4 → 0

Árbol de estados del problema

- La raíz representa el comienzo del viaje partiendo de la ciudad inicial
- Cada nodo representa la inclusión de una ciudad en el recorrido que no fue previamente visitada
 - Tiene como posibles estados descendientes las posibles elecciones de próximas ciudades (restringidas por las previamente visitadas).
 - Al elegir una opción se debe sumar el costo del traslado
- El camino desde la raíz hasta el nodo representa el recorrido realizado desde la ciudad inicial hasta el momento
 - El costo del recorrido es la suma de los costos de los trayectos entre las ciudades realizadas
- El árbol para n ciudades tiene un total de $n!$ estados solución

Ejemplo: Árbol de estados del problema

Supongamos la siguiente instancia del problema:

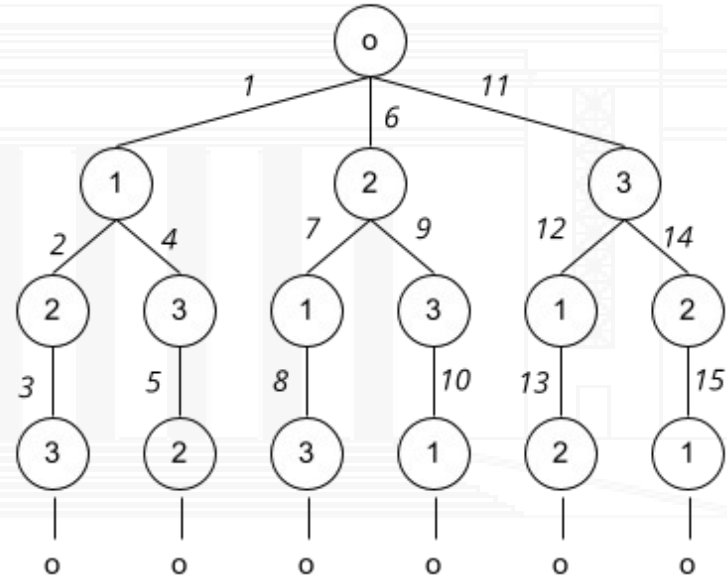


Tenemos:

$3! = 6$ estados solución (permutaciones de 3 ciudades)

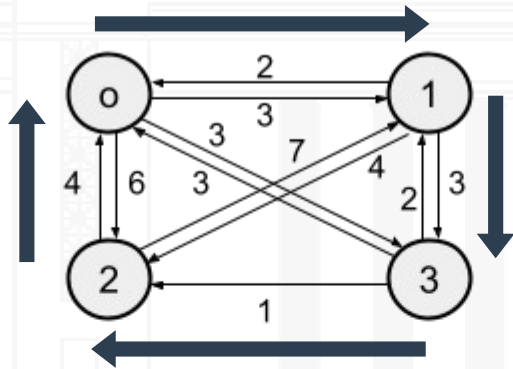
$1 + \sum_{i=1}^n \prod_{j=1}^i (n+1) - i$ estados del problema

Los estados se pueden representar como un árbol de permutaciones:



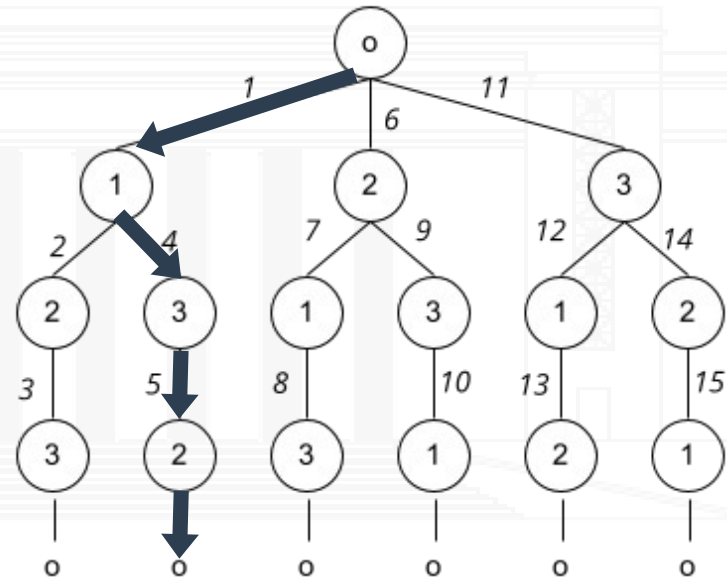
Ejemplo: Árbol de estados del problema

Posible circuito:



Costo: 11

Equivale al recorrido en el árbol:



Poda del árbol

No todos los caminos son posibles

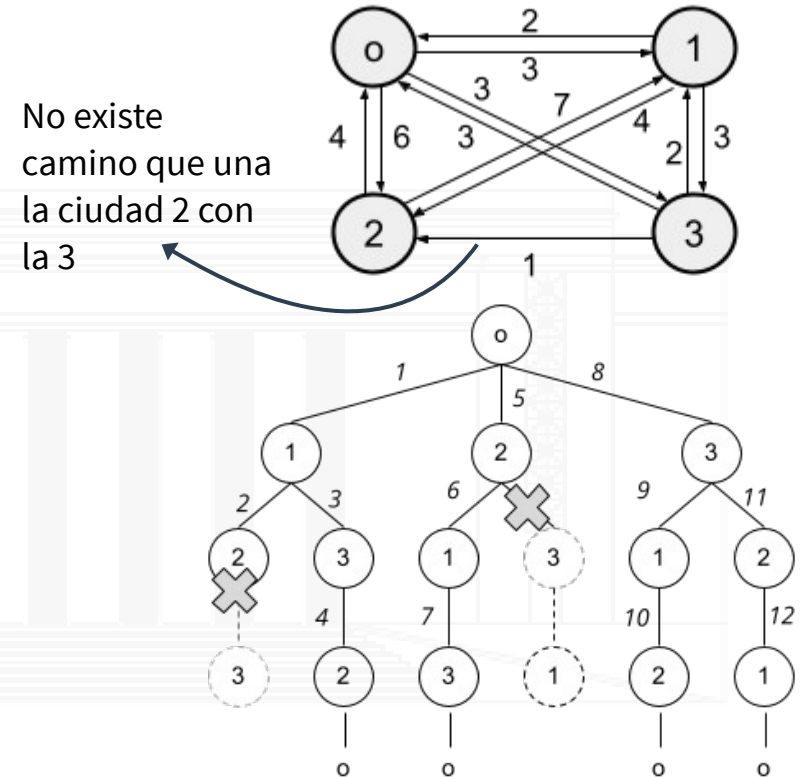
Si no existe un camino que une dos ciudades a y b podemos podar cualquier circuito que lo utilice

La función límite verifica la existencia de un camino que una a las ciudades

En caso de ausencia, poda el nodo y desestima la inspección de todas las ramas del árbol que las contiene

Si el grafo tiene una cantidad importante de nodos no comunicados

La poda reduce considerablemente el árbol de estados.



Backtracking - Pseudocódigo

Sea $C[x,y]$ el costo de ir de la ciudad x a la y
Sea $A[y]$ la lista de ciudades adyacentes de la ciudad y .
Sea camino el recorrido realizada hasta el momento
Sea minimoCosto el costo del camino minimo encontrado
Sea minimoCamino el circuito de menor costo encontrado

camino={o}.
minimoCosto=infinito
minimoCamino={}

Backtrack(camino)

Backtracking – Pseudocódigo (II)

Backtrack (camino):

Sea x la ultima ciudad visitada.

Si longitud del camino es $n-1$

Si la ciudad o se encuentra en $A[x]$

Agregar al final de camino la ciudad o

Sea costoCamino la suma de los costos de camino

Si $\text{costoCamino} < \text{minimoCosto}$

$\text{minimoCosto} = \text{costoCamino}$

$\text{minimoCamino} = \text{camino}$

Remover o de camino

Sino

Por cada ciudad y en $A[x]$ no visitada previamente excepto “ o ”

Agregar y a camino

Backtrack(camino)

Quitar y de camino

Complejidad Temporal

En el peor de los casos no es posible podar nodos del arbol

Debemos recorrer cada uno de los nodos del árbol con una complejidad $O\left(\sum_{i=1}^n \prod_{j=1}^i (n+1)-i\right)$

En los nodos en el peor de los casos debemos hacer un trabajo $O(n)$

Para obtener los siguientes nodos posibles de visitar

Para calcular el costo del circuito

La multiplicación de ambas complejidades nos brinda la complejidad temporal del algoritmo.

Complejidad Espacial

Si utilizamos una implementación recursiva

Por cada llamado en profundidad en el árbol incluimos el consumo de memoria adicional

La memoria utilizada

Es proporcional a la profundidad máxima de la recursión generada

Para el árbol de permutaciones la profundidad máxima es “n”.

En cada nivel de profundidad

Se agrega una nueva ciudad al vector

Se realizan cálculos que requieren $O(1)$ de almacenamiento

Por lo que la complejidad espacial es $O(n)$

Backtracking: K coloreo de grafos

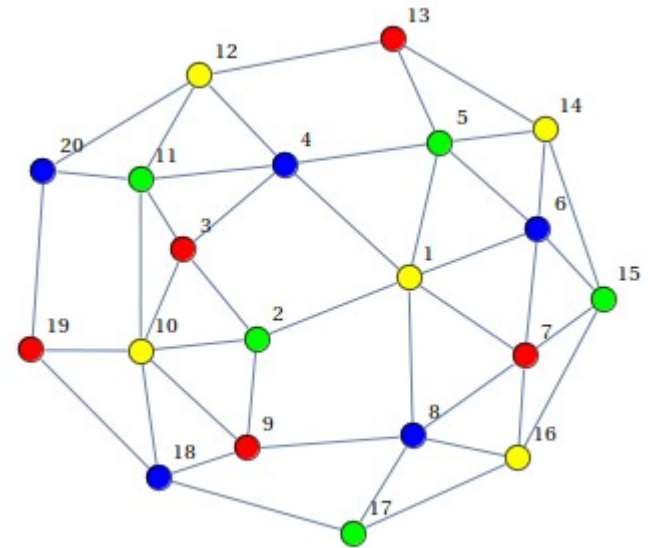
Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

K coloreo de grafos

- **Contamos con un**
grafo $G=(V,E)$ con “n” vértices y “m” ejes.
- **Queremos**
asignarles no más de K colores a sus vértices
- **De modo que**
para cualquier par de vértices adyacentes no compartan el mismo color.



Representación de un coloreo

Estableceremos un orden de 1 a “n” para los vértices del grafo.

Llamaremos v_i al vértice en la posición i

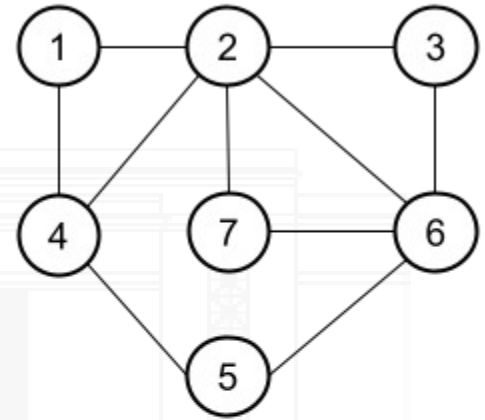
Utilizaremos k etiquetas

para representar los color

Utilizaremos un vector de n posiciones

Para representar un coloreo utilizando como mucho k etiquetas

En la posición i del vector se definirá la etiqueta del vértice v_i



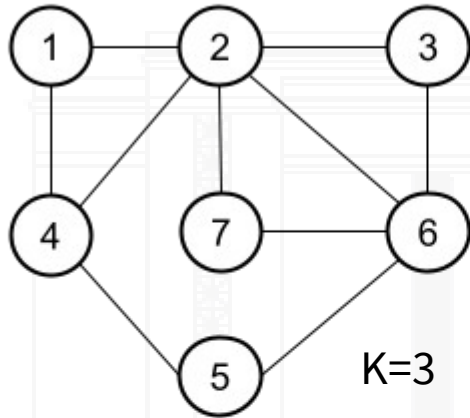
1	2	3	4	5	6	7
1	2	1	3	1	3	1

Árbol de estados del problema

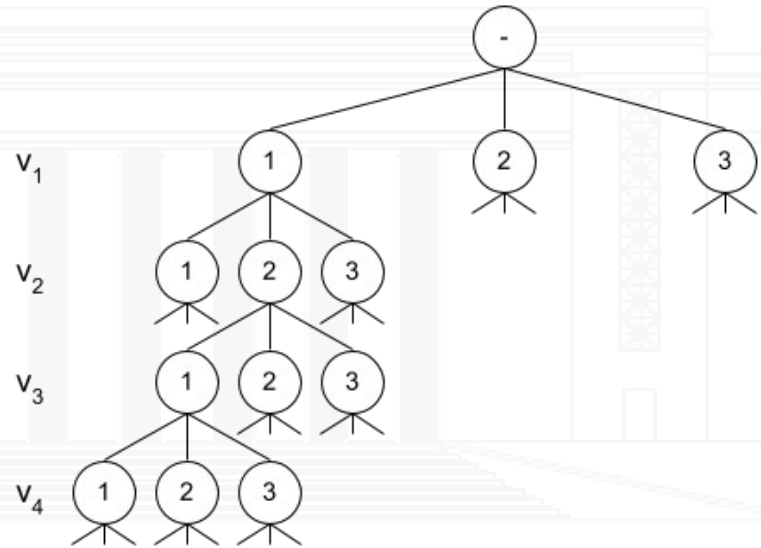
- La raíz representa al grafo sin ninguna etiqueta en sus vértices
- Un nodo a profundidad i corresponde a la elección de una etiqueta al vértice v_i
 - Se agrega a los anteriores incluidos
- Los descendientes de cada nodo
 - Son k nodos, uno por cada etiqueta
 - Al etiquetar (colorear) un nodo podemos determinar si es un coloreo compatible con los antes coloreados
- El árbol resultando corresponderá a un k -ario.
 - Los estados a profundidad n del árbol corresponde a los estados solución
 - El total de estados del problema corresponde a k^n

Ejemplo: Árbol de estados del problema

Supongamos la siguiente instancia del problema:

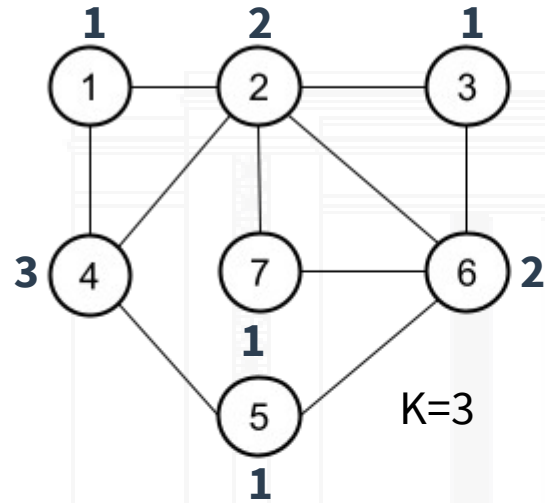


Los estados se pueden representar como un árbol k-ario (porción):

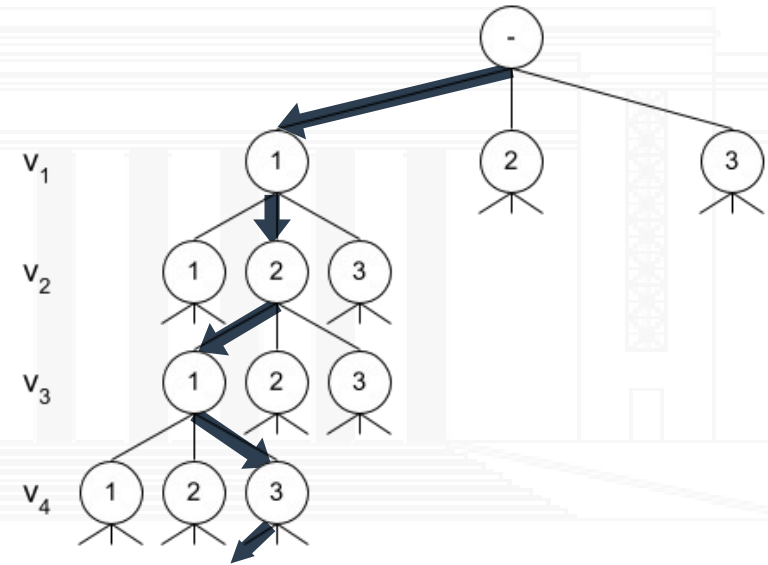


Ejemplo: Árbol de estados del problema

Posible coloreo:



Equivale al recorrido en el árbol
(porción):



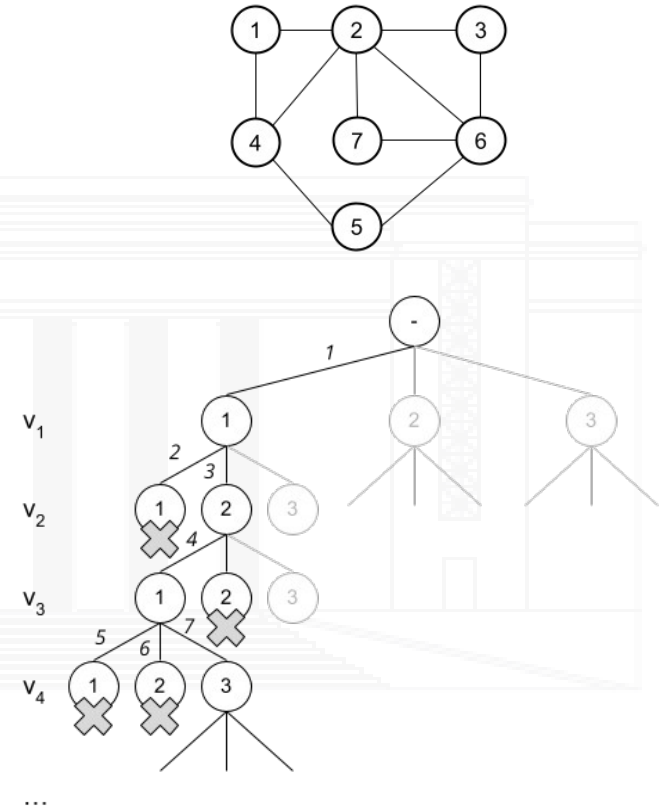
Poda del árbol

No todos los coloreos son posibles

Si asignamos un color a un vértice del grafo que tiene el mismo que un vértice previamente asignado, podemos podar todos los coloreos que los utilicen

La función limite verifica las etiquetas de los vértices adyacente del último vértice etiquetado

En caso de coincidencia, poda el nodo y desestima la inspección de todas las ramas descendientes del mismo



Backtracking - Pseudocódigo

Sea $C[x]$ el color asignado al vértice x
Sea $A[x]$ la lista de vértices adyacentes de la vertice x .

```
vertice=1  
Si Backtrack(vertice)  
    Imprimir  $C[x]$  para todo vertice  $x$  del grafo  
Sino  
    Imprimir 'no hay un coloreo posible'
```

Backtracking – Pseudocódigo (II)

Backtrack (nroVertice):

```
Por cada color disponible
  C[nroVertice]=color

  Sea coloreoValido = true
  Por cada vertice x en A[nroVertice]
    Si C[nroVertice] == C[x]
      coloreoValido =false

  Si (coloreoValido y nroVertice==n)
    retornar true
  Si (coloreoValido y nroVertice<n)
    Si Backtrack(nroVertice+1)
      retornar true
  Quitar color a C[nroVertice]
Retornar false
```

Complejidad temporal

En el peor de los casos no es posible podar nodos del arbol

Debemos recorrer cada uno de los nodos del árbol con una complejidad $O(k^n)$

En los nodos en el peor de los casos debemos hacer un trabajo $O(n)$

Correspondiente a comparar el etiquetado de los vértices adyacentes del grafo

En los nodos debemos hacer un trabajo $O(k)$

Para generar los nodos descendientes por cada color posible

La Complejidad final corresponde a la multiplicación de estas complejidades

Complejidad espacial

Por la implementación recursiva

Por cada llamado en profundidad en el árbol incluimos el consumo de memoria adicional

La memoria utilizada

Es proporcional a la profundidad máxima de la recursión generada

Para el árbol combinatorio la profundidad máxima es “n”.

En cada nivel de profundidad

Se define un color a un vértice

Se realizan cálculos que requieren $O(1)$ de almacenamiento

Por lo que la complejidad espacial es $O(n)$

Branch & Bound: Introducción

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Branch & Bound (Ramificación y poda)

- **Variante de Backtracking para problemas de optimización**
 - Se busca responder como resultado la maximización o minimización de cierto valor.
- **Utiliza un árbol de espacio de estados.**
 - Recorre todo el espacio de estados del problema (en el peor de los casos)
 - Es considerado un algoritmo de búsqueda exhaustiva.
- **Utiliza la propiedad de corte para la poda de los estados en el árbol**
- **Agrega una función costo**
 - que permite un criterio de poda adicional
 - Que determina el orden de inspección de los estados
- **Permite el recorrido del árbol según diferentes métodos**

Problemas de optimización y soluciones óptimas

En el árbol de estados pueden existir varios estados respuesta

Estos estados cumplen las restricciones implícitas y explícitas del problema.

Cada uno de ellos tendrá un valor característico (de acuerdo al problema).

Corresponde a soluciones factibles

Una solución factible es una solución óptima

Si Maximiza/minimiza (según lo buscado) el valor característico.

Al recorrer el árbol de estados

Guardamos la mejor solución factible encontrada hasta el momento

Al finalizar la exploración tendremos la solución óptima (si existe)

Poda del árbol

Para expresar una posible solución

utilizaremos una tupla de como mucho $t \leq n$ elementos $(x_1, x_2, \dots, x_{t-1}, x_t)$

Un estado del problema

Contendrá una tupla parcial (o total) de la tupla solución

Corresponderá a un prefijo de un conjunto de estados respuesta (soluciones factibles)

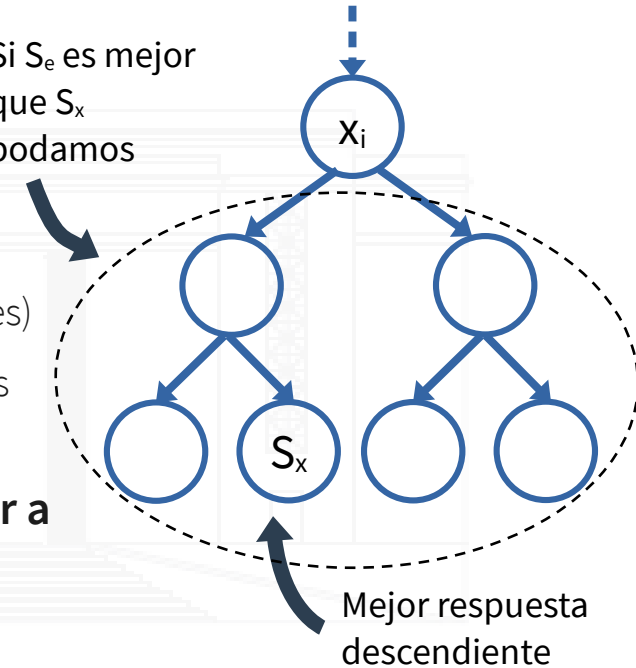
Existirá un estado respuesta dentro de ese conjunto que sera el mayor (o menor si es un problema de minimización) entre ellos

Si el “mejor” estado respuesta descendiente del estado actual es peor a la mejor solución previamente encontrada

Podemos evitar inspeccionar esas ramas del árbol

S_e : Mejor respuesta encontrada

Si S_e es mejor
que S_x
podamos



Función costo

Llamaremos función de costo

A la función que, dado un estado del problema determina el posible valor de un posible estado respuesta descendiente del mismo.

La función costo

Realiza una estimación el mejor valor descendiente posible (un valor aproximado)

Puede sobre estimar este valor, pero no debe infravalorarlo.

En caso de hacerlo se podría podar ramas incorrectamente y perder el resultado óptimo.

Podemos comparar dos estados del problema según sus funciones costo

Aquella con mejor valor es mejor candidato a tener entre sus descendientes la solución óptima al problema

Recorrido del árbol

En backtracking

Utilizabamos Depth-First Search para recorrer el árbol

No había un criterio establecido para determinar que rama profundizar del árbol

En Branch & Bound

Se pueden aplicar diferentes estrategias de recorridos del árbol

Éstas aprovechan la función costo

Se privilegia la exploración de ramas con mejor valor en la función costo

Recorrido del árbol: Depth-first branch-and-bound

Partimos de un nodo

expandimos todos los nodos descendientes.

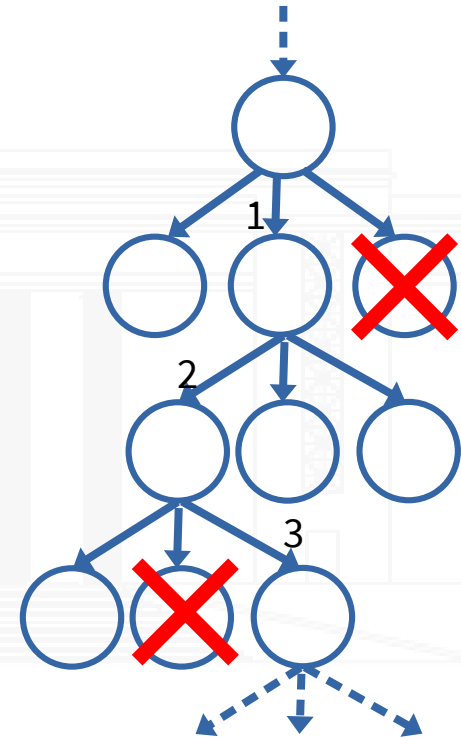
Se evalúan, podando utilizando la función límite

Entre los restantes se realiza la exploración seleccionando el más promisorio aún no explorado teniendo en cuenta la función costo

Al no quedar nodos por explorar se regresa al nodo padre.

El proceso finaliza

al no quedar nodos por explorar



Depth-first branch-and-bound – Pseudocódigo

Sea estadoInicial la raiz del arbol de estados
BranchAndBound(estadoInicial)

BranchAndBound (estadoActual):

Sea descendientes los nodos descendientes de estadoActual

Por cada posible estadoDescendiente de estadoActual

 Si estadoDescendiente supera la propiedad de corte

 Calcular fc funcion costo de estadoDescendiente

 Agregar estadoDescendiente a descendientes con fc

Mientras existan estados descendientes no explorados

 Sea estadoProximo el estado en descendientes aún no analizado de mayor fc

 Si el fc de estadoProximo es mayor a la mejor solución obtenida

 Si estadoProximo es un estado respuesta y es superior a la mejor

 mejor = estadoProximo

Backtrack(estadoProximo)

Recorrido del árbol – Best-first search

Se inicia con una cola de prioridad utilizando la función costo de forma descendente.

En la cola de prioridad se incluye inicialmente la raíz del árbol

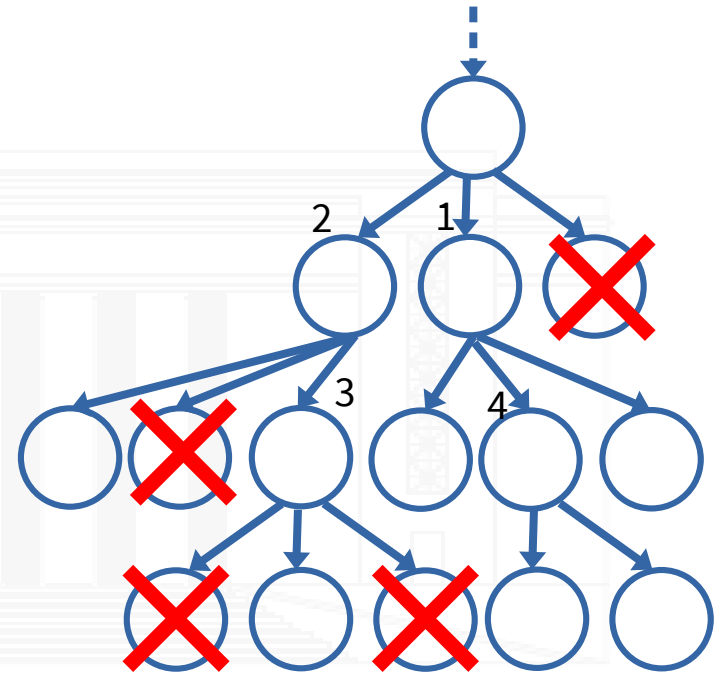
Se toma el nodo en la cola de mayor valor de costo que supere a la mejor solución encontrada

Determinamos si corresponde a un estado respuesta y supera el mejor encontrado actual

Expandimos todos los nodos descendientes.

Podamos aquellos que no superan la función límite

Calculamos su función costo y aquellos que superan al mejor encontrado actual lo insertamos en la cola



Best-first search – Pseudocódigo

Sea estadosDisponibles los estados en el árbol expandidos por analizar

Sea estadoInicial la raíz del árbol de estados

Calcular fc función costo de estadoInicial

Agregar estadoInicial con fc a estadosDisponibles

Mientras queden estados en estadosDisponibles

 Sea estadoActual el estado de mayor fc en estadosDisponibles

 Si estadoActual supera la propiedad de corte

 Si fc de estadoActual es mayor a la mejor solución obtenida

 Si estadoActual es un estado respuesta

 mejor = estadoProximo

 Por cada posible estadoSucesor de estadoActual

 Calcular fc función costo de estadoInicial

 Agregar estadoSucesor con fc a estadosDisponibles

Branch & Bound: Problema de la mochila

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Problema de la mochila

- **Contamos con:**
 - una mochila con una capacidad de K kilos
 - un subconjunto del conjunto E de “ n ” elementos
- **Cada elemento i tiene:**
 - un peso de k_i kilos
 - un valor de v_i .
- **Queremos seleccionar un subconjunto de E**
 - con el objetivo de maximizar la ganancia.
 - el peso total seleccionado no puede superar la capacidad de la mochila.



Valor por unidad del elemento

A cada elemento le calcularemos su valor por unidad u_i

Cociente entre el cociente entre su valor v_i y su peso k_i .

Un elemento con mayor valor por unidad que otro

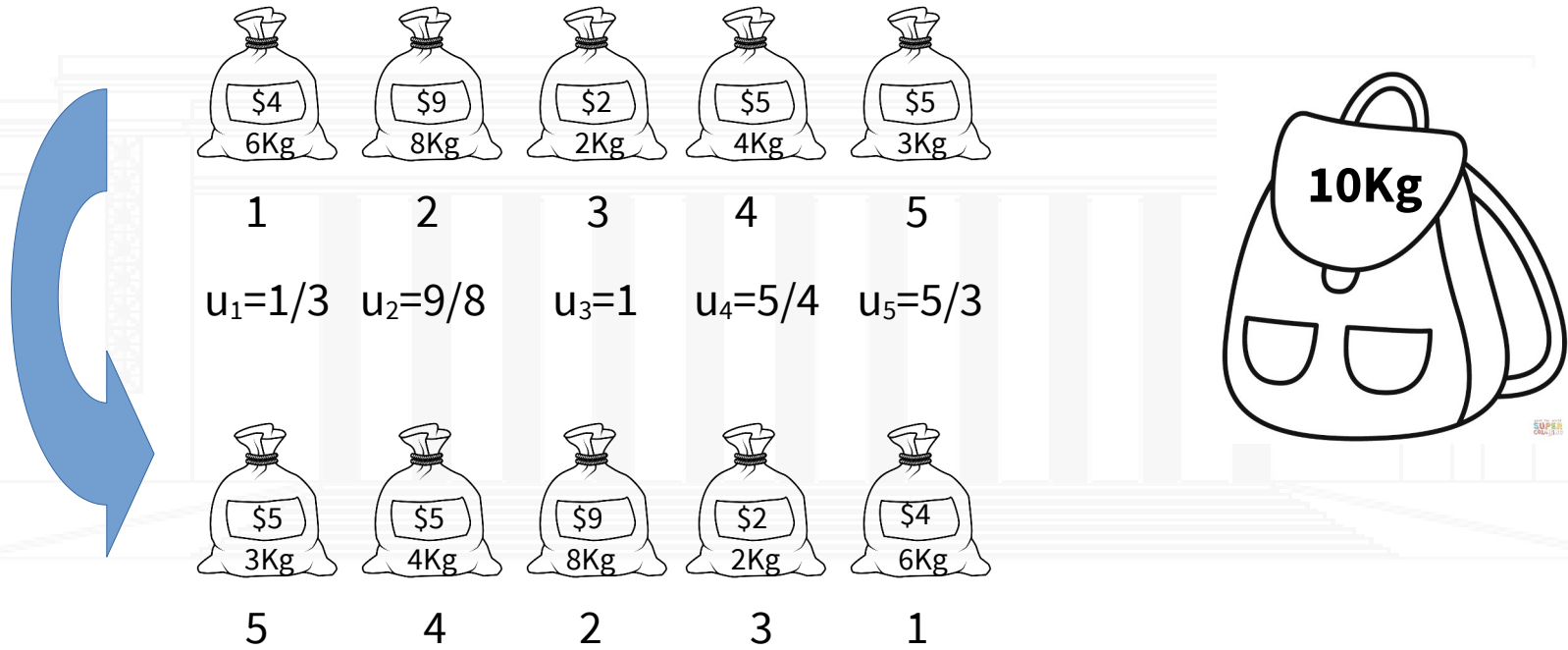
Produce una mayor ganancia ante misma cantidad de espacio ocupado

Ordenaremos de forma descendiente los elementos por valor unidad

asignaremos a cada elemento un identificador único (un valor entero) según este orden

Ejemplo: Valor por unidad

Supongamos la siguiente instancia



Árbol de estados del problema

Utilizaremos una estructura de árbol binario

cada nivel corresponde a determinar si el elemento i se incluye o no dentro de la mochila.

Utilizará el orden de elementos por valor por unidad

La raíz corresponde a la mochila vacía.

Cada nodo en el nivel “ i ” tiene

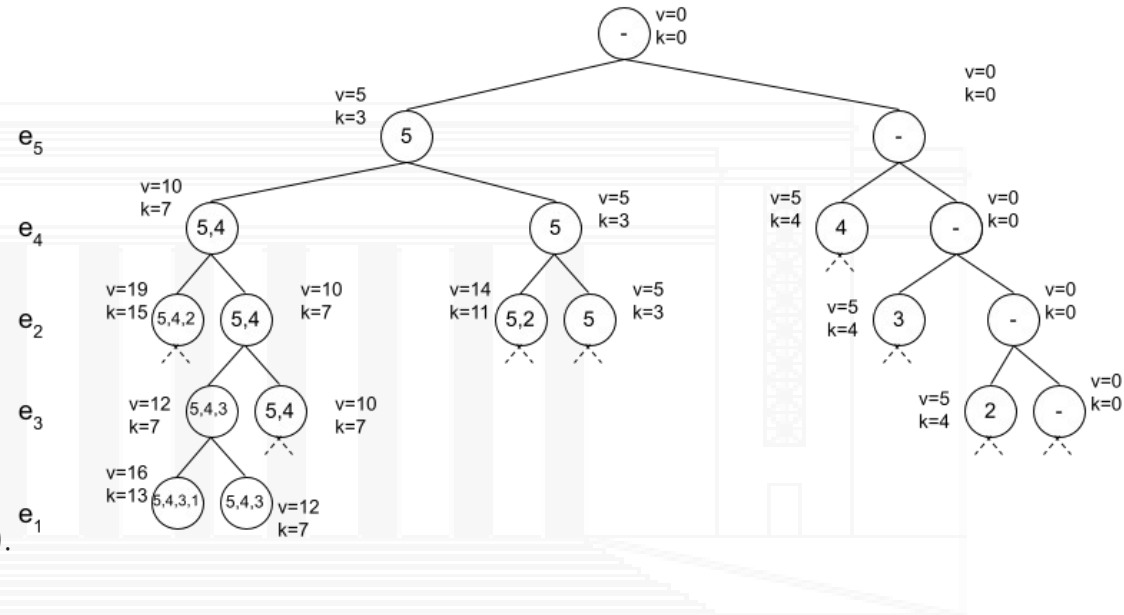
el peso cargado “ k ” en la mochila

el valor “ v ” de ganancia obtenido

dos descendientes (contiene o no el siguiente elemento).

Si tenemos n elementos en la mochila

la profundidad máxima corresponderá a n .



Función Costo “l”

Dado un nodo a una profundidad del árbol i

veremos cual es el elemento aun no evaluado de mayor valor por unidad (Dado el ordenamiento planteado, corresponde simplemente al elemento $i+1$)

Obtenemos cuál es el espacio disponible en la mochila $(K-k)$.

Supondremos que

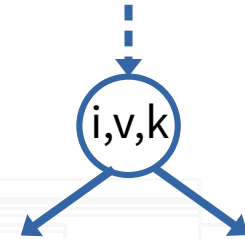
en el mejor de los casos todos ese espacio se llenará con el elemento $i+1$ (u otros posteriores con mismo valor por unidad).

Calcularemos la función costo de un nodo

Sumando el valor de la ganancia obtenida en el nodo actual “v” con la mayor ganancia hipotética posible

Corresponde al tope a la ganancia de cualquier estado del árbol que descienda del nodo.

Si este valor es inferior a la máxima ganancia encontrada, podemos realizar la poda.



$$l = v + (K-k) * u_{i+1}.$$



Recorrido del árbol de estados

Para recorrer el árbol utilizaremos depth-first branch-and-bound.

Comenzará por la raíz

Inicialmente la mejor solución es la mochila vacía ($k=0, v=0$)

Dado el nodo actual

Verificamos si es mejor que la mejor solución actual

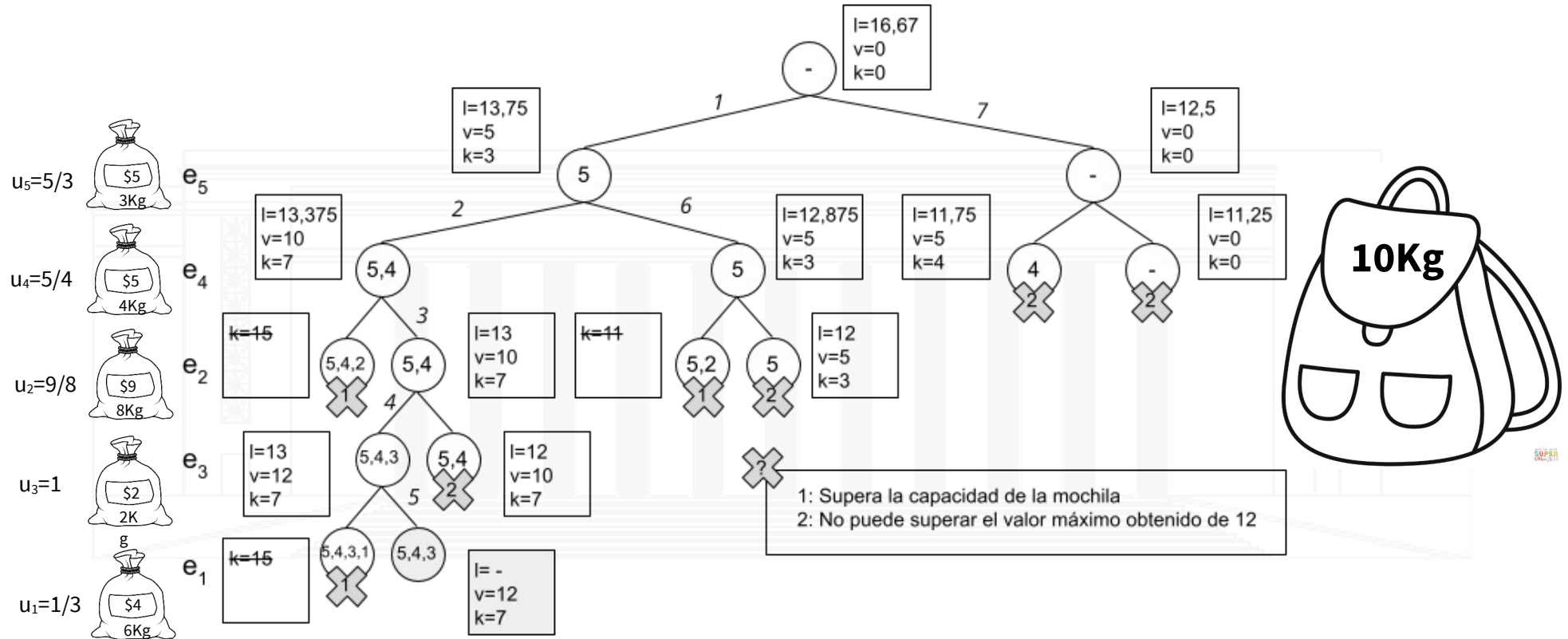
Expandimos cada uno de sus descendientes y calculamos su función costo

Dados sus descendientes no explorados que no superen la capacidad de la mochila y pueden tener una mejor solución a la actual

Seleccionamos al mejor de ellos (mayor función costo) y lo definimos como el nodo actual

Si no quedan descendientes por explorar se regresa al nodo padre

Exploración del árbol en el ejemplo



Branch & Bound – Pseudocódigo

Sea mochila la mochila inicialmente vacia
Sea K la capacidad de la mochila
Sea elementos el listado de n elementos ordenados segun su valor por unidad
Sea nroelemento la posicion en el listado de elemento a evaluar
Sea mejorMochila la combinacion de la mochila que da mayor ganancia encontrada
Sea mejorGanancia la combinacion de la mochila que da mayor ganancia encontrada

```
nroelemento = 1  
mejorResultado = {}  
mejorGanancia = 0  
Backtrack(mochila, nroelemento)
```

Branch & Bound – Pseudocódigo

Backtrack (mochila, nro):

Sea elemento el elemento en la posicion nro en elementos.

Sea mochilaAmpliada = mochila \cup elemento

Sea descendientes los nodos descendientes de estadoActual

Sea pesoActual el peso de los elementos en la mochila

Sea gananciaActual la suma de los valores de los elementos en mochilaAmpliada

Sea pesoAmpliado el peso de los elementos en la mochilaAmpliada

Sea gananciaAmpliada la suma de los valores de los elementos en mochilaAmpliada

Si $nro == n$

 Si $\text{pesoActual} \leq K$ y $\text{gananciaActual} > \text{mejorGanancia}$

 mejorMochila = mochila

 mejorGanancia = gananciaActual

 Si $\text{pesoAmpliado} \leq K$ y $\text{gananciaAmpliada} > \text{mejorGanancia}$

 mejorMochila = mochilaAmpliada

 mejorGanancia = gananciaAmpliada

sino

Branch & Bound – Pseudocódigo

Sea elementoSig el elemento en la posicion nro+1 en elementos.

Sea valorUnSig el valor por unidad de elementoSig

Sea $CotaActual = gananciaActual + (K - pesoActual) * valorUnSig$

Sea $CotaAmpliada = gananciaAmpliada + (K - pesoAmpliado) * valorUnSig$

Sea opciones los estados descendientes posibles.

Si $pesoActual \leq K$ y $CotaActual > mejorGanancia$

 Agregar mochila a descendientes con CotaActual

Si $pesoActual \leq K$ y $CotaActual > mejorGanancia$

 Agregar mochilaAmpliada a descendientes con CotaAmpliada

Mientras existan mochilas en descendientes no explorados

 Sea mochilaDesc en descendientes aún no analizada con mayor cota

 Si $cota > mejorGanancia$

 Backtrack(mochilaDesc, nro+1)

Complejidad Temporal

En el peor de los casos se analiza 2^n estados del problema.

En cada estado se realizan procedimientos $O(1)$ para los cálculos de los límites y costos

Actualizar la mejor solución encontrada es $O(n)$

En el peor de los casos se realiza ante cada agregado de un elemento

La complejidad temporal es la multiplicación de estos dos valores

Complejidad Espacial

Por la implementación recursiva

Por cada llamado en profundidad en el árbol incluimos el consumo de memoria adicional

La memoria utilizada

Es proporcional a la profundidad máxima de la recursión generada

La profundidad máxima es “n”.

En cada nivel de profundidad

Se agrega (o no) un elemento a la mochila

Se realizan cálculos que requieren $O(1)$ de almacenamiento

Branch and Bound: Problema del viajante de comercio

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

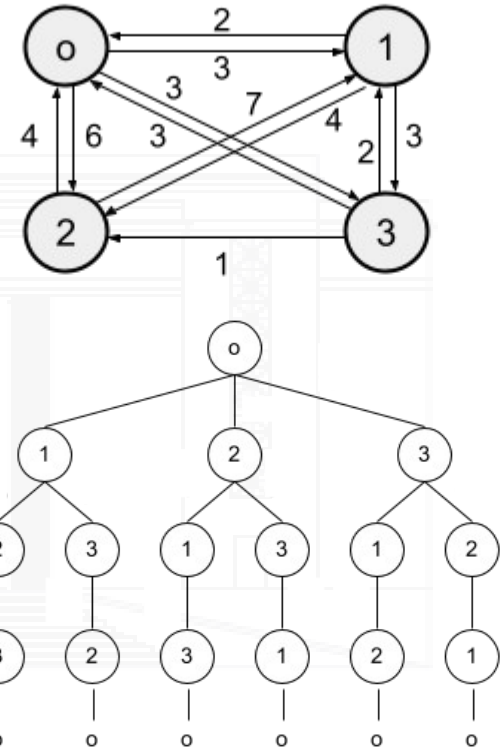
Problema del viajante de comercio

- Contamos con un conjunto de “n” ciudades a visitar.
 - Existen caminos que unen pares de ciudades.
- Cada camino
 - inicia en una ciudad “x” y finaliza en la ciudad “y”
 - tiene asociado un costo de tránsito de $w_{x,y}$.
- Partiendo desde una ciudad inicial y finalizando en la misma se quiere
 - construir un circuito que visite cada ciudad una y solo una vez minimizando el costo total.



Árbol de estados del problema

- La raíz representa el comienzo del viaje partiendo de la ciudad inicial
- Cada nodo representa la inclusión de una ciudad en el recorrido que no fue previamente visitada
 - Tiene como posibles estados descendientes las posibles elecciones de próximas ciudades (restringidas por las previamente visitadas).
 - Al elegir una opción se debe sumar el costo del traslado
- El camino desde la raíz hasta el nodo representa el recorrido realizado desde la ciudad inicial hasta el momento
 - El costo del recorrido es la suma de los costos de los trayectos entre las ciudades realizadas
- El árbol para n ciudades tiene un total de $1 + \sum_{i=1}^n \prod_{j=1}^i (n+1) - i$ estados del problema



Costo del Ciclo

- Dado una posible solución correspondiente a un ciclo $C=[o,x_1,x_2,x_{n-1},o]$, podemos calcular su costo como:

$$Costo(C) = w_{o,x_1} + \left(\sum_{j=1}^{n-1} w_{x_j,x_{j+1}} \right) + w_{x_{n-1},o}$$

- Podemos separar el circuito en dos partes $C_1=[o,x_1,...,x_i]$ y $C_2=[x_{i+1},...,x_{n-1},o]$,

$$Costo(C) = Costo(C_1) + Costo(C_2)$$

$$Costo(C) = \left[w_{o,x_1} + \left(\sum_{j=1}^{i-1} w_{x_j,x_{j+1}} \right) \right] + \left[\left(\sum_{j=i}^{n-1} w_{x_j,x_{j+1}} \right) + w_{x_{n-1},o} \right]$$

Existen varios Ciclos que comparte el orden de visitas de las primeras i ciudades

Para todas ellas el costo C_1 es el mismo

Función costo “l”

Dado un nodo a profundidad “i” del árbol de estados

Sabemos las ciudades visitadas y podemos calcular el costo acumulado C_1

Llamamos X al conjunto de todas las ciudades a recorrer

podemos calcular el subconjunto $Y \subseteq X$ como aquellas ciudades aún no visitadas al llegar a ese nodo.

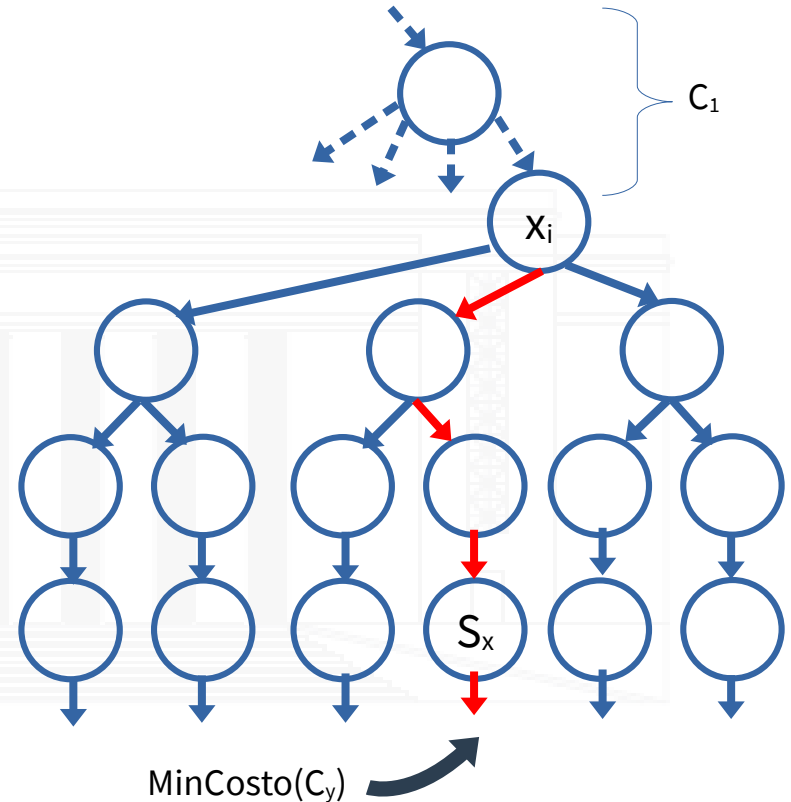
Existirán posiblemente varias alternativas para generar el camino C_2 .

En el caso extremo todas las permutaciones posibles de las ciudades en Y, cada una de estas con un costo diferente.

Nos interesa poder medir el menor de los costos posibles al que llamaremos $\text{MinCosto}(Y)$.

Si $\text{MejorActual} \leq l = \text{Costo}(C_1) + \text{MinCosto}(Y)$, sabemos que no hay un circuito mejor al existente que comience con el camino C_1 .

Buscaremos estimar $\text{MinCosto}(Y)$



Estimación de MinCosto(Y)

Supongamos que todas las ciudades en Y están comunicadas entre sí y además comunicadas con la última ciudad x_i de C_1

Podemos expresar los valores en una grilla donde la columna representa la ciudad de origen y la fila la de destino

Un camino C_2 representado en la grilla corresponde a

una celda por fila y columna

Las permutaciones posibles son los posibles caminos

La suma de los valores de las celdas seleccionadas corresponde al costo del camino.

	x_{i+1}	...	x_n	x_o
x_i	$w_{xi,xi+1}$...	$w_{xi,xn}$	
x_{i+1}		...	$w_{xi+1,xn}$	$w_{xi+1,xo}$
...
x_n	$w_{xn,xi+1}$...		$w_{xn,xo}$

No se puede ir de una ciudad a si misma

No llegar hasta el ultimo paso

Estimación de MinCosto(Y) - Continuación

Llamaremos MinCosto(C_y) a la estimación de MinCosto(Y)

Se propone tomar por cada fila de la matriz el menor peso disponible

Como precaución no seleccionar para x_i la ciudad de origen
(a menos que sea la única posibilidad).

$$\min \left\{ w_{x_i, z} / z \in Y \right\} + \sum_{y \in Y} \min \left\{ w_{y, z} / z \in Y \cup \{o\} \right\}$$

	x _{i+1}	...	x _n	x _o
x _i	w _{xi,xi+1}	...	w _{xi,xn}	
x _{i+1}		...	w _{xi+1,xn}	w _{xi+1,xo}
...
x _n	w _{xn,xi+1}	...		w _{xn,xo}

Las celdas seleccionadas podrían no corresponder a un camino válido

(Puede incluir 2 o más celdas de la misma columna).

Pero me aseguro que la sumatoria de los pesos van a ser igual o menor al costo del menor camino mínimo válido posible en C_y : MinCosto(C_y) ≤ MinCosto(Y)

Función costo “l” estimada

Con la estimación de $\text{MinCosto}(Y)$

La podemos reemplazar en la función costo “l”

Nos servirá para determinar qué nodo podar.

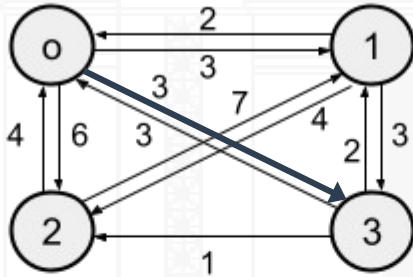
Si $\text{MejorActual} \leq \text{Costo}(C1) + \text{MinCosto}(Y)$, entonces $\text{MejorActual} \leq \text{Costo}(C1) + \text{MinCosto}(C_y)$ y por lo tanto no hay forma de – explorando esa rama del árbol – encontrar una mejor solución a la actual.

Nos servirá para determinar qué nodo explorar primero

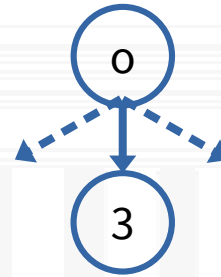
Si comparamos entre dos nodos, la que tiene mejor “l” posiblemente tenga un ciclo mejor entre sus descendientes.

Ejemplo

Supongamos la siguiente instancia del problema:



Debemos calcular la función costo “l” del nodo [0,3]:



$$C_1 = [0, 3]$$

$$Y = \{1, 2\}$$

$$\text{Costo}(C_1) = 3$$

	1	2	0
3	2	1	X
1	X	4	2
2	7	X	4

$$\text{Costo}(Y) = 7$$

$$l = 3 + 7 = 10$$

Recorrido del árbol de estados

Para recorrer el árbol utilizaremos Best-First.

Se incluye en la cola de prioridad la raíz del árbol con su valor de función de costo “l”

Se toma el camino parcial desde la ciudad “o” en la cola de mayor valor de costo que supere a la mejor solución encontrada

Determinamos si corresponde a un camino de longitud n en ese caso un podemos cerrar el ciclo

Sino, obtenemos las ciudades que aun no se incluyen y expandimos cada posible camino parcial.

Por cada uno de los caminos parciales verificamos si supera la función limite y se agregan en la cola de prioridad

Branch & Bound – Pseudocódigo

Sea $C[x,y]$ el costo de ir de la ciudad x a la y
Sea $A[y]$ la lista de ciudades adyacentes de la ciudad y .
Sea camino el recorrido realizada hasta el momento
Sea minimoCosto el costo del camino minimo encontrado
Sea minimoCamino el circuito de menor costo encontrado

camino={o}.
minimoCosto=infinito
minimoCamino={}

Definir $fc=0$ para costo de camino
Agregar camino con fc a caminosDisponibles

Branch & Bound – Pseudocódigo

```
Mientras queden caminos estos en caminosDisponibles
  Sea caminoActual el camino de mayor fc en caminosDisponibles
  Sea x la ultima ciudad visitada en caminoActual.
  Si longitud del camino es n
    Si la ciudad o se encuentra en A[x]
      Agregar al final de caminoActual la ciudad o
      Sea costoCamino la suma de los costos de caminoActual
      Si costoCamino < minimoCosto
        minimoCosto = costoCamino
        minimoCamino=camino
    Sino
      Por cada ciudad y en A[x] no visitada previamente excepto "o"
        Sea caminoAmpliado = caminoActual+{y}
        Determinar Y ciudades aun no visitadas en caminoAmpliado
        Sea costoCamino la suma de los costos de caminoAmpliado
        Calcular fc=costoCamino +MinCosto(x,Y)
        Si fc < minimoCosto
          Agregar caminoAmpliado con fc a caminosDisponibles
```

Complejidad Temporal

Debemos explorar

en el peor de los casos explorar todos los estados del árbol de estados en $O(n!)$.

Calcular para cada nodo

su $\text{MinCosto}(Y)$ con complejidad $O(n^2)$

Calcular en $O(n)$ todos sus descendientes posibles

Realizar operaciones $O(1)$

Agregar y eliminar un nodo a la cola de prioridad

$\log(A)$ donde A es la cantidad de nodos agregados. $A=n!$

Unificando y simplificando tendremos $O(n! \log n!)$

Complejidad Espacial

Cada nodo

requiere $O(n)$ para generar el camino parcial

$O(1)$ por otras operaciones

En de la cola de prioridad

Se almacenan los nodos con el parcial que requiere $O(n)$ y su valor de prioridad " l "

En el peor de los casos

tendremos $O(n!)$ nodos en la cola de prioridad

Tendremos un consumo elevado de memoria de

$O(n \cdot n!)$