



# Teoría de Algoritmos

Apunte

# Búsqueda Exhaustiva

## Presentación

- En el peor de los casos se prueban todas las soluciones posibles

Metodologías que usan fuerza bruta

- Generar y Probar
- Backtracking
- Branch and bound

# Búsqueda Exhaustiva

## Generar y Probar

- Función generativa : recorre todo el espacio de soluciones posibles , con restricciones explícitas.
  - una estructura para representar una posible solución
  - una manera de obtener la próxima solución
  - una manera de determinar la sencuencia de próximas soluciones
- Función de prueba: verifica cada solución cumpla con los requerimientos ( restricciones implícitas)
  - la lógica para revisar la solución
  - la lógica para comparar una solución contra otra

# Búsqueda Exhaustiva

## Espacios de soluciones

- N-tuplas
- Permutaciones
- Combinaciones
- Particiones de conjunto

## Explosión combinatoria

- A medida que el tamaño de la instancia crece linealmente el espacio de solución crece exponencialmente (o peor)
- En Generar y probar para instancias relativamente grandes el tiempo de ejecución se vuelve innombrable

# Búsqueda Exhaustiva

## N-tuplas

Espacio de soluciones

- Subconjunto de  $n$  elementos
- Los elementos son únicos e indivisibles
- Un elemento puede estar seleccionado o no
- No importa el orden de los elementos

## Función generativa

- Generación de  $n$  tuplas
- Correspondrá a las restricciones del problema
- Las tuplas deben ser generadas en orden lexicográfico  
 $(001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow \text{etc})$

# Búsqueda Exhaustiva

```
def incrementar(c):
    n = len(c)

    pos = n - 1

    # Desde la derecha, cambiamos todos los 1 por 0 hasta encontrar un 0
    while c[pos] == 1 and pos >= 0:
        c[pos] = 0
        pos -= 1

    # Si no encontramos un 0, terminamos
    if pos == -1:
        return 'fin'

    # Si encontramos un 0, lo cambiamos por 1
    c[pos] = 1
    return c
```

# Búsqueda Exhaustiva

## Permutaciones

Espacio de soluciones

- Vamos a determinar un orden de  $n$  elementos
- Los elementos son únicos e indivisibles
- Todos los elementos deben ser seleccionados

Función generativa

- Generación de las permutaciones de los elementos
- Correspondrá a las restricciones del problema
- Orden lexicográfico

$$1 \ 2 \ 3 \ 4 \rightarrow 1 \ 2 \ 4 \ 3 \rightarrow 1 \ 3 \ 2 \ 4 \rightarrow 1 \ 3 \ 4 \ 2$$

# Búsqueda Exhaustiva

```
def permutar(c):
    n = len(c) - 1

    # En indice1 guardamos el índice del primer elemento que es menor que el siguiente
    indice1 = n - 1
    while c[indice1] >= c[indice1 + 1]:
        indice1 -= 1

    if indice1 == -1:
        return 'fin'

    # En indice2 guardamos el índice del menor elemento mayor que c[indice1]
    indice2 = n
    while c[indice1] >= c[indice2]:
        indice2 -= 1

    # Intercambiamos c[indice1] con c[indice2]
    aux_valor1 = c[indice1]
    aux_valor2 = c[indice2]

    c[indice1] = aux_valor2
    c[indice2] = aux_valor1

    # Ordenamos de menor a mayor todos los elementos salvo el que quedo en la posición indice1
    k = indice1 + 1
    l = n
    while k < l:
        aux_valorK = c[k]
        aux_valorL = c[l]

        c[k] = aux_valorL
        c[l] = aux_valorK

        k += 1
        l -= 1

    return c
```

# Búsqueda Exhaustiva

## Combinaciones

Espacio de soluciones

- Selección de un subconjunto de  $m$  elementos entre  $n$
- Los elementos son únicos e indivisibles
- No importa el orden

Función generativa

- Generación de las combinaciones de los elementos
- Correspondrá a las restricciones del problema
- Orden lexicográfico + 2 sentinelas ( $0, n+1$ )  
sentinelas

0 8 | 4 3 2 1

0 8 | 5 3 2 1

0 8 | 6 3 2 1

↑ ↑  
0 n+1

• Con  $n=7$  y  
 $m=4$

# Búsqueda Exhaustiva

Ejemplo: Problema del clique

- Dado un grafo no dirigido  $G = (V, E)$ 
  - $V$ : conjunto de vértices
  - $E$ : aristas
- Queremos obtener un clique de tamaño  $m$  dentro de él
  - los vértices deben ser adyacentes
- Cantidad de subconjuntos  $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

```
def incrementar(c, m):
    j = 0

    # Si los contiguos son consecutivos, incrementamos j
    while c[j]+1 == c[j+1]:
        c[j] = j + 1
        j += 1

    # Si j+1 > m, terminamos
    if j+1 > m:
        return 'fin'

    # Si no, incrementamos c[j]
    c[j] += 1
    return c
```

En este caso trabajamos con el vector invertido

1 2 3 4 | 8 0

# Búsqueda Exhaustiva

## Espacio de estados finitos

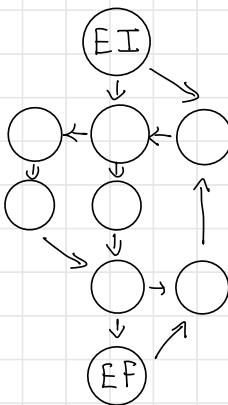
- Estado inicial



- Estados intermedios



- Estado final



## Métodos de búsqueda

Forma exhaustiva:

- Búsqueda por anchura
- Búsqueda en profundidad
- Best first
- Generar y probar
- Programación dinámica

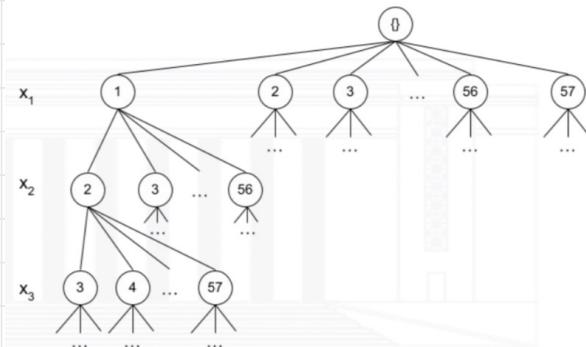
Forma heurística:

- Local
- Voraz

# Búsqueda Exhaustiva

## Backtracking

- Es un problema combinatorio que se puede representar en un árbol combinatorio



## Clasificación de estados

- Estados del problema: todos los nodos
- Estados solución: cumplen las restricciones explícitas
- Estados respuesta: cumplen las restricciones implícitas

# Búsqueda Exhaustiva

## Propiedad de corte

- Aplicamos una función límite para evaluar la propiedad de corte

## Recorrido

- Partimos de la raíz
- Por cada estado de problema descendiente se generan sus descendientes
- Usamos Depth-First-Search
- Se retrocede cuando no quedan caminos por recorrer en la rama actual

```
def backtrack(estado_actual):  
    if es_solucion(estado_actual):  
        return estado_actual  
  
    if supera_propiedad_corte(estado_actual):  
        for posible_sucesor in sucesores(estado_actual):  
            resultado = backtrack(posible_sucesor)  
  
            if es_solucion(resultado):  
                return resultado  
  
    return None
```

# Búsqueda Exhaustiva

## Backtracking : Problema de la mochila

- Raíz : mochila vacía = lista vacía
- Cada nodo agrega un elemento a la lista + el peso

Función límite : revisa que no se supere el peso máximo

Complejidad temporal: en el peor de los casos tengo que recorrer todos los nodos del arbol ( $\mathcal{O}(2^n)$ ) y en cada nodo el peor caso es  $\mathcal{O}(n)$

$$\mathcal{O}(n 2^n)$$

Complejidad espacial : la profundidad máxima del arbol es  $\mathcal{O}(n)$  y en cada nodo tengo  $\mathcal{O}(1)$

$$\mathcal{O}(n)$$

# Búsqueda Exhaustiva

## Backtracking: Problema del viajante

- Tengo que visitar  $n$  ciudades
  - $X \rightarrow T = W$
  - Raíz = ciudad inicial
  - Cada nodo tiene el costo total
- (OJO: el último nodo también tiene que ser esta ciudad)

Función límite: verifica que existe el camino

Complejidad temporal: se recorren los nodos con una complejidad  $O\left(\sum_{i=1}^n \prod_{f=1}^i (n+1) - i\right)$  y en cada nodo se realiza  $O(n)$

$$O\left(n \sum_{i=1}^n \prod_{f=1}^i (n+1) - i\right)$$

Complejidad espacial:  $O(n)$

# Búsqueda Exhaustiva

Backtracking: Coloreo de grafos

# Búsqueda Exhaustiva

## Branch and Bound

- Variante de Backtracking
- Agrega una función costo que realiza una poda adicional y determina un orden de inspección
- Guardamos la solución óptima
- Descartamos soluciones que sean peores que la ya encontrada

### Función costo

- Dado un estado del problema determina el posible valor de un posible estado respuesta descendiente del mismo
- Realiza una estimación del mejor valor descendiente posible
- Puede sobreestimar, nunca infravalorar

# Búsqueda Exhaustivo

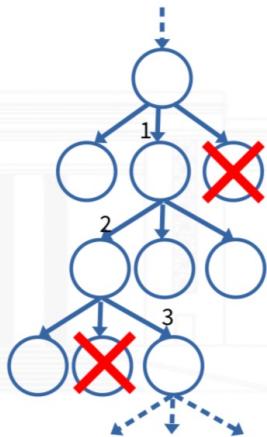
Recorrido: Depth - first

1 - Expandimos todos los nodos descendientes

2 - Podemos usando la función límite

3 - Elegimos el mejor de los restantes usando la función costo

4 - Finaliza cuando no quedan nodos por explorar

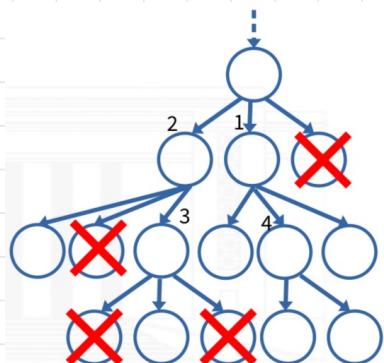


Recorrido: Best - first

- Arrancamos con una cola de prioridad que está ordenada por el costo

- Primero solo está la raíz

- Arranca el loop



1 - Tomamos el nodo en la cola de mejor valor costo que supera la solución encontrada

2 - Expandimos los descendientes

3 - Podemos con la función límite

4 - Calculamos el costo e insertamos en la cola a los que superen el mejor actual

# Búsqueda Exhaustiva

## Branch and Bound: Problema de la mochila

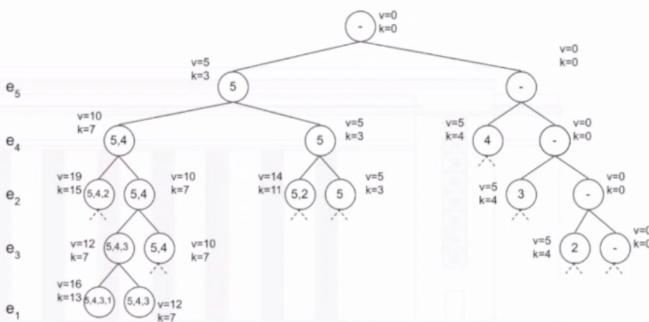
- Cada elemento tiene : peso  $k_i$  y valor  $v_i$
- Calculamos un costo  $u_i = v_i/k_i$
- Ordenamos según costo
- Cada nodo determina si el elemento  $i$  se incluye
- También cada nodo tiene el peso total de la mochila y el valor total de la mochila

Supongamos la siguiente instancia

1	2	3	4	5
$u_1=1/3$	$u_2=9/8$	$u_3=1$	$u_4=5/4$	$u_5=5/3$

5	4	2	3	1



Función costo: el valor actual + valor del próximo elemento de mayor ganancia

- Poda 1 : supera el peso máximo de la mochila
- Poda 2 : no supera el máximo valor obtenido

# Búsqueda Exhaustiva

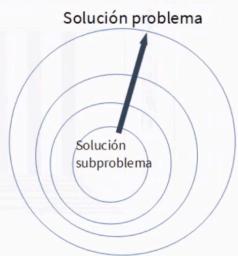
Complejidad temporal: igual a caso backtracking  $\mathcal{O}(n2^n)$

Complejidad espacial: igual a caso backtracking  $\mathcal{O}(n)$

# Greedy

## Presentación

- Resolver problemas de minimización y maximización
- Divide el problema en subproblemas
- Cada subproblema se resuelve de manera iterativa
- Cada subproblema habilita nuevas subproblemas
- Algunos algoritmos greedy no resultan en una solución óptima
- No todos los problemas pueden resolverse, deben tener ciertas propiedades (elección greedy y subestructuras óptimas)



## Elección greedy

- Seleccionar una solución óptima local que nos acerque a la global
- La solución de un subproblema analiza el estado actual del problema
- Un subproblema está condicionado por los anteriores y a su vez condiciona a los siguientes

# Greedy

## Subestructura óptima

- Un problema contiene una subestructura óptima si la solución óptima global del mismo contiene en su interior las soluciones óptimas de subproblemas
- La elección greedy resuelve de manera iterativa hasta llegar a la solución óptima global

## Es óptima?

- Es complicado demostrar que el algoritmo greedy resuelve optimamente el problema
- El algoritmo greedy tiene que ser óptimo en todas las instancias del problema

# Greedy

Pasos para la construcción de un algoritmo greedy

- 1 - Determinar la subestructura óptima del problema
- 2 - Construir una solución recursiva
- 3 - Mostrar que al realizar la elección greedy nos quedaremos con 1 subproblema
- 4 - Mostrar que la elección greedy se puede realizar siempre, sin importar la instancia
- 5 - Construir el algoritmo recursivo que soluciona el problema
- 6 - Convertirlo a iterativo

# Greedy

## Mochila Fraccionaria

- Capacidad  $W$
- $n$  elementos fraccionables de valor  $v_i$  y peso  $w_i$
- Queremos seleccionar un subconjunto de elementos o fracciones para maximizar el valor almacenado

Solución greedy: priorizar los elementos más valiosos por unidad

- Complejidad: ordenar los elementos según su valor  $O(n \log n)$  y iterar  $O(n)$ . Nos da  $O(n \log n)$
- Es óptima?

## Análisis de optimidad

- Si las soluciones greedy que encontramos son  $G = \{g_1, g_2, \dots, g_n\}$
- Y las soluciones óptimas son  $O = \{o_1, o_2, \dots, o_n\}$
- Si encontramos un  $o_i > g_i$  ya sabemos que NO ES OPTIMA! porque NO SELECCIONAMOS un elemento disponible. Esto es ABSURDO!  
Se demuestra, es óptimo!

# Greedy

## Cambio mínimo

- Tengo un conjunto de monedas de distintas denominaciones
- Un importe  $X$  de cambio a dar
- Quiero entregar la menor cantidad posible de monedas

Solución greedy: Busco la mayor moneda posible hasta completar el vuelto

- Complejidad: recorro todas las monedas  $O(n)$

- Es óptima?: Busco un contraejemplo

Tengo estas monedas  $\$ = (1, 2, 4, 5, 10)$

Si tengo que dar cambio de 8 uso 5, 2 y 1

Pero podría dar 2 de 4. No ES OPTIMO!

# Greedy

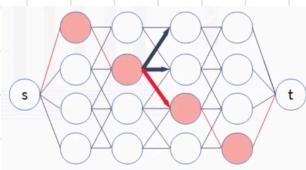
## Seam Carving + Camino mínimo

### Seam Carving

- Para recortar y ajustar imágenes
- Busca los pixeles poco importantes, calcula su importancia
- Ejemplo: si hay un pixel rodeado de similares
- Se busca una veta (horizontal o vertical) cuya suma de importancia sea la menor posible

### Camino mínimo

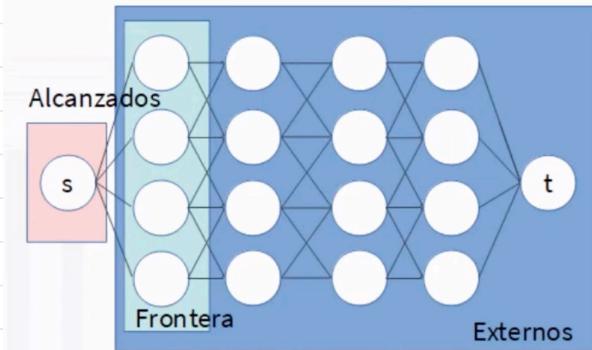
- Tratamos la imagen como una grilla de pixeles
- Cada pixel es un nodo de un árbol
- Los ejes son los posibles caminos de la veta
- Calculamos el camino mínimo entre  $s$  y  $t$



# Greedy

## Dijkstra (camino mínimo)

- Es greedy
- Para grafo dirigido y ponderado (costos positivos)
- "s" inicio y "t" fin
- Busca el camino mínimo
- Es iterativo
- Divide los nodos en : alcanzados, externos y frontera



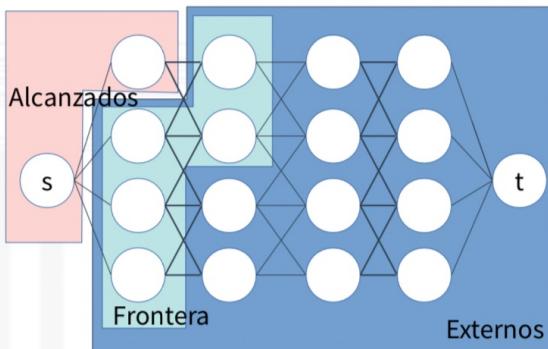
1

# Greedy

Paso a paso:

- 1 - Se obtiene el nodo  $x$  de la "frontera" cuyo costo de llegada desde "s" sea el menor posible  
*elección greedy*
- 2 - Se agrega el nodo  $x$  a "alcanzados" con su costo
- 3 - Se actualizan los nodos "frontera" con sus costos
- 4 - Se actualizan los costos de la "frontera" con conexión al nodo  $x$  si el nuevo costo es menor

Repite hasta quedarne sin nodos frontera o llego al nodo  $t$



# Greedy

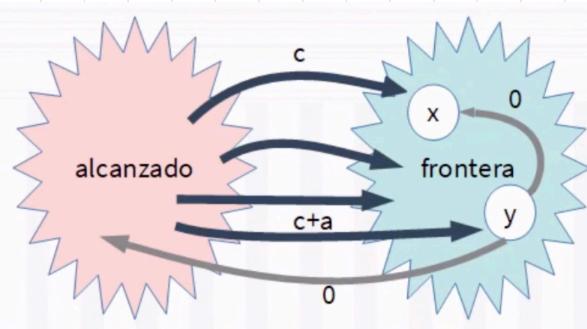
## Complejidad

- Cada vez que obtengo el nodo con menor costo  $\rightarrow O(\log n)$
- Insertar en la frontera  $\rightarrow O(\log n)$
- En la frontera se realizan  $m$  actualizaciones  $\rightarrow O(\log n)$   
 $\uparrow$   
cantidad de ejes

$$O(n + m \log n)$$

## Elección greedy

Aunque de ya a x el costo sea cero, no hay manera de que el costo hacia y sea menor que hacia x porque siempre elijo el mínimo.



# Greedy

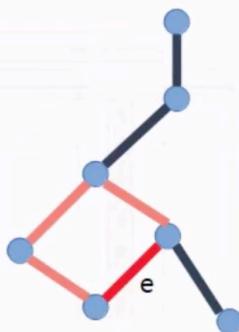
## Árbol recubridor mínimo (MST)

- Quiero seleccionar un subconjunto  $T$  de un grafo conexo y ponderado  $G = (V, E)$
- El nuevo grafo  $G'$  tiene que ser conexo y su costo total sea el mínimo
- Si  $G'$  tiene ciclos y extraemos un eje  $e$ ,  $G'$  sigue siendo conexo y tendrá un costo menor

Nos quedaría: un árbol, que es un grafo no dirigido conexo y sin ciclos

Ejemplo 

Si quitamos  $e$   
sigue siendo un árbol  
conexo



# Greedy

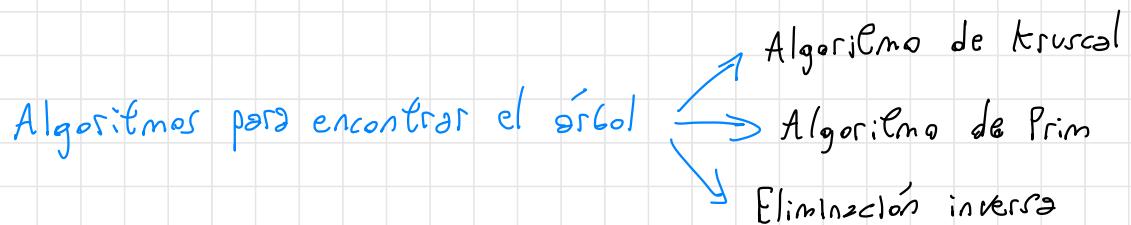
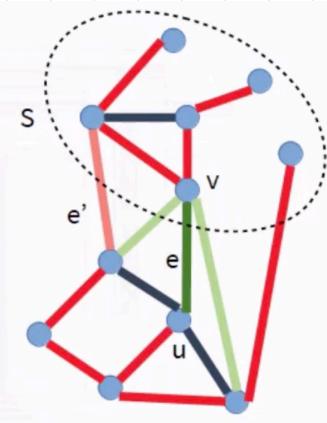
- Pueden existir varios árboles recubridores, pero solo algunos serán mínimos

## Propiedad de cortijo

Si tenemos un árbol recubridor  $T$  (en rojo) que no contiene a  $e$  pero sí a  $e'$ .

Podemos intercambiar  $e'$  por  $e$  siempre que se mantenga conexo.

Si  $W_e < W_{e'}$  entonces el costo de  $T'$  será menor al de  $T$ .

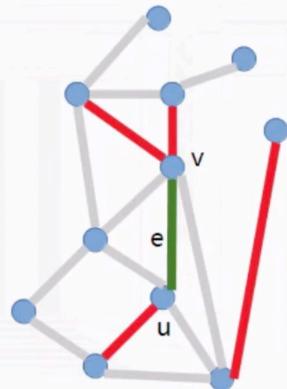


Si tengo ejes con costos iguales: voy a tener varios árboles recubridores mínimos

# Greedy

## Algoritmo de Kruskal

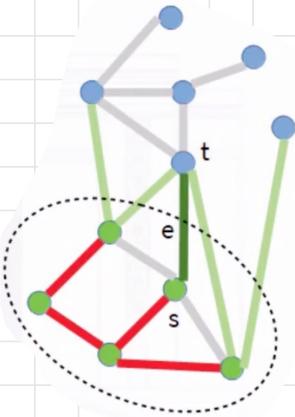
- Iterativamente recorro mi árbol en orden creciente de costo
- En cada iteración analizo un eje  $e = (u, v)$
- Si al agregar el eje se genera un ciclo lo desecho



# Greedy

## Algoritmo de Prim

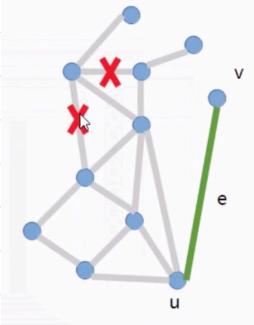
- Inicialmente el algoritmo selecciona un eje y determina que este pertenece al conjunto  $S$
- En cada iteración selecciona un eje  $e = (s, t)$  donde  $t$  todavía no pertenece a  $S$  y sea el de menor costo



# Greedy

## Algoritmo de eliminación inversa

- Iniciamos con el grafo completo
- En cada iteración busco el eje de mayor costo
- Elimino el eje si el resultado sigue siendo conexo



### Propiedad de ciclo

Si existe un ciclo  $C$  en el grafo  $G$ , el eje de mayor costo en  $C$  no pertenece al árbol recubridor mínimo.

# Greedy

## Interval Scheduling

- P un subconjunto de pedidos  $\{p_1, p_2, \dots, p_n\}$
- Cada  $p_i$  tiene un  $t_i^s$  y  $t_i^f$
- Tengo que seleccionar la mayor cantidad de tareas que no se solapen
- Iterativamente seleccionamos los pedidos usando una elección greedy

### Elección greedy

- ~~Aquel que comienza antes~~
- ~~Aquel que dura menor tiempo~~
- ~~Aquel que tiene menos incompatibilidades~~
- Aquel que termina antes  $\rightarrow$  parece imposible encontrar un contraejemplo  
 $\downarrow$   
es óptimo!

# Greedy

## Implementación

- Ordenamos por tiempo de finalización
- Iteramos ignorando los incompatibles (los que tienen una fecha de inicio anteriores a la fecha de finalización actual)

## Complejidad

- Ordenar  $\rightarrow O(n \log n)$
- Iteración  $\rightarrow O(n)$
- Si todos los pedidos son compatibles  $\rightarrow O(n)$  (especial)

# División y Conquista

## Presentación

- Se divide un problema en subproblemas de menor tamaño
- Los subproblemas se resuelven de forma recursiva
- Se aplica donde la solución por fuerza bruta tiene complejidad polinomial

## Relación de recurrencia

### Ecuación que define una secuencia recursiva

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

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

## Pasos:

- 1 - Dividir el problema en "q" subproblemas de tamaño reducido
- 2 - Resolver cada subproblema mediante recursión
- 3 - Combinar el resultado

# División y Conquista

## Merge Sort

MERGESORT(A)

Si size A == 2 → comparar y devolver ordenado

A1 = (size A)/2 primeros elementos de A

A2 = (size A)/2 últimos elementos de A

Retornar MERGE (MERGESORT(A1), MERGESORT(A2))

Ejemplo:  $A = [2, 1, 4, 3]$

MERGESORT (A)      A1 = [2, 1]      A2 = [4, 3]

↓

MERGE SORT (A1) → A1 = [1, 2]

↓

MERGESORT (A2) → A2 = [3, 4]

↓

MERGE (A1, A2) → A = [1, 2, 3, 4]

# División y Conquista

## Análisis de la complejidad

- $T(n)$  es el peor caso de tiempo de ejecución para "n" elementos
- Dividir el problema en 2 subproblemas es  $O(1)$  en este caso
- Unir los subproblemas (merge) es  $O(N)$

mañ el costo de dividir y unir

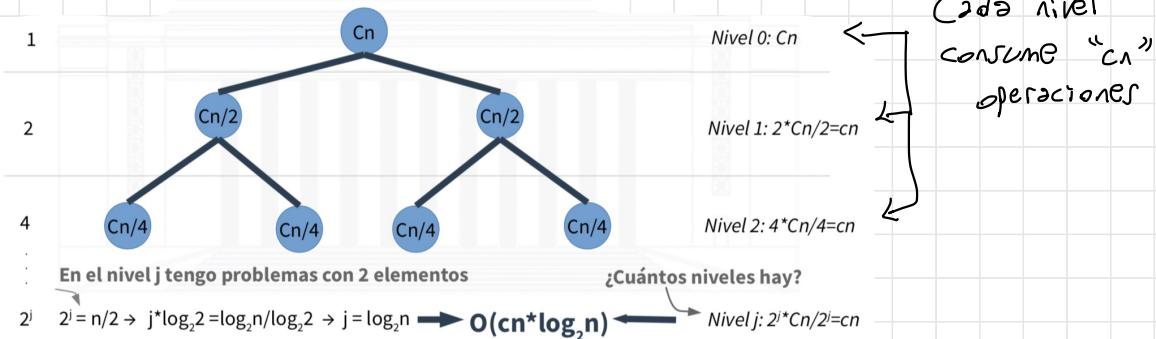
$$T(n) \leq 2 T\left(\frac{n}{2}\right) + \underbrace{\text{DIV} + \text{UNIR}}$$

↓  
porque se dividen en 2 mitades

el mismo problema en la mitad de elementos

$$T(n) \leq 2 T\left(\frac{n}{2}\right) + cn \quad n > 2$$
$$T(2) \leq d \quad n = 2$$

## Desarrollando



# División y Conquista

Validando el resultado

Probar en la recurrencia la validez del resultado

$$T(2) \leq c$$

$$T(n) \leq 2*T(n/2) + cn$$

$$T(n=2) = cn \log_2 n = 2c$$

$$T(2) \leq d < 2c$$

$$\begin{aligned} T(n) &\leq 2c(n/2) \log_2(n/2) + cn \\ &= cn[(\log_2 n) - 1] + cn \\ &= (cn \log_2 n) - cn + cn \\ &= cn \log_2 n. \end{aligned}$$

"Adivinar" y verificar

Podemos calcular  $T(n)$  probando...

Ej:  $T(n) = O(n)$ ,  $O(n^2)$ ,  $O(n \log n)$

$$T(n) \leq 2*T(n/2) + cn \quad n > 2$$

$$O(n) : kn \leq 2 * kn/2 + cn \leq (k+c)n$$

$$O(n^2) : kn^2 \leq 2 * kn^2/2 + cn \leq kn^2 + cn$$

$$O(n \log n) : kn \log_2 n \leq 2k(n/2) \log_2 (n/2) + cn =$$

$$= kn * (\log_2 n - 1) + cn = kn \log_2 n - kn + cn$$

$$kn \log_2 n \leq kn \log_2 n + (c-k)n$$

Mejor!

$k_n + c_n$

sii  $c=0$  X

para todo  $c \geq 0$  ✓

sii  $k \leq c$  ✓

# División y Conquista

Que pasa si se divide en q subproblemas?

(en vez de solo 2)

Podemos explorar la recurrencia como:

$r = q/2 > 1 \rightarrow$  podemos usar la suma geométrica

$$\sum_{i=0}^x r^i = \frac{1-r^{(x+1)}}{1-r}$$

$$T(n) \leq \sum_{j=0}^{\log_2 n - 1} \left(\frac{q}{2}\right)^j cn$$

$n^{\log_2 r}$

$$T(n) \leq cn \frac{(1-r^{\log_2 n})}{1-r} \leq cn \frac{r^{\log_2 n}}{r-1}$$

$$T(n) \leq \frac{c}{\frac{q}{2}-1} nn^{\log_2 \frac{q}{2}} = \frac{c}{\frac{q}{2}-1} n^{1+\log_2 q - 1} \longrightarrow O(n^{\log_2 q})$$

# División y Conquista

## Teorema Maestro

Séan:

- $a \geq 1$  y  $b > 1$
  - $f(n)$
  - $T(n) = aT(n/b) + f(n)$  con  $T(0) = c\epsilon$
- . Si se cumple alguno de los siguientes casos sobre las complejidades temporales

¡¡ Preguntas !!  
como funcionan  
las acotaciones !!

$$1) \text{ Si } f(n) = O(n^{\log_b a - e}), e > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$$

$$2) \text{ Si } f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} * \log n)$$

$$3) \text{ Si } f(n) = \Omega(n^{\log_b a + e}), e > 0 \Rightarrow T(n) = \Theta(f(n))$$

Y  $af(n/b) \leq cf(n)$ ,  $c < 1$  y  $n >>$

$\mathcal{O}$  : acotado superiormente

$\mathcal{\Omega}$  : acotado superior e inferiormente

$\mathcal{\Omega}$  : acotado inferiormente

# División y Conquista

Otra forma de verlo

$$\mathcal{T}(n) = A\mathcal{T}\left(\frac{n}{B}\right) + \mathcal{O}(n^C)$$

$$\begin{array}{ll} < & \rightarrow \mathcal{T}(n) = \mathcal{O}(n^C) \\ \text{Si } \log_B(A) = C & \rightarrow \mathcal{T}(n) = \mathcal{O}(n^C \log_B n) = \mathcal{O}(n^C \log n) \\ > & \rightarrow \mathcal{T}(n) = \mathcal{O}(n^{\log_B A}) \end{array}$$

Solo para casos donde  $f(n)$  tiene la forma  $\mathcal{O}(n^c)$

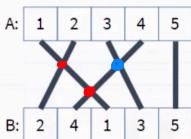
# División y Conquista

## Contando inversiones

- Conjunto de "n" elementos
- Dos listas ordenadas de los "n" elementos

Queremos tener una medida de semejanza / diferencia entre las dos listas.

Ejemplo:



Podemos ver que

- 1 está "invertido" con 2 elementos: 2 y 4 → (2,1) (4,1)
- 4 está "invertido" con 1 elemento: 3 → (4,3)

En total hay 3 inversiones

Para poder medir que tan distintas son las listas vamos a contar inversiones.

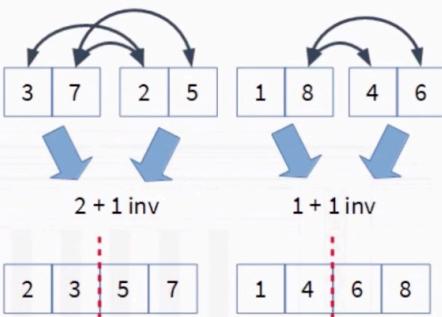
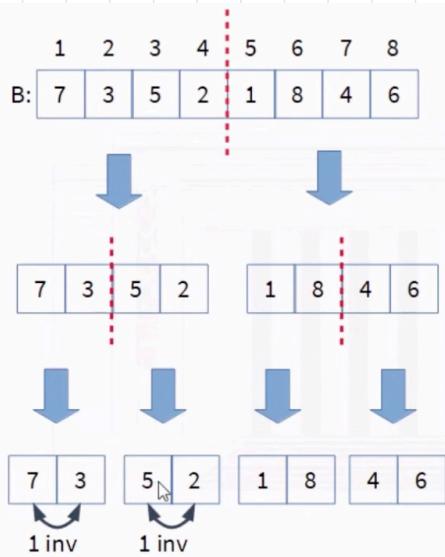
Inversión: si  $b_i > b_j$  con  $i < j$

Complejidad:  $O(n^2)$

# División y Conquista

## Resolución por división y conquista

- Partir por la mitad la lista & de forma recursiva hasta tener listas con solo dos elementos



- Aquí tenemos 2 inversiones
    - Ahora hago el merge y cuento las inversiones realizadas
- 3 7      2 5
- Primero va el 2, como 2 mi izquierda tengo 2 números (3 7) cuento 2 inversiones
  - El 5 va luego del 7 y cuenta como 1 inv

- Luego vuelvo a hacer otro merge contando las inversiones y me quedan

$$1 + 1 + 3 + 2 + 4 + 2 + 1$$

$$= 14$$

# División y Conquista

## Complejidad

- Cada problema se divide en 2 subproblemas con  $n/2$  elementos
- La unión de los resultados recorre  $n$  elementos  $\rightarrow O(n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad a = 2, b = 2$$
$$f(n) = O(n)$$

Usando el teorema maestro:

Comienzo por el paso 2 :

$$f(n) = O(n) = \Theta(n^{\log_2 2}) = \Theta(n) \quad \checkmark$$

$$\boxed{O(n \log n)}$$

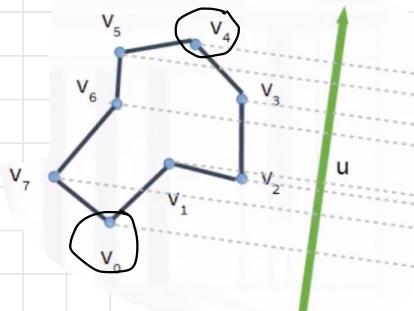
# División y Conquista

## Punto extremo en polígono convexo

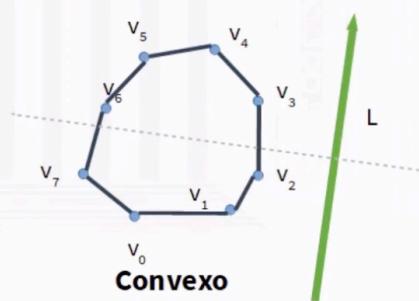
. Sea un polígono de  $n$  vértices  $V = (v_0, v_1, \dots, v_n)$

. Con  $v_n = v_0$ , en orden contra reloj 

. Queremos determinar el vértice extremo a un eje  $u$   
(mayor o menor)



Este no es  
convexo igual...



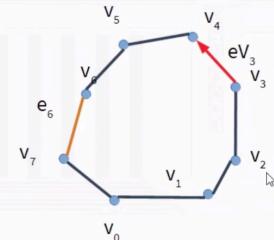
Este si!

# División y Conquista

## Nomenclatura

- $C_i$  segmento que va de  $V_i \rightarrow V_{i+1}$  para  $i = 0 \dots n-1$
- $eV_i$  vector que forma  $V_{i+1} - V_i$

## Idea

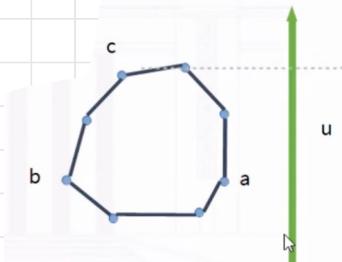


- Agarrar los vértices 2 y 6
- Suponemos que el extremo se encuentra entre ambos
- Luego agarramos un vértice c que esté en el medio y analizamos las pendientes

$[2, \dots, c, \dots, 6]$



$[2, \dots, c] \quad [c, \dots, 6]$



## Casos:

$$1 - \begin{array}{ccc} \nearrow & \searrow \\ 2 & c & \end{array} \rightarrow [2, \dots, c]$$

$$2 - \begin{array}{ccc} \nearrow & \nearrow \\ 2 & c & \end{array} \rightarrow [c, 6] \quad (\text{con } c \text{ ancha de } 2)$$

$$3 - \begin{array}{ccc} \nearrow & \nearrow \\ 2 & c & \end{array} \rightarrow [2, c] \quad (\text{con } c \text{ alijo de } 2)$$

$$3 - \begin{array}{ccc} \searrow & \nearrow \\ 2 & c & \end{array} \rightarrow [c, 6]$$

$$4 - \begin{array}{ccc} \downarrow & \nearrow \\ 2 & c & \end{array} \rightarrow [c, 6] \quad (\text{con } c \text{ alijo de } 2)$$

$$5 - \begin{array}{ccc} \nearrow & \nearrow \\ 2 & c & \end{array} \rightarrow [2, c] \quad (\text{con } c \text{ ancha de } 2)$$

# División y Conquista

## Solución

- Iniciamos con  $\alpha = 0$  y  $b = 0$
- $c = n/2$
- Iteramos actualizando los vértices
- Si solo nos quedan 3 vértices comparamos uno a uno

## Complejidad

- $T(n) = 2T(n/2) + f(n)$
- $\alpha = 1$  porque me quedo con una mitad
- $b = 2$  porque repaso en las mitades
- $f(n) \in O(1)$  porque realiza siempre la misma cantidad de operaciones

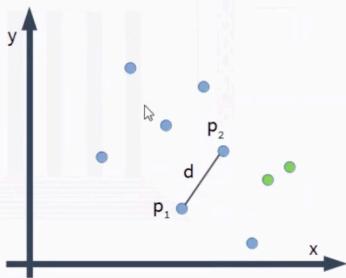
$$T(n) = T(n/2) + O(1)$$

$$\alpha = 1 \quad b = 2 \quad c = 0 \quad \Rightarrow \text{Caso 2: } O(\log n)$$

# División y Conquista

## Puntos cercanos en el plano

- Sea  $P$  un conjunto de " $n^3$ " puntos en el plano
- Quiero buscar los dos puntos más cercanos



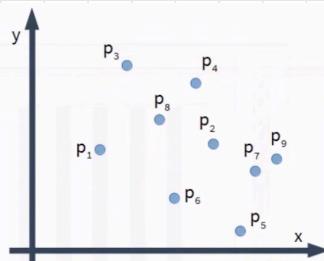
### Ideas

- Vamos a armar dos listas  $P_x$  y  $P_y$ , donde los puntos están ordenados según las coordenadas

$P: p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

$P_x: p_1, p_3, p_8, p_6, p_4, p_2, p_5, p_7, p_9$

$P_y: p_5, p_6, p_7, p_9, p_1, p_2, p_8, p_4, p_3$



# División y Conquista

## Subproblemas

$$P_x = [p_1, p_3, p_8, p_6, p_4, p_2, p_5, p_7, p_9]$$

$$P_y = [p_5, p_6, p_7, p_9, p_1, p_2, p_8, p_4, p_3]$$

- Dividido  $P$  en  $Q$  y  $R$  según sus coordenadas en  $x$

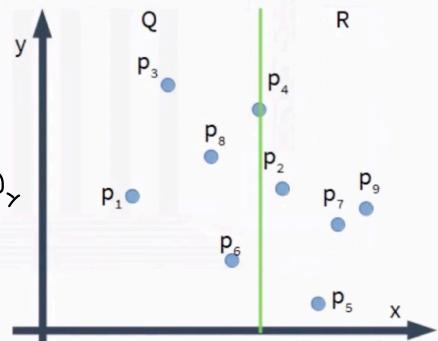
- Para  $Q$  y  $R$  armó  $Q_x, Q_y, P_x, P_y$

$$Q_x = [p_1, p_3, p_9, p_6, p_4]$$

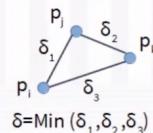
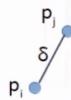
$$R_x = [p_4, p_2, p_5, p_7, p_9]$$

$$Q_y = [p_6, p_7, p_8, p_4, p_3]$$

$$R_y = [p_5, p_7, p_9, p_2, p_4]$$



- Seguirnos dividiendo hasta tener 2 o 3 puntos



# División y Conquista

## Combinación

- Comparamos los pares de puntos retornados de cada subproblema

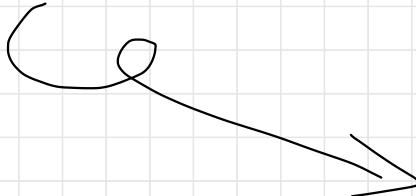
$$\rightarrow S = \min(S_q, S_r)$$

- Hasta aca  $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

↑  
el trabajo que  
hago para  
armar los  
subproblemas

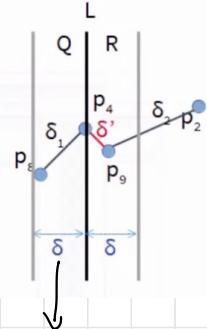
Problema: ¿puedo haber una distancia mínima entre un punto de Q y uno de R

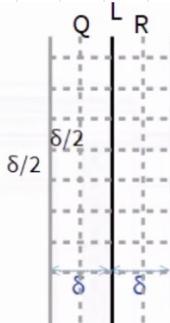
- Para solucionar este problema tengo que tener en cuenta los puntos de ambos subproblemas al combinar



# División y Conquista

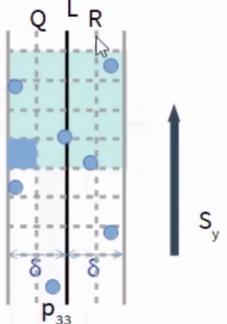
## Solución

- $x^*$  la coordenada del punto Q más a la derecha
- L linea vertical que pasa por  $x^*$  ( $x = x^*$ )
- L separa Q y R (es la linea verde de los ejemplos de antes)
- Si la distancia mínima entre Q y R, los puntos no pueden estar a más distancia de L
- De esta manera sabemos que podemos trabajar sobre los puntos  $p_8, p_4$  y  $p_9$   

$$\delta = \min(\delta_L, \delta_R)$$
- A estos puntos los vamos a llamar  $S \subseteq P$
- S se obtiene recorriendo P en  $O(n)$
- Si: puntos de S ordenados por Y
- S: subdividimos en celdas de  $\frac{\delta}{2} \times \frac{\delta}{2}$  podemos decir que solo puede haber un punto S por celda



# División y Conquista

- De esta manera debemos comparar únicamente puntos que estén en celdas cercanas
- Recorro en orden de  $S_y$
- Solo tengo que recorrer las 15 celdas siguientes
- Osea que haremos  $C = 15$  comparaciones por punto en  $S$  en solo una pasada
- En el peor de los casos Generar  $15n$  comparaciones  $\rightarrow \mathcal{O}(n)$



## Pseudocódigo

```
Puntos-cercanos(P)
    Construir Px y Py // O(nlogn)
    (p0, p1) = puntos-cerc-rec(Px,Py)

puntos-cerc-rec(Px,Py)
    Si |Px|<=3
        retornar (p0, p1) par mas cercano comparando todos los puntos
    Sino
        Construir Qx, Qy, Rx, Rz // O(n)
        (q0, q1) = puntos-cerc-rec(Qx,Qy)
        (r0, r1) = puntos-cerc-rec(Rx,Ry)

        d = min (dist(q0,q1),dist(r0,r1))
        x' = máxima coordenada x de punto en Q
        L = {(x,y) : x=x'}
        S = puntos de P a distancia d de L

        Construir Sy // O(n)
        Por cada punto s de Sy // O(n)
            computar distancia con próximos 15 puntos de Sy
            sea s, s' el par de puntos de menor distancia

        Si dist(s,s')< d
            retornar (s,s')
        Sino si dist(q0, q1)< dist(r0, r1)
            retornar (q0, q1)
        sino
            retornar (r0, r1)
```

# División y Conquista

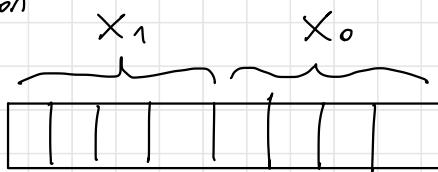
## Karatsuba

- Sean  $X$  y  $Y$  números enteros de  $n$  bits cada uno

- Quiero calcular su multiplicación

$$X = X_1 \cdot 2^{n/2} + X_0$$

$$Y = Y_1 \cdot 2^{n/2} + Y_0$$



### Idea

$$X \cdot Y = X_1 Y_1 \cdot 2^n + (X_0 Y_1 + X_1 Y_0) \cdot 2^{n/2} + X_0 Y_0$$

- En vez de 1 multiplicación de  $n$  bits tenemos 4 de  $n/2$  bits
- Realizamos recursivamente hasta multiplicar 1 bit por 1 bit

$$T(n) = 4T(n/2) + cn \rightarrow O(n^2)$$

### Según Karatsuba ...

$$(X_0 Y_1 + X_1 Y_0) = (X_0 + X_1) \cdot (Y_0 + Y_1) - X_1 Y_1 - X_0 Y_0$$

$$X \cdot Y = X_1 Y_1 \cdot 2^{n/2} +$$

$$[(X_0 + X_1) \cdot (Y_0 + Y_1) - X_1 Y_1 - X_0 Y_0] \cdot 2^{n/2} +$$

$$X_0 Y_0$$

# División y Conquista

## Complejidad

$$T(n) = 3T(n/2) + cn \rightarrow O(n^{1.59})$$

$$T(1) = \alpha$$

. Es más eficiente, sobre todo para  $n$  grande

# Programación Dinámica

## Presentación

- Para problemas de optimización (minimización + maximización)
- Divide en subproblemas con jerarquía
- Cada subproblema puede ser reutilizado en diferentes subproblemas mayores

## Propiedades del subproblema

- Subestructura óptima (si se puede partir en más chicas)
- Subproblemas superpuestos

## Relación de recurrencia

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

• Cada término es definido como una función de términos anteriores

# Programación Dinámica

## Memorización

- Técnica para almacenar resultados previamente calculados
- Para reducir la cantidad de cálculos

# Programación Dinámica

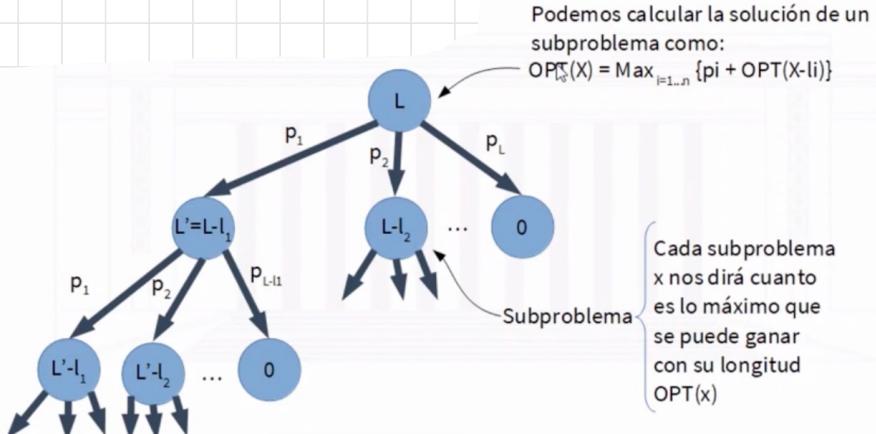
## Corte de sogas

- Sea una soga de longitud  $L$
- Una tabla de precios por longitud de la soga
- Queremos saber que cortes realizar para maximizar la ganancia

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

## Análisis

- Corte  $l_i$  nos da una ganancia  $p_i$
- Nos deja una soga  $L - l_i$
- Tenemos que evaluar todos los cortes posibles



# Programación Dinámica

## Relación de recursividad

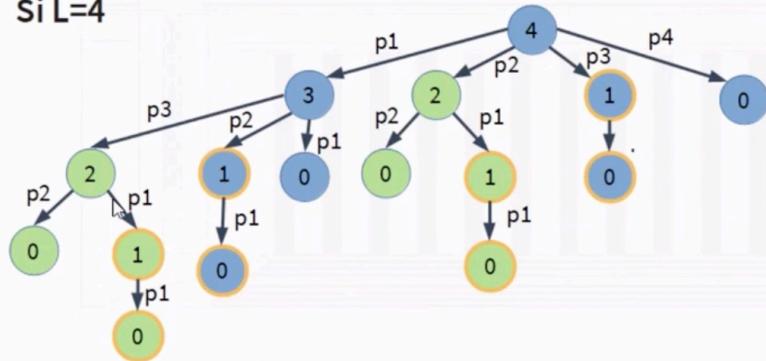
$$OPT(x) = \max_{i=1}^x \{ p_i + OPT(x-i) \} \quad x > 0$$

$$OPT(x) = 0 \quad x \leq 0$$

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

Solo debemos calcular  $L$  subproblemas !!

. Almacenamos los resultados en una tabla

OPT	Gan.
1	$g_1$
2	$g_2$
3	$g_3$
4	$g_4$

# Programación Dinámica

## Solución iterativa

- Resolvemos desde las más chicas hasta las más grandes

$$OPT(0) = p_0$$

$$OPT(1) = p_1$$

$$OPT(2) = \max \{ p_2 + OPT(0), p_1 + OPT(1) \}$$

...

$$OPT(L) = \dots$$

## Complejidad

- En la tabla guardo  $L$  óptimos  $\rightarrow O(L)$  (ESPAZIAL)
- Calculo  $L$  subproblemas y por cada uno los subproblemas  $\rightarrow O(L^2)$  (TEMPORAL)

# Programación Dinámica

## Maximum Subarray problem

- Sea una lista de "n" elementos
- Cada elemento  $e_i$  tiene un valor numérico  $v_i$
- Queremos calcular un subconjunto contiguo de elementos  $S$  tal que la suma sea la máxima posible

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

## Indices

$\text{MAX}(i)$  es el máximo subvector que termina en  $i$

Si  $i = 3$ , tengo 3 subvectores

$$(e_3) = -3$$

$$(e_2, e_3) = 2 \rightarrow \text{me quedo con el} \\ \text{máximo}$$

$$(e_1, e_2, e_3) = -1$$

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

# Programación Dinámica

## Algoritmo Kadane

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

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

## Recurrencia

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

## Complejidad

- Tengo que recorrer todos los elementos del vector

Temporal :  $\Theta(n)$

- Espacial :  $\Theta(1)$

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

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

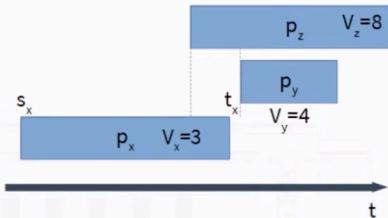
    if MaximoLocal > MaximoGlobal
        MaximoGlobal = MaximoLocal
        IdxFinMaximo = i

Retornar MaximoGlobal
```

# Programación Dinámica

## Weighted Interval Scheduling

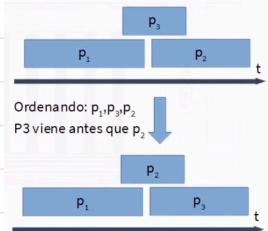
- Cada intervalo tiene un valor asociado
- P conjunto de pedidos  $\{P_1, P_2, \dots, P_n\}$
- $P_i$  tiene
  - $s_i$ : tiempo donde comienza
  - $t_i$ : tiempo donde finaliza
  - $V_i$ : valor



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

### Orden inicial

- Ordenar por tiempo de finalización
- Con esto podemos decir que si  $i < j$ ,  $P_i$  termina antes que  $P_j$



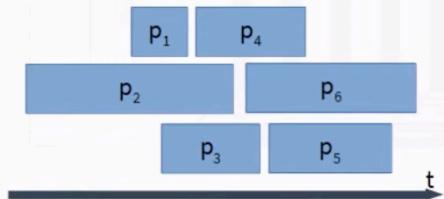
# Programación Dinámica

## Tareas compatibles anteriores

- Nos interesa conocer la primera tarea anterior con la que es compatible  $P(i)$

$$P(i) = x, \text{ con } x < i$$

- 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)

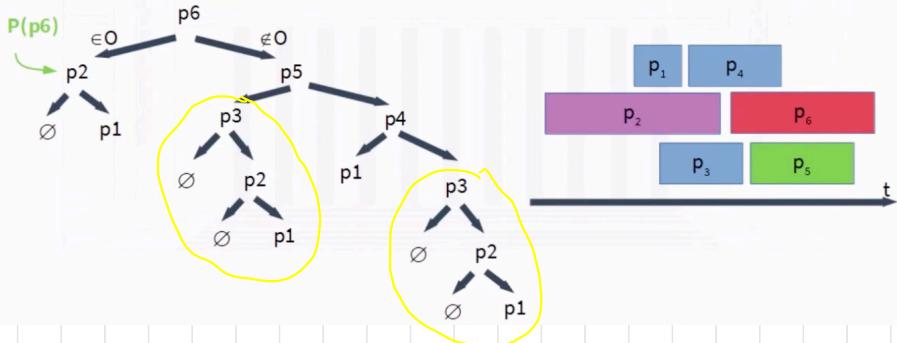
## Perfección de una tarea al óptimo

- Si una tarea pertenece al óptimo, las incompatibles con esa tarea no van a pertenecer al óptimo
- La tarea  $i-1$  podría pertenecer

# Programación Dinámica

## Árbol de decisión

- Suponiendo que  $p_6$  pertenece al óptimo



- Elegimos el mayor valor entre  $V(p_i) + OPT(i-1)$  y  $OPT(p_{\{i\}})$
- Podemos ver que algunos subproblemas se repiten

## Recurrencia

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

## Complejidad

Temporal:  $O(n)$  (solo una iteración)

Espacial:  $O(n)$  (se suman n resultados)

# Programación Dinámica

Reconstruir las elecciones

- Para cada subproblema almaceno si la tasa se eligió

# Programación Dinámica

## Mínimos cuadrados segmentados

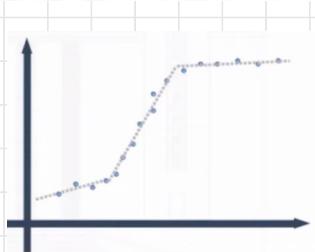
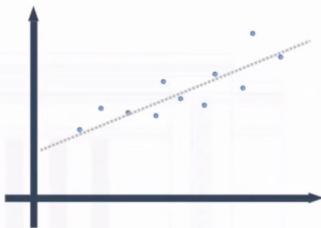
- P un cf de puntos  $p_i = (x_i, \tau_i)$
- Queremos aproximar mediante segmentos los puntos de p
- Minimizando el error

### Segmentos

$$L = ax + b$$

$$a = \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}$$

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



3 segmentos. Error aceptable

### Calculo del error

$$E(L, P) = \sum_i^n (y_i - ax_i - b)^2$$

# Programación Dinámica

Idea

•  $C$  es la penalidad por cada segmento añadido

•  $C > 0$

+  $C \rightarrow$  menos segmentos

-  $C \rightarrow$  menos error

•  $p_n$  es un punto que pertenece el último segmento que empieza en  $p_x$

• En la solución óptima conocemos el error  $e_{x,n}$

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

. Pero no conocemos el óptimo...

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

. Entonces elegir en ese último segmento el que minimice el error general.

# Programación Dinámica

$$\left\{ \begin{array}{l} OPT(x) = \min_{1 \leq x \leq n} (c_{x,i} + c + OPT(x-1)) \\ OPT(0) = 0 \end{array} \right.$$

## Complejidad

- El cálculo de cada óptimo es  $O(n)$
- Se calculan  $n$  óptimos en el peor caso
- El cálculo del error es  $O(1)$

$$O(n^3)$$

$$OPT[0] = 0$$

Para todo par  $i,j$  con  $i \leq j$   
Calcular  $e[i][j]$

Desde  $j=1$  a  $n$

$$OPTIMO[j] = +\infty$$

Desde  $i=0$  a  $j-1$   
segmento =  $e[i][j] + c + OPT[i-1]$

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

Retornar  $OPT[n]$

Especial:  $O(n^2)$  se almacenan errores  $O(n^2)$  y el óptimo  $O(n)$

# Programación Dinámica

## Seam Carving

- Dada una imagen de  $h \times w$  pixeles
- Cada pixel tiene un valor de energía  $e(i, j)$
- Encontrar la veta (horizontal o vertical) de menor energía

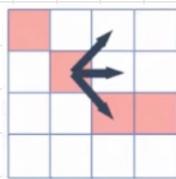
• Ver la imagen como una grilla de pixeles

• De un pixel se puede acceder a otros 3 ( $\circ 2$  en los extremos)



• En la primera columna, si tengo que eliminar un pixel es obvio cual (el de menor energía)

• En la segunda columna la energía acumulada es la del pixel + como se llego



## Subproblema

• Calcular para cada pixel " $j$ " de la columna " $i$ " la energía mínima para llegar a este

Caso base : la energía es la del propio pixel " $j$ "

# Programación Dinámica

## Recurrencia

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

## Complejidad

Temporal :  $\mathcal{O}(w \times h)$

Espacial :  $\mathcal{O}(w \times h)$   
cantidad  
de pixeles

```
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
```

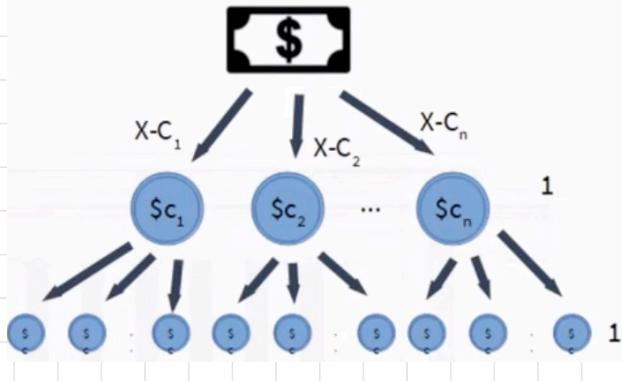
# Programación Dinámica

## Cambio mínimo en monedas

- Tengo un conjunto de monedas de distintas denominaciones
- Un importe  $X$  de cambio a dar
- Quiero entregar la menor cantidad posible de monedas
- Algunos de los caminos son iguales

### Subproblema

- Calcular el óptimo del cambio  $X$  debe usar el mínimo entre los subproblemas  $X - C_j$  para  $j = 1, \dots, n$
- Cada vez que paso por un subproblema incremento en 1 la cantidad de monedas que estoy dando



### Recurrencia

$$\left\{ \begin{array}{ll} \text{OPT}(x) = 0 & x = 0 \\ \text{OPT}(x) = 1 + \min_i \{ \text{OPT}(x - C_i) \} & x > 0 \end{array} \right.$$

# Programación Dinámica

## Complejidad

Temporal :  $O(n \cdot x)$

Espacial :  $O(x)$

- Es un algoritmo pseudo polinómico

$OPT[0] = 0$

Desde  $i=1$  a  $x$

```
minimo = +∞  
Desde j=1 a n  
resto =  $X - C[j]$   
si resto  $\geq 0$  y minimo >  $OPT[resto]$   
minimo =  $OPT[resto]$ 
```

$OPT[i] = 1 + \minimo$

Retornar  $OPT[x]$

## Reconstrucción

- Se almacena el resultado para cada  $C_j$

```
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  $\geq 0$  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  $OPT[x]$ 
```

# Programación Dinámica

## Subset Sums

- $E = \{e_1, \dots, e_n\}$
- $w_i$  el peso asociado a  $e_i$
- Queremos seleccionar un subconjunto de elementos de  $E$  con el mayor peso posible que NO supere un valor  $W$

### Solución

- Si  $e_i \notin \text{solución} \rightarrow \text{MAX-PESO}(e_i, w) = \text{MAX-PESO}(e_{i-1}, w)$
- Si  $e_i \in \text{solución} \rightarrow \text{MAX-PESO}(e_i, w) = w_i + \text{MAX-PESO}(e_{i-1}, w - w_i)$

La mejor solución hasta  $e_i$  es  $\max \{e_i \notin \text{solución}, e_i \in \text{solución}\}$

# Programación Dinámica

## Subproblemas

- $\text{MAX\_PESO}(i, p)$  es el peso máximo que no supera  $p$ , utilizando los primeros  $i$  elementos.
- Nosotros buscamos  $\text{MAX\_PESO}(n, w)$

## Recurrencia

$$\begin{cases} \text{MAX\_PESO}(i, p) = 0 & \text{si } i=0 \text{ o } p=0 \\ \text{MAX\_PESO}(i, p) = \max \begin{cases} w_i + \text{MAX\_PESO}(i-1, p-w_i) \\ \text{MAX\_PESO}(i-1, p) \end{cases} & \text{otro caso} \end{cases}$$

## Complejidad

Temporal:  $O(n \cdot w)$

Espacial:  $O(n \cdot w)$

- Si el peso es muy grande nos obliga a hacer muchas rutas

```
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[i][p] = enOptimo
        sino
            OPT[i][p] = noEnOptimo

Retornar OPT[n,W]
```

# Programación Dinámica

## Knapsacks

- Es una variante del problema anterior
  - $E = \{e_1, \dots, e_n\}$
  - $w_i$  el peso asociado a  $e_i$
  - $v_i$  la ganancia de  $e_i$
- Queremos seleccionar un subconjunto de elementos de  $E$  con el mayor  $\downarrow$  posible que no supere un valor  $W$   
ganancia

## Subproblema

$\text{MAX-GANANCIA}(i, t)$  la ganancia máxima que no supera p  
utilizando los primeros  $i$  elementos

# Programación Dinámica

## Recurrencia

$$\left\{ \begin{array}{l} \text{MAX-GANANCIA}(i, p) = 0 \quad i=0 \text{ o } p=0 \\ \text{MAX-GANANCIA}(i, p) = \begin{cases} v_i + \text{MAX-GANANCIA}(i-1, p-w_i) \\ \text{MAX-GANANCIA}(i-1, p) \end{cases} \end{array} \right.$$

## Complejidad

Temporal:  $\mathcal{O}(n, w)$

Espacial:  $\mathcal{O}(n, w)$

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  $OPT[n, W]$

# Programación Dinámica

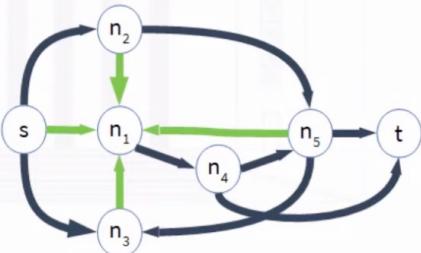
## Bellman Ford

- Algoritmo para camino mínimo SIN ciclos negativos

### Análisis

- Para llegar desde "s" a un nodo  $n_i$  solo puedes llegar desde sus nodos que lo conectan  $\text{pred}[n_i]$

- $j$ : longitud máxima
- $\text{minPath}(n_i, j)$ : camino mínimo que llega a  $n_i$  con longitud entre 0 y  $j$
- $n_x = \text{Pred}(n_i)$  predecessores de  $n_i$



### Recurrencia

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

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

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

ciclo de recorrido  
de  $n_x \rightarrow n_i$

# Programación Dinámica

## Complejidad

$m = \text{número de aristas}$

Temporal:  $\mathcal{O}(m \cdot n)$

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

Desde  $v=0$  a  $n-1$   
 $OPT[0][v] = +\infty$

Desde  $l=1$  a  $n-1$  // max longitud del camino  
Desde  $v=1$  a  $n$  // nodo

$OPT[l][v] = OPT[l-1][v]$

$m$  veces ← Por cada  $p$  predecessor de  $v$   
si  $OPT[l][v] > OPT[l-1][p] + w(p, v)$

$OPT[l][v] = OPT[l-1][p] + w(p, v)$

Retornar  $OPT[n-1, n]$

## Reconstrucción

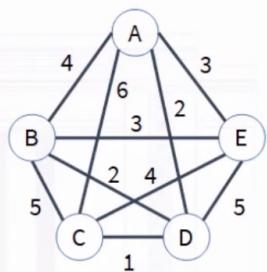
- Agregamos un vector pred de longitud  $n$

# Programación Dinámica

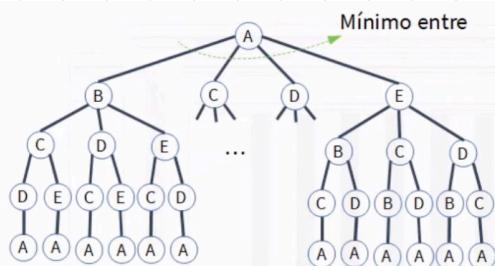
## Problema del viajante (Bellman - Held - Karp)

- $C$ : conjunto de  $n$  ciudades
- El costo de la ruta puede ser simétrico o asimétrico
- Queremos obtener el circuito de menor costo

### Algoritmo Bellman - Held - Karp



descomponemos



Algunas se repiten

$$\left\{ \begin{array}{l} \text{OPT}(i, \{s\}) = \min_{j \in \{s\}} (\omega(i, j) + \text{OPT}(j, \{s-j\})) \\ \text{OPT}(i, \emptyset) = \omega(i, \text{start}) \end{array} \right.$$

# Programación Dinámica

## Complejidad

. C : conjunto de ciudades

. 1 ciudad inicial

Temporal :  $O(n^2 2^n)$

Desde  $i=2$  a  $n$  //ciudad 1 es la inicial  
 $OPT[i][\emptyset] = W[i][1]$

Desde  $k=1$  a  $n-2 \rightarrow O(n)$

Para todo subset S de  $C-\{1\}$  de tamaño k  
 $O(2^n)$   
Para cada elemento i de S  
 $OPT['i', S-\{i\}] = +\infty$

$O(1)$  ↙ 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 =  $+\infty$

Desde  $j=2$  a  $n$

ciclo =  $OPT[i, S-\{1, i\}] + w[1][i]$

Si ( $CaminoMinimo > ciclo$ )

CaminoMinimo = ciclo

Retornar caminoMinimo

# Redes de Flujo

## Presentación

- Problemas que se pueden representar como grafos donde los ejes transportan algún tipo de flujo
- Capacidad: cantidad máxima que un eje puede transportar
- Fuente: vértice que genera tráfico saliente
- Sumidero: vértice que absorbe tráfico
- Flujo: cantidad transportada por un eje

# Redes de Flujo

## Definición formal

- $G = (V, E)$  un grafo dirigido
- $C_e$  la capacidad de cada  $e \in E$
- $s \in V$  es la fuente única (source)
- $t \in V$  es el único sumidero

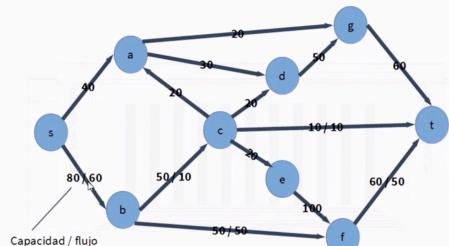
## Flujo s-t

- Es una función  $f : E \rightarrow \mathbb{R}^+$
- Condición de capacidad:  $0 \leq f(e) \leq C_e$
- Condición de conservación:

$$F_{\text{in}}(v) \leftarrow \sum_{e \text{ in } v} f(e) = \sum_{e \text{ out } v} f(e) \rightarrow F_{\text{out}}(v)$$

## Valor del flujo

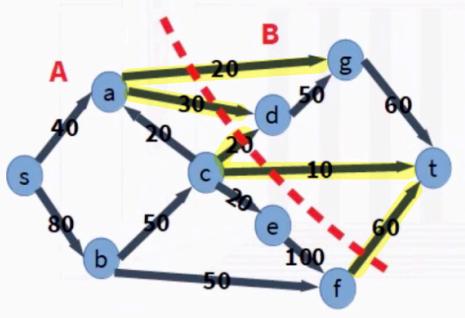
$$V(f) = \sum_{e \text{ out } s} f(e)$$



# Redes de Flujo

## Corte del grafo

- Dividimos el grafo en 2 partes



- El corte define un caudal máximo de flujo

$$C(A, B) = 20 + 30 + 20 + 10 + 60 = 140$$

- Puedo hacer otros cortes que me den una capacidad distinta
- Solo cuento las → que van de izquierdas → derechos

# Redes de Flujo

## Ford Fulkerson

- Algoritmo para encontrar el flujo máximo

### Grafo residual

- Dado un grafo  $G$  y un flujo  $f$  en  $G$
- $G_f$  es el grafo residual con los mismos vértices de  $G$

### Ejes hacia delante:

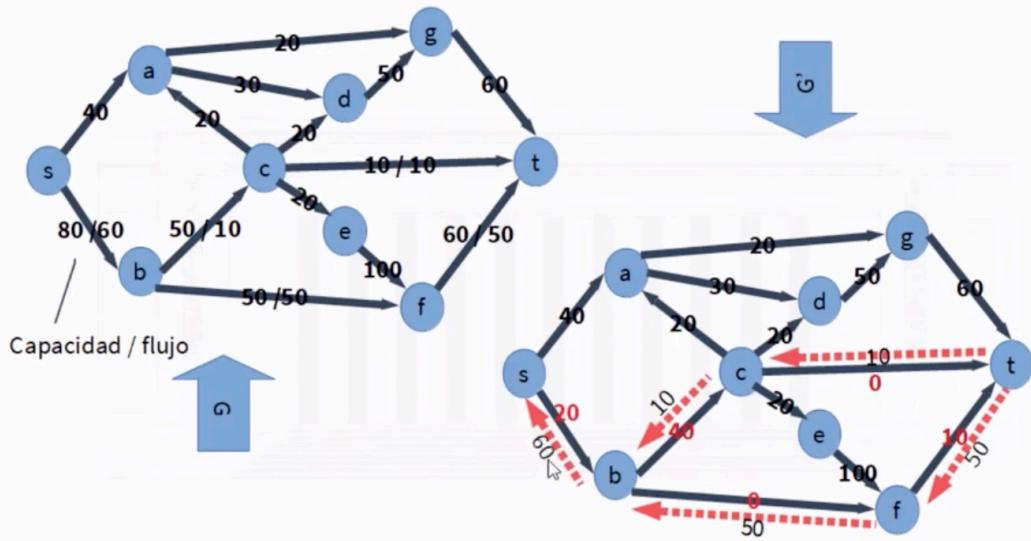
Los incluyo si  $f(e) < C_e$  con capacidad  $C_e - f(e)$

### Ejes hacia atrás:

Los incluyo si  $f(e) > 0$  con capacidad  $f(e)$

# Redes de Flujo

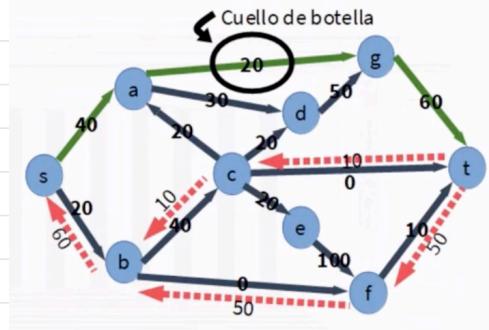
Ejemplo:



# Redes de Flujo

## Cuello de botella

- $P$ : camino entre  $s$  y  $t$  que no visita un vértice más de una vez
- Bottleneck: es la capacidad residual mínima de cualquier eje de  $P$  con respecto al flujo  $f$



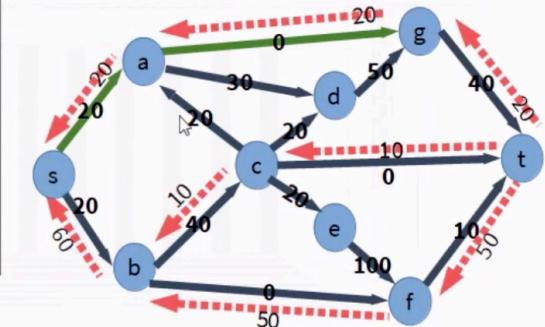
## Camino de aumento

augment( $f$ ,  $P$ )

Sea  $b = \text{bottleneck}(P, f)$

Para cada eje  $e = (u, v) \in P$   
Si  $e = (u, v)$  eje hacia adelante  
 $f(e) + b$  en  $G$   
Sino si es eje para atrás  
 $e' = (v, u)$   
 $f(e') - b$  en  $G$

Retornar  $f$



- Le restamos 20 a los ejes del camino  $P$

# Redes de Flujo

## Validar nuevo flujo

- Condiciones de conservación se cumplen
- Solo se modifican ejes del camino de aumento

## Pseudocódigo Ford-Fulkerson

Max-Flow

Inicialmente  $f(e) = 0$  Para todo  $e$  en  $G$

Mientras haya un camino  $s-t$  en  $G_f$

Sea  $P$  un camino  $s-t$  simple en  $G_f$   
 $f' = \text{augment}(f, P)$

Actualizar  $f$  para ser  $f'$

Update  $G_f$  para ser  $G_f'$

Retornar  $f$

1 - Los valores de flujo y capacidades son enteros

2 - En cada instancia el valor del flujo aumenta

3 - El algoritmo termina en un número finito de iteraciones

4 -  $V(f) = f_{out}(A) - f_{in}(A)$  ( $\text{flujo total de un vértice } A$ )

5 -  $V(f) \leq c(A, B)$

$C = \text{flujo máximo}$

$E = \text{cantidad de ejes}$

$(A, B) = \text{cualquier corte}$

Complejidad temporal

$\Theta(C \cdot (E + V))$

$\Theta(C \cdot E)$  si tengo  
más ejes que vértices

Es óptimo !

# Redes de Flujo

## Edmond - Karp

- Es como Ford - Fulkerson pero utiliza BFS para encontrar el camino de menor longitud posible

Complejidad espacial:  $O(V + E)$  (grafo residual)

Complejidad temporal:  $O(V) \leftarrow$  caminos de máxima long

$O(V \cdot E) \leftarrow$  caminar de aumento

$O(V \cdot E^2) \leftarrow$  cada uno lo calcula  
 $V + E$

# Redes de Flujo

## Circulación con demandas

- S sea  $G = (V, E)$

- $d_v$ : demanda enresa de  $v \in V$

$d_v > 0$  : es sumidero  
 $d_v = 0$  : es interno  
 $d_v < 0$  : es fuente

- Existen:

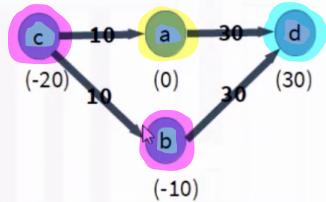
- set S de fuentes

- set T de sumideros

- S y T pueden tener vértices de entrada y de salida

- La circulación con demanda es una función  $f$  que asigna un número real no negativo a cada vértice

. DETERMINAR si existe una circulación que cumple con las condiciones de demanda.



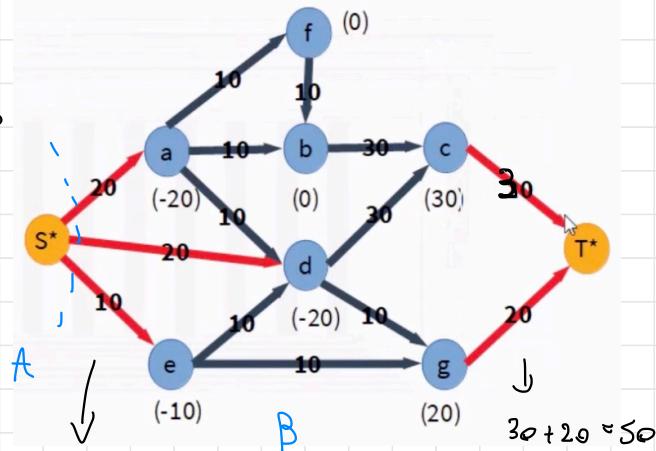
# Redes de Flujo

- $\sum d_v = \sum f_{in}(v) - f_{out}(v)$
- Si existe una circulación con demandas  $\{d_v\}$  tales  $\sum d_v = 0$

Reducir al problema de flujo máximo

- Creamos una "super" fuente  $S^*$
- Creamos un "super" sumidero  $T^*$
- A los vértices con  $d_v > 0$  los conectamos con  $T^*$  y  $C_c = d_v$
- A los vértices con  $d_v < 0$  los conectamos con  $S^*$  y  $C_c = -d_v$
- Resolvemos el problema
- Hacemos un corte alejando  $S^*$  en un lado

↳ si  $f_{out}(A) = D$   
entonces existe una  
circulación con demanda  
factible



$20 + 20 + 10 = 50 \rightarrow$  EXISTE!

# Redes de Flujo

## Circulación con demandas y límite inferior

- Se agrega como condición que ciertos ejes tengan un flujo mínimo

$$l_e \leq f(e) \leq C_e$$

- Construimos un nuevo grafo  $G^*$  ficticio donde forzaremos que los ejes con un límite inferior tengan  $l_e$  como mínimo

- Cada eje cambia su capacidad a  $C_e - l_e$

- Cada vértice cambia su demanda a  $d_v - L_v \rightarrow L_v = \sum l_{e, \text{in}} - \sum l_{e, \text{out}}$

$$L_d = 0 - 5 = -5$$

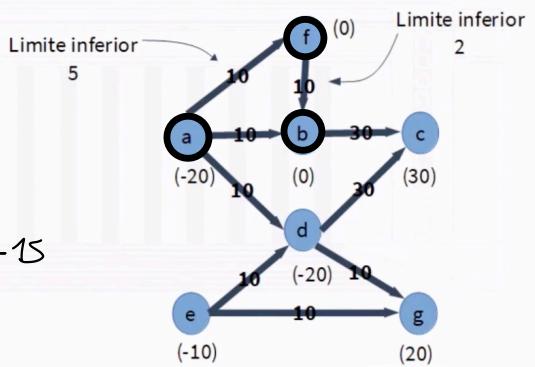
$$L_f = 5 - 2 = 3$$

$$L_b = 2 - 0 = 2$$

$$D_d^* = D_d - L_d = -20 - (-5) = -15$$

$$D_f^* = 0 - 3 = -3$$

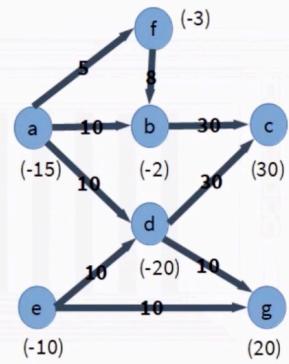
$$D_b^* = 0 - 2 = -2$$



# Redes de Flujo

$$C_{af}^* = C_{af} - l_{af} = 10 - 5 = 5$$

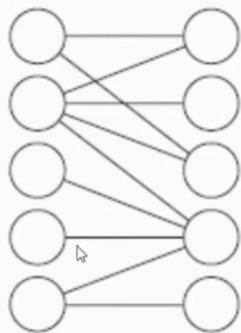
$$C_{fb}^* = C_{fb} - l_{fb} = 10 - 2 = 8$$



# Redes de Flujo

## Grafo bipartito y Matching

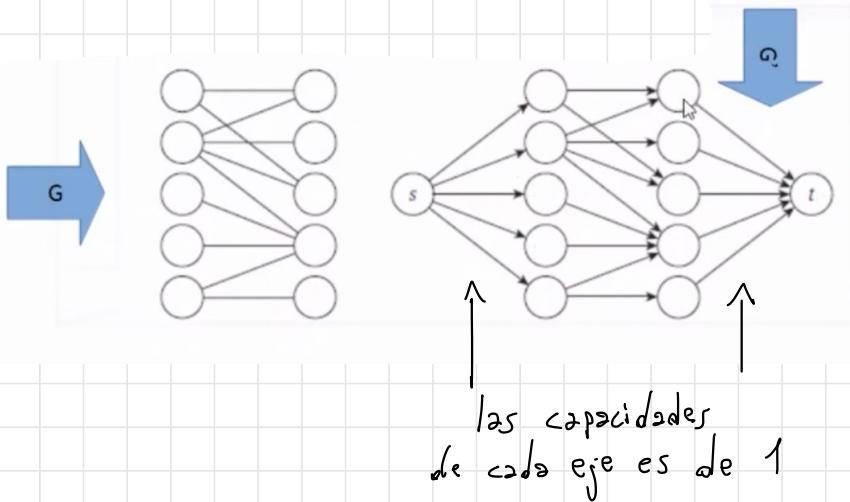
- Un grafo es bipartito si se puede dividir en  $V = X \cup Y$   
 $\uparrow$   
unión
- Tal que cada eje sale de  $X$ , llega a  $Y$
- Un matching  $M$  en  $G$  es el subconjunto de ejes tal que cada nodo aparece como mucho en un eje.
- Si quiere buscar el set  $M$  del mayor tamaño posible



# Redes de Flujo

Resolución utilizando problema flujo máximo

- Construimos un nuevo grafo  $G'$  con un "super" sistema

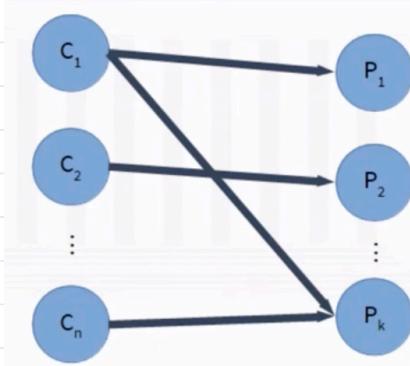


- Ahora resolvemos el problema de flujo máximo
- El valor del flujo  $\text{f}_{\text{total}} = \text{matching máximo}$
- Aquellos ejes que van de  $X$  a  $Y$  con flujo 1 forman una pareja
- La suma del flujo de los ejes que salen de  $s$  nos da la cantidad de parejas

# Redes de Flujo

## Diseño de encuestas

- $k$  productos que vende una empresa
- $n$  clientes que realizarán compras
- Construir una encuesta de satisfacción personalizada
  - Cada cliente puede responder por los productos que compró
  - El cliente  $i$  puede responder consultas entre  $c_i$  y  $c_i'$  productos
  - El producto  $j$  debe tener entre  $p_j$  y  $p_j'$  respuestas de clientes



$c_i$ : consultas mínimas

$c_i'$ : consultas máximas

# Redes de Flujo

## Restricciones

- Para el cliente  $i$  tenera que elegir un subconjunto  $X$  de sus relaciones para que cumpla

$$C_i \leq |X| \leq C'_i$$

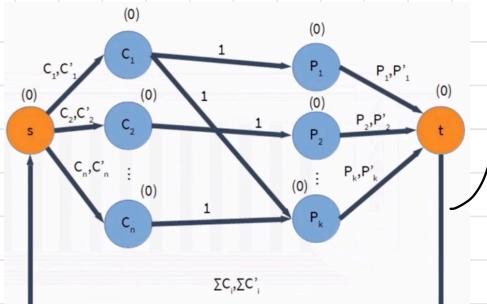
- Para el producto  $j$  hacer lo mismo

$$P_j \leq |Y| \leq P'_j$$

## Reducción a flujo máximo

- Cada cliente y producto es un nodo
- Cada eje tiene 1 capacidad
- Agregamos los "super" nodos
- Los "super" ejes tienen capacidad y límite inferior
- Agregamos eje que va de  $t$  a  $s$

Como  $s$  no produce  
nada, tengo que unir  
 $t$  y  $s$ .  
(ver apunte)



- Arrancan con demanda cero

# Redes de Flujo

## Solución

- . Quedo un problema de circulación con demanda y límite inferior
- . Lo reducimos a un problema con demanda y (vego de flujo máximo)
- . Resolvemos con Ford-Fulkerson

# Redes de Flujo

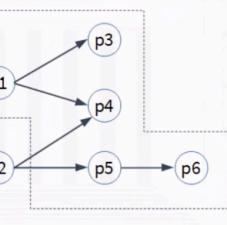
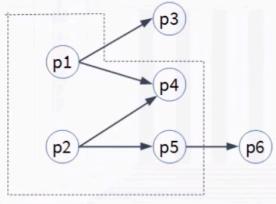
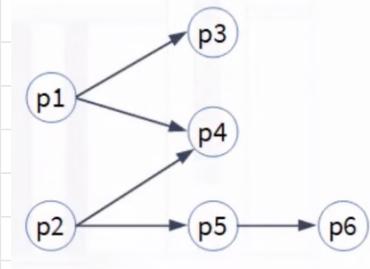
## Selección de Proyectos

- P proyectos para seleccionar
- Cada proyecto tiene un retorno económico
- Cada proyecto puede tener un subconjunto de problemas que son requisito
- Seleccionar proyectos que maximicen ganancia

### Factibilidad de proyectos

Un subconjunto de proyectos  $A \subseteq P$  es factible

Si los requisitos de cada proyecto en A, también pertenecen a A



# Redes de Flujo

## Ganancia

- Dado un subconjunto factible A

$$\text{Ganancia}(A) = \sum p_i$$

. Para los proyectos con relojero positivo llamaremos tope de ganancia a la suma de los relojeros

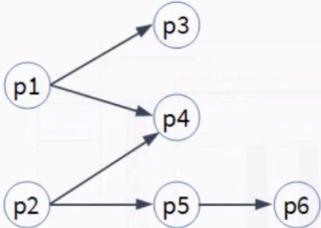
$$C = \sum_{\text{if } p_i > 0} p_i$$

## Construcción de red de flujos

- Agregar "super" nodos
- Los ejes entre los proyectos tendrán capacidad  $C + 1$
- Los ejes de los "super" nodos tendrán la ganancia

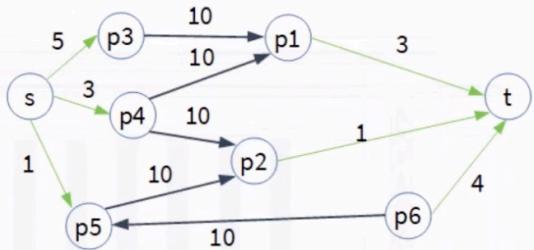
# Redes de Flujo

Ejemplo



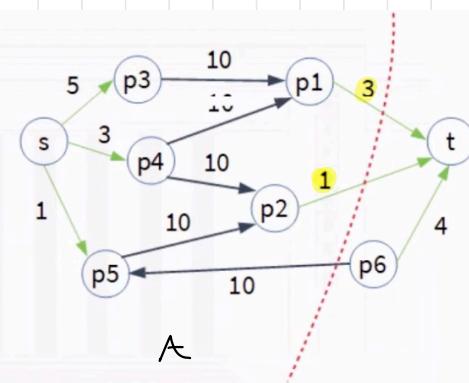
i	1	2	3	4	5	6
p	-3	-1	<u>5</u>	<u>3</u>	<u>1</u>	-4

$$\text{tope ganancia: } C = 5 + 3 + 1$$



Corte mínimo  $C(A', B')$

Corresponde a la ganancia máxima



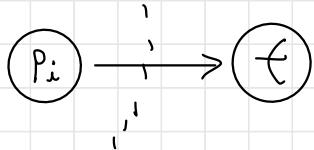
En este caso la ganancia máxima sería 4

# Redes de Flujo

## Análisis

, Los ejes que ingresan a " $t$ " contribuye así al corte

$$\sum_{i \in A / p_i < 0} -p_i$$



, Los que salen de " $s$ "

$$\sum_{i \notin A / p_i > 0} p_i$$



$$C(A', B') = \sum_{p_i < 0} -p_i + \sum_{p_i > 0} p_i$$

$$C(A', B') = C - \sum_{i \in A} p_i$$

$$C(A', B') = C - \text{Ganancia}(A)$$

En el ejemplo :  $C(A', B') = -(-3) + -(-1) + 0 = 4$

$$C(A', B') = 9 - (-3 - 1 + 5 + 3 - 1) = 4$$

# Redes de Flujo

## Segmentación de una imagen

- Queremos quitar el fondo de una imagen



- $V$ : conjunto de píxeles que forman la imagen (cuadrícula)
- Cada pixel  $i$  tiene un conjunto de píxeles vecinos

↳ 8 si son internos



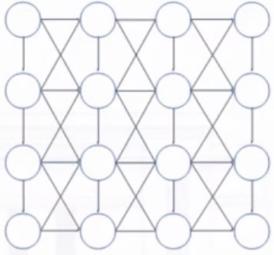
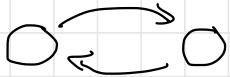
↳ 3 o 5 si son externos



# Redes de Flujo

## Representación

- Podemos verlo como un grafo  $G = (V, E)$
- Cada relación es bidireccional
- Para evitar tener ejes bidireccionales los podemos partir en dos



## Probabilidad de pertenencia

- Para cada pixel tenemos
  - $a_i$ : valor "deseo" primer plano
  - $b_i$ : valor "deseo" fondo
- $a_i > b_i \rightarrow$  pertenece al primer plano
- $a_i < b_i \rightarrow$  pertenece al fondo

Pero también hay que tener en cuenta la decisión de los vecinos

# Redes de Flujo

## Penalidad por cambio

- Si muchos vecinos de  $i$  pertenecen al fondo, entonces  $i$  debería también
  - $p_{ij}$ : penalidad del par de pixeles  $(i, j)$  de que pertenezcan a diferentes segmentos
- A : conjunto de pixeles del primer plano
- B : conjunto de pixeles del fondo
- La "descalificación" se calcula como:

$$q(A, B) = \sum_{i \in A} \alpha_i + \sum_{j \in B} \beta_j - \sum_{\substack{i \in A \\ j \in B \\ i, j \text{ vecinos}}} p_{ij}$$

- Queremos seleccionar la elección del segmento A y B que maximice  $q(A, B)$

# Redes de Flujo

Transformación a un problema de minimización

$$Q = \sum_{i \in E} a_i + b_i$$

$$\sum_{i \in A} a_i + \sum_{j \in B} b_j = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j$$

$$q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{i \in A \\ j \in B}} p_{ij}$$

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{i \in A \\ j \in B}} p_{ij}$$

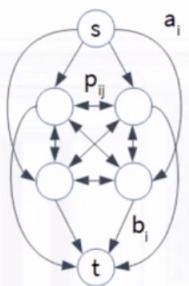
# Redes de Flujo

Transformación en un problema de corte mínimo

- "s" segmento fondo
- "t" segmento primos plano
- Se agregan:
  - eje s-i por cada pixel con capacidad  $a_i$
  - eje i-t por cada pixel con capacidad  $b_i$
  - eje i-j por cada pixel vecino con capacidad  $p_{ij}$  ( $\geq$  generar un eje de ida y otro de vuelta)

No queda  $G' = (V', E')$  con:

- $V' = \text{pixeles} + \{s, t\}$
- $E' = \text{ejes vecindad} + \text{ejes } s-i + \text{ejes } i-t$



# Redes de Flujo

## Corte s-t

Si analizamos

un coste A-B generico, con  $s \in A$  y  $t \in B$

Llamaremos  $c(A,B)$

A la capacidad del corte A-B

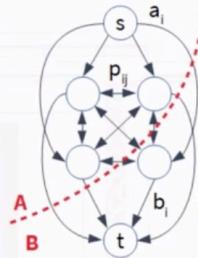
(suma de las capacidades de los ejes que van de nodos de A a nodos de B)

Los ejes de  $c(A,B)$  corresponden a

Ejes  $s-j$ , con  $j \in B$  (con capacidad  $a_j$ )

Ejes  $i-t$  con  $i \in A$  (con capacidad  $b_i$ )

Ejes  $i,j$  con  $i \in A$  y  $j \in B$  (con capacidad  $p_{ij}$ ) ← los ejes de vecindad



- Si sumamos las capacidades nos da lo mismo que  $c'(A',B')$
- Nos interesa el corte mínimo

## Corte mínimo

- Se resuelve el problema del flujo máximo
- Quedan separados nodos alcanzables desde  $s$  y no alcanzables (que se unen a  $t$ )
- La frontera serán ejes con capacidad cara a A o B
- Estos ejes conforman el corte mínimo

# Redes de Flujo

## Resolución

- Construir red de flujo
- Resolver mediante FF
- Con el grafo residual, realizar BFS desde  $S$ 
  - Los nodos alcanzables desde  $S$  serán del "fondo"
  - El resto será del primer plano

# Complejidad

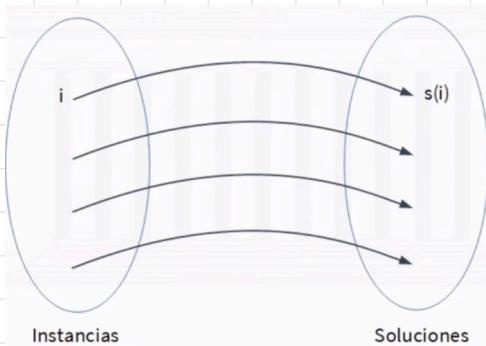
## Presentación

- Tractable: algoritmo con resolución polinomial
- Queremos poder clasificar los problemas de acuerdo a su complejidad

## Problema abstracto

- Relación binaria entre la instancia y la solución

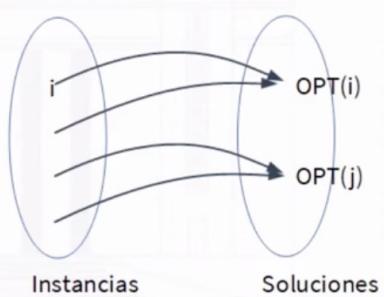
↑  
ej: red de flujo



## Complejidad

## Problema de optimización

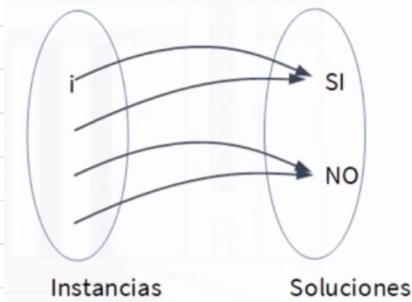
- Buscan la mejor solución para maximizar o minimizar un resultado
  - Por ejemplo: ganancia en mochila, cambio de monedas, flujo transportado en una red



- Para cada problema vamos a tener varias instancias

## Problema de decisión

- Son aquellos donde la solución puede tomar 2 valores: Si / No
  - Por ejemplo: ¿es número primo? ¿existe matching perfecto entre A y B? , el grafo es conexo? ¿se satisface la demanda de flujo?



# Complejidad

## Adecuación de problema de optimización a decisión

- Q: problema de optimización
- Queremos reformular Q para poder resolverlo en tiempo polinómico

Ejemplo:

- Optimización: Cambio mínimo en monedas del valor X
- Decisión: se puede dar el cambio?

Cambio mínimo en monedas del valor X

Desde #m=1 hasta X

Si (Se puede dar cambio cambio mínimo de X con #m monedas)  
retornar #m

## Caracterización de la entrada de un algoritmo

- Tenemos que representar la instancia de modo que una computadora lo entienda
- Codificamos los parámetros como una cadena binaria "s"
- La longitud de la cadena "s" es  $n = |s|$
- n se utiliza para medir la complejidad

# Complejidad

¿Cuando un algoritmo es EFICIENTE?

. S: problema de decisión

. Un algoritmo A resuelve eficientemente si :

- Para toda instancia i de S encuentra la solución en tiempo polinomial

↳  $O(n^k)$ , tiene que existir k constante

# Complejidad

## Clase "P"

- P : conjunto de problemas de decisión para los que existe un algoritmo eficiente que los **resuelve**

### Certificación eficiente

- Un algoritmo  $B$  certifica (o verifica) eficientemente un problema de decisión  $S$ 
  - ↳ si para todo  $i \in S$  existe un certificado  $t$  que puede verificar en tiempo polinomial
- Certificado  $t$  contiene evidencia de la solución de la instancia  $i$ 
  - existe  $k$  constante tal que  $B \leq O(n^k)$
- El algoritmo  $B$  recibe dos parámetros : instancia  $i$  y certificado  $t$ 
  - ↳ responde SI / NO

# Complejidad

Ejemplo: viajante de comercio

¿ Existe un camino de longitud menor a  $L$  para un grafo

$$G = (C, R) \text{ (ciudades y rutas)}$$

• El algoritmo de certificación recibe:

- el grafo (instancia)

- circuito  $C$  de ciudades para el viajante (certificado)

• Calcula y verifica (en tiempo polinomial)

↳ agarra el circuito, calcula la longitud y verifica que sea menor a  $L$  y atraviese todas las ciudades

• Devuelve si o no

# Complejidad

## Clase "NP"

- NP: conjunto de problemas de decisión para los que existe un algoritmo eficiente que los certifique

¿ P ⊆ NP? (son todos P pertenecientes a NP?)

- Si  $Q \in P$ , existe un algoritmo  $A = O(n^k)$  que lo resuelve
- $B(I, t)$  es el algoritmo que certifica en tiempo polinomial

```
B(I,t)
  s = A(I)
  Si s == t
    retornar "si"
    retornar "no"
```

Si  $Q \in P \rightarrow Q \in NP$

# Complejidad

¿ NP ⊆ P? .

- Existe  $B = O(n^k)$  que lo certifica
- Solo si existe un algoritmo A polinomial que resuelva
- En ese caso  $P = NP$ , tiene la misma complejidad  
resolver que verificar

P vs NP

No tiene respuesta! (Algunos piensan que  $P \neq NP$ )

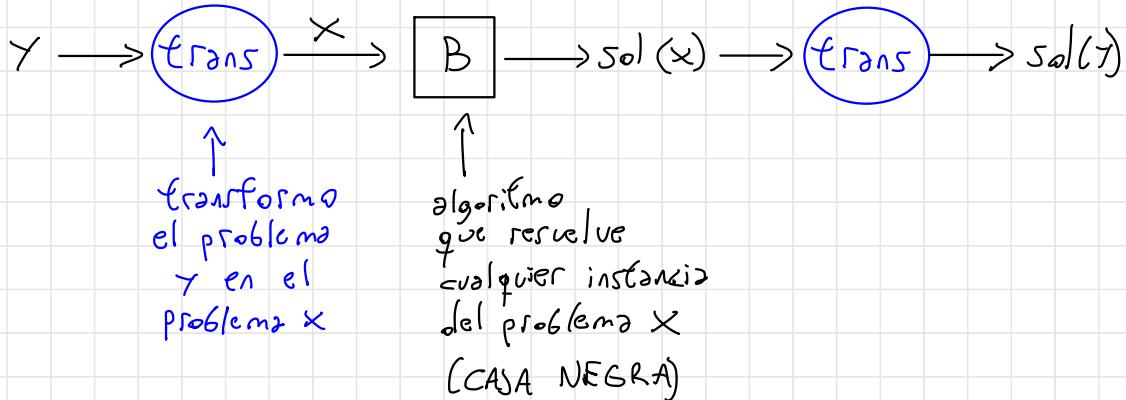
# Complejidad

## Reducciones polinomiales

- Queremos clasificar problemas computacionales de acuerdo a la complejidad de su resolución
- Necesitamos una herramienta para comparar 2 problemas y asignarlos a un clase ( $P$ ,  $NP$ )

## Reducciones

- Lo pensamos como una caja negra



# Complejidad

## Reducción polinomial

- Ambas transformaciones se realizan en tiempo polinomial
- Sean  $X, Y$  problemas

↳  $Y \leq_p X$  ( $Y$  es polinomialmente reducible a  $X$ ) si podemos transformar cualquier instancia de  $Y$  en una instancia de  $X$  en tiempo polinómico

- Las reducciones funcionan como cajón negro y como medida de complejidad para comparar problemas

## Comparar con reducciones

- Si  $Y \leq_p X$  podemos decir que  $X$  es al menos tan difícil que el problema  $Y$

# Complejidad

Ejemplo:

MAX-MATCHING

↑

- Problema: hallar el matching más grande en un grafo bipartito
- Sabemos que se puede reducir polinomialmente al problema de flujo máximo en una red

↓

MAX-FLOW

$$\text{MAX-MATCHING} \leq_p \text{MAX-FLOW}$$

(MAX-FLOW es al menos tan difícil de resolver que MAX-MATCHING)

# Complejidad

Aclarar un problema a la clase P

- Sean  $X, Y$  problemas
  - Si  $X \in P$  y  $Y \leq_p X$  entonces  $Y \in P$
- 

- Sean  $X, Y$  problemas
- Si  $Y \notin P$  y  $Y \leq_p X$  entonces  $X \notin P$

Equivalencias

- Si  $X \leq_p Y$  y  $Y \leq_p X$ , entonces tienen la misma complejidad

Transitividad

- Si  $Z \leq_p T$  y  $T \leq_p X \rightarrow Z \leq_p X$

# Complejidad

## Problemas NP Completos

### Boolean satisfiability problem (SAT)

- Dado un conjunto de variables booleanas
- Que definen una expresión booleana (usando OR, AND, NOT, etc)
- Queremos determinar si existen valores que den True

(no se conoce como resolverlo en tiempo polinomial)

### SAT E NP

- Sea  $i$  una instancia del problema SAT
- Un certificado corresponde  $\Rightarrow$  un valor de asignación de cada variable
- Si la asignación de variables da TRUE podemos certificar en tiempo polinomial

#### Ejemplo

$$I = (V_1 \text{ or } V_2 \text{ or } \bar{V}_3) \text{ and } (\bar{V}1 \text{ or } V4) \text{ and } (V_2 \text{ or } V_3 \text{ or } \bar{V}_4 \text{ or } V_5) \text{ and } (\bar{V}_2 \text{ or } \bar{V}_5 \text{ or } \bar{V}_4)$$

$$V_1 = \text{true} \quad V_2 = \text{false} \quad V_3 = \text{true} \quad V_4 = \text{true} \quad V_5 = \text{false}$$

# Complejidad

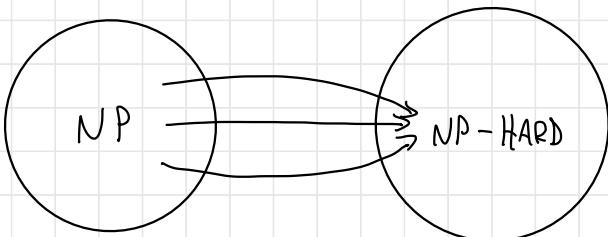
## Teorema Cook - Levin

$$\cdot S: X \in NP \rightarrow X \leq_p SAT$$

“Todo problema perteneciente a NP es a lo sumo tan complejo de resolver como SAT”

## NP - HARD

- Sea  $X$  un problema tal que para todo  $Y \in NP$   
 $Y \leq_p X$
  - Entonces  $X \in NP\text{-HARD}$
- “ $X$  es al menos igual de difícil que cualquier problema NP”

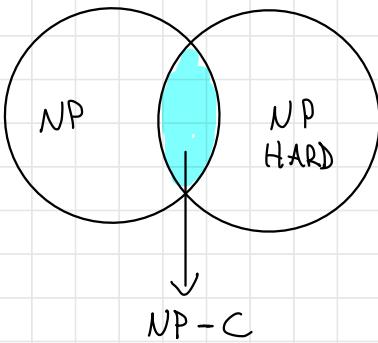


# Complejidad

## NP - COMPLETE

- Sea  $X \in \text{NP-HARD}$  y  $X \in \text{NP}$
- Entonces  $X \in \text{NP-C}$

" $X$  es uno de los problemas más difíciles de NP"



$\text{SAT} \in \text{NP-C}$

# Complejidad

Probar que un problema es NP-C

- Sea  $X$  el problema
- Definir el certificado eficiente  $\rightarrow X \in \text{NP}$
- Dado  $Y \in \text{NP-C}$ , reducir polinómicamente  $Y \geq X \rightarrow X \in \text{NP-H}$

$$X \in \text{NP-C}$$

Probar que  $P = \text{NP}$

- Tomar  $X \in \text{NP-C}$
- Construir un algoritmo que resuelva  $X$  en tiempo polinomial

Todo problema NP-C se puede reducir en tiempo polinomial

- Y todo problema NP se puede reducir a NP-C

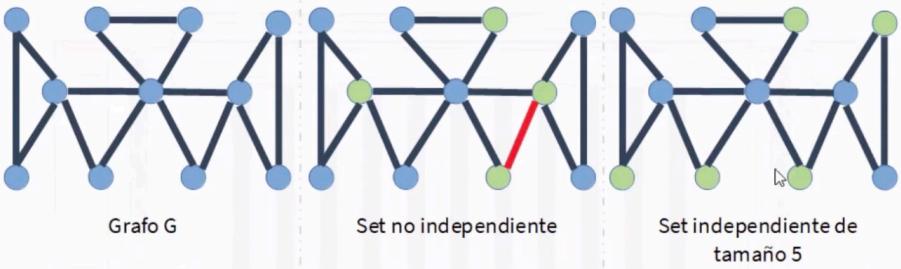
Entonces  $P = \text{NP}$

# Complejidad

## Conjunto independiente

- Sea un grafo  $G = (V, E)$
- Un valor  $k$
- Determinar si existe un conjunto independiente de nodos de como mucho tamaño  $k$
- Un conjunto de nodos  $C \subseteq V$  es independiente si no existe  $a, b \in C$  unidos por un eje
- El tamaño del conjunto es la cantidad de nodos dentro del conjunto  $C$

### Ejemplo



# Complejidad

¿ Conjunto independiente es NP ?

. Dado  $G = (V, E)$

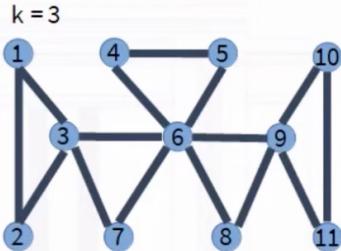
.  $k$  tamaño del conjunto

.  $T$  certificado = subconjunto de nodos

. Puedo verificar que (en tiempo polinomial)

-  $|T| = k \rightarrow T = \{3, 4, 9\} \quad \checkmark$   
 $3 = 3$

-  $\forall a, b \in T, \nexists (a, b) \in E \quad \checkmark$



INDEPENDENT-SET  $\in$  NP

# Complejidad

¿Conjunto independiente  $\in P$ ?

- No se conoce un algoritmo que lo resuelva en tiempo polinómico
- Pero si podemos probar que es  $NP-C$  usando 3SAT y en ese caso  $\text{INDEPENDENT-SET} \in P \Leftrightarrow P = NP$

## 3SAT

, Es una variante de SAT  $\rightarrow SAT \leq_p 3SAT \rightarrow 3SAT \in NP-C$

### Dado

$X = \{x_1, \dots, x_n\}$  conjunto de  $n$  Variables booleanas  $= \{0, 1\}$

$k$  cláusulas booleanas  $T_i = (t_{i1} \vee t_{i2} \vee t_{i3})$   $\leftarrow$  en 3SAT siempre tienen 3 variables por cláusula  
Con cada  $t_j \in X \cup \bar{X} \cup \{1\}$

### Determinar

Si existe asignación de variables tal que  $T_1 \wedge T_2 \wedge \dots \wedge T_k = 1$

Sea la expresión

$$E = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

Ejemplo  $\rightarrow$

La asignación

$$x_1 = 0 \quad x_2 = 0 \quad x_3 = 0 \quad x_4 = 0, \text{ Genera } E = 0$$

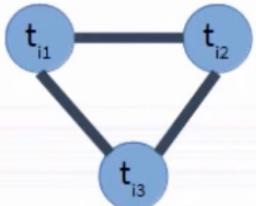
$$x_1 = 1 \quad x_2 = 0 \quad x_3 = 0 \quad x_4 = 1, \text{ Genera } E = 1$$

# Complejidad

## Reducción de 3SAT a INDEPENDENT-SET

. Por cada  $T_i$  se crean 3 vértices

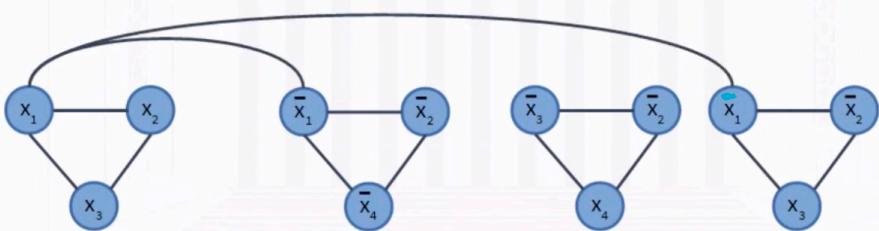
. El grafo resultante  $G$  corresponde a una instancia del problema IND-SET



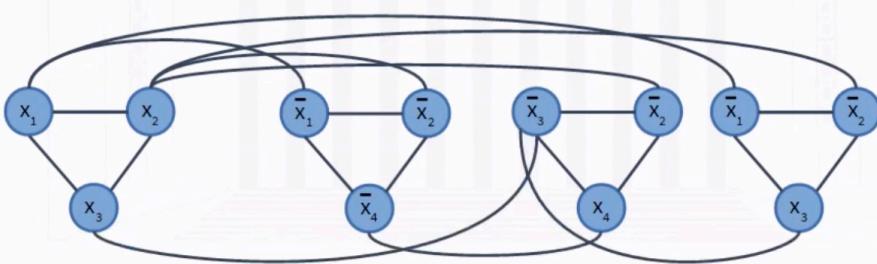
Sea la expresión

$$E = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

La reducimos polinomialmente a:

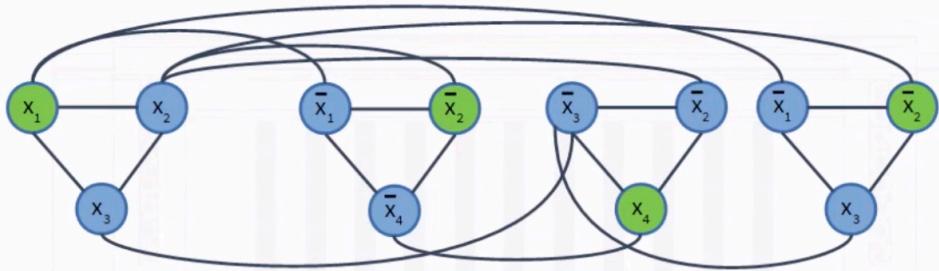


Nos queda así:



# Complejidad

Si lo resuelvo con  $k=4$



Entonces resuelvo 3SAT

$$x_1 = 1 \quad \bar{x}_2 = 1 \rightarrow x_2 = 0 \quad x_4 = 1 \quad x_3 = 0 \text{ (en este caso es indistinto 0 o 1)}$$

• Entonces como

INDEPENDENT-SET  $\in NP$   
 $3SAT \leq_p INDEPENDENT-SET$

}  $\rightarrow$  INDEPENDENT-SET  $\in NP-C$

# Complejidad

# Vertex Cover

- $G = (V, E)$
  - Diremos que el subset  $S \subseteq V$  es una covertura de vértices: para todo vértice de  $G$ , alguno de los vértices está en  $S$

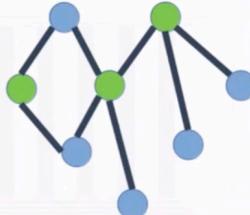
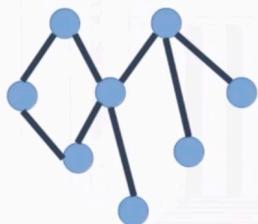
$\forall e \in E = (u, v), u \in S \vee v \in S$

Como problema de decisión: la cubierta tiene un tamaño al menos  $k$ ?

Jump to

## Dado el Grafo

Existe una cobertura de vértices de  $k=3$ ?



# Complejidad

¿ VERTEX-COVER ∈ NP ?

- $G = (V, E)$
- Certificado  $t$ : conjunto de nodos de  $V$  que forman el cuñamiento
- Verificamos que
  - $\forall e = (u, v) \in E, u \in t \wedge v \in t \rightarrow O(V, E)$
  - si:  $|t| = k$  (tenemos  $k$  elementos en el certificado)

VERTEX-COVER ∈ NP

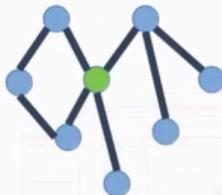
Relación entre INDEPENDENT-SET y VERTEX-COVER

- $S$ : conjunto independiente de tamaño  $|S|$
- $C = V - S$  (complemento de  $S$ )
- $\forall e = (u, v) \in E, u \in S \rightarrow v \in C$  (porque  $S$  es set ind)
- Para todo eje al menos un vértice pertenece a  $V - S$

$V - S$  es una cubierta de tamaño  $|V - S|$

# Complejidad

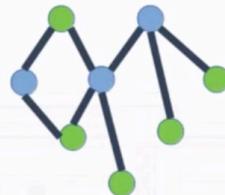
Ejemplos



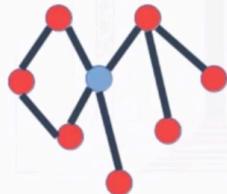
Set independiente  $k=1$



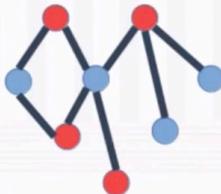
Set independiente  $k=4$



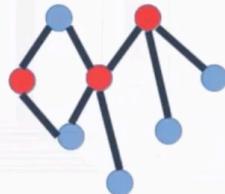
Set independiente  $k=5$



Cobertura de vértices  $k=7$



Cobertura de vértices  $k=4$



Cobertura de vértices  $k=3$

$\text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER}$  pero tambien

$\text{VERTEX-COVER} \leq_p \text{INDEPENDENT-SET}$

$\boxed{\text{VERTEX-COVER ES NP-C}}$

# Complejidad

## Set Cover

- $U$ : conjunto de  $n$  elementos
- Una colección  $S_1, \dots, S_m$  de subconjuntos de  $U$
- Queremos saber si existe una colección de  $k$  subconjuntos que sea igual a  $U$

### Ejemplo

$$U = \{a, b, c, d, e, f, g, h, i\}$$

$$S_1 = \{a, b, c, d\}$$

$$S_2 = \{a, b, f, i\}$$

$$S_3 = \{a, e, h, g\}$$

$$S_4 = \{b, c, g\}$$

$$S_5 = \{a, d, e\}$$

$$\nwarrow S_6 = \{g, h\}$$

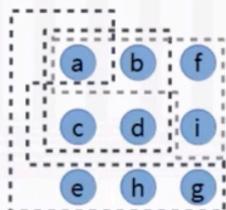
$$S_7 = \{e, i\}$$

$$S_8 = \{f\}$$

$$S_9 = \{i\}$$

Existe  $k=3$ ?

$$S_1 \cup S_2 \cup S_3 = U$$



# Complejidad

¿SET-COVER es NP?

- $U$ : conjunto de elementos
- $k$ : tamaño buscado
- $S_1, \dots, S_m$
- $T$ : certificado con subconjunto de conjuntos

Verificar  $\rightarrow |T| = k$  y para todo elemento en  $U$ , existe en algún subconjunto dentro de  $T$



Si se logra  $\text{SET-COVER} \in \text{NP}$

Reducción de VERTEX-COVER a SET-COVER

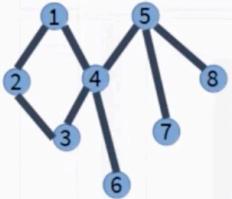
- Partimos de  $G = (V, E)$  y  $k$
- $U = E$
- Por cada  $v \in V$  crearemos un subconjunto  $S_v$  con todos los ejes incidentes a él
- Mantendremos en  $k$  la cantidad de subconjuntos a buscar

# Complejidad

Ejemplo

Sea el problema

Vertex-cover ( $k=3$ )



Reducimos a

Set Cover ( $k=3$ )

$$U = \{1-2, 1-4, 2-3, 3-4, 4-5, 4-6, 5-7, 5-8\}$$

$$S_1 = \{1-2, 1-4\}$$

$$S_2 = \{1-2, 2-3\}$$

$$S_3 = \{2-3, 3-4\}$$

$$S_4 = \{1-4, 3-4, 4-5, 4-6\}$$

$$S_5 = \{4-5, 5-7, 5-8\}$$

$$S_6 = \{4-6\}$$

$$S_7 = \{5-7\}$$

$$S_8 = \{5-8\}$$

SET-COVER  $\in$  NP - C

- Reducirnos vertex-cover a SET-COVER en tiempo polinomial
- Si resolvemos cualquier instancia de SET-COVER, podemos resolver cualquier para vertex-cover

# Complejidad

## Clique

- $G = (V, E)$  grafo no dirigido
- Clique: subconjunto de  $V' \subseteq V$  tal que para todo  $v, u \in V'$ , el eje  $(v, u) \in E$ . (es decir que todos los vértices están conectados entre sí)
- Dado un valor  $k$  queremos saber si existe un clique de tamaño  $k$

¿ Clique  $\in NP$  ?

- $G = (V, E)$
- $k$ : tamaño del clique
- $T$  certificado: subconjunto de nodos de  $V$

Puedo verificar (en tiempo polinomial) que la cantidad de nodos en  $T = k$

# Complejidad

¿Cliques  $\in$  P?

No existe un algoritmo polinomial que lo resuelva

¿Cliques  $\in$  NP-HARD?

- Usamos 3SAT  $\rightarrow$  3SAT  $\leq_p$  CLIQUES
  - Dada una instancia  $i$  de 3SAT con  $k$  cláusulas y  $n$  variables
  - Creamos un nodo por cada variable en una cláusula
  - Por cada par de variables de diferentes cláusulas creamos un eje
- Si no corresponden a la misma variable negada

$$E = (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_4)$$

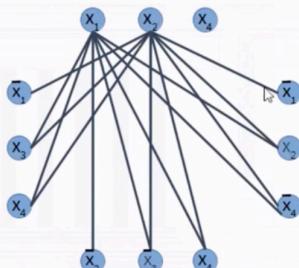
Con 4 variables y 4 cláusulas

Armo los nodos

para cada variable de cada cláusula

Agrego los ejes

según condición de construcción



# Complejidad

## Busco clique

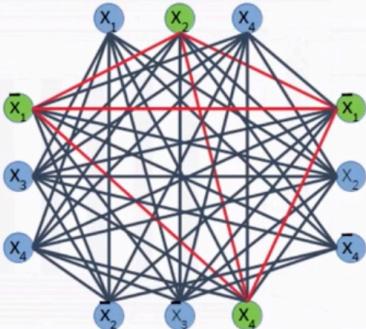
De tamaño  $k=4$

(para activar las 4 cláusulas)

## Los nodos dentro del clique

Indican el valor de las variables

(las variables que no están en el clique se pueden poner en true o en false)

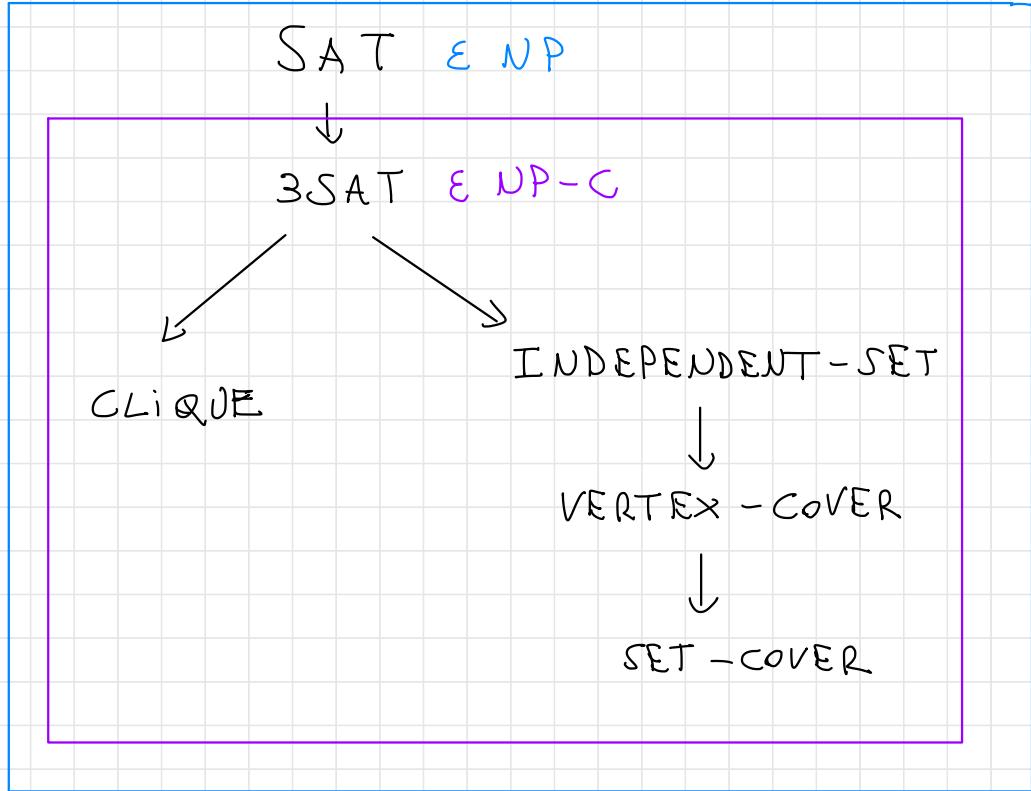


Como CLIQUE EN P y SAT  $\leq_p$  CLIQUE, entonces

CLIQUE ENP-C

# Complejidad

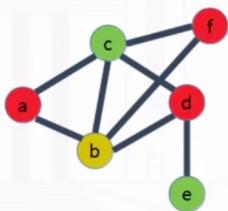
Resumen



# Complejidad

## Función de coloración

- Sea  $G = (V, E)$  grafo no dirigido
- $f: V \rightarrow \{1, 2, \dots, k\}$  función que asigna un color a cada vértice tal que  $\forall e = (u, v) \in E, f(u) \neq f(v)$

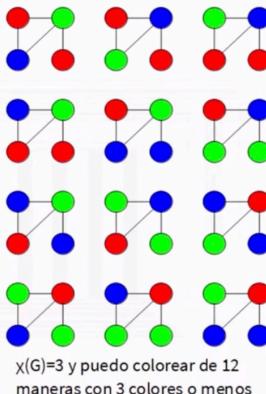


## Número cromático

- Es el menor número de colores  $\chi(G)$  necesarios para colorear el grafo

## Polinomio cromático

- Ecación que permite contar el número de maneras en las cuales puede ser coloreado un grafo usando no más de  $k$  colores



$\chi(G)=3$  y puedo colorear de 12 maneras con 3 colores o menos

# Complejidad

## Clase de color

- Subconjunto de vértices coloreados con el mismo color

## k-coloración

- Una partición de nodos en k conjuntos independientes
- Si un grafo es k-colorable entonces es k-partido

## Problema de decisión

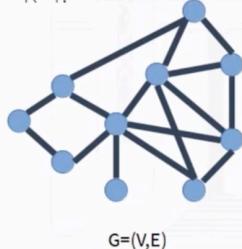
- Sea  $G = (V, E)$
- $k$ : valor numérico

¿ Es posible definir una función de coloración que utilice k o menor colores ?

Se puede colorear G con menos de:

K=3?

K=4?



# Complejidad

## Caso especial : Grafo completo

- Todos los vértices están comunicados entre si
- Se requieren  $|V|$  cantidad de colores

## Caso especial : Grafo bipartito

- Se requieren 2 colores
- Usando BFS coloreamos

¿ Es NP ?

- $G = (V, E)$
- $k$  colores
- T certificado : para todo vector que asigna a cada vértice un color

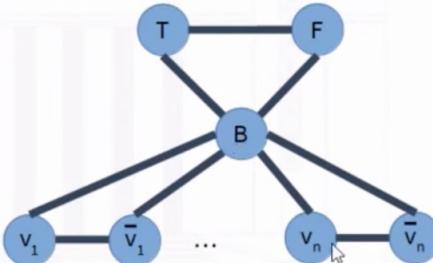
Puedo verificar (en polinomial)

- Todos los vértices de  $V$  están en  $T$
- Los colores usados son  $\leq k$
- No hay vértices adyacentes con el mismo color

# Complejidad

¿ Es NP-Hard ?

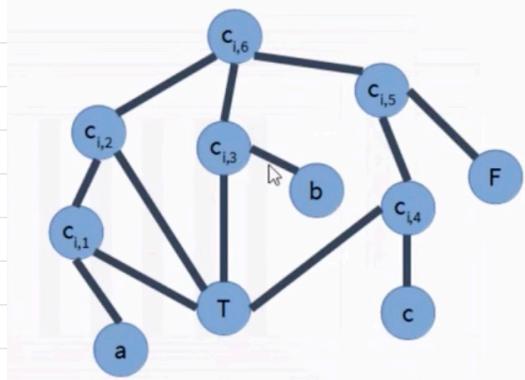
- Usamos 3SAT  $\leq_p$  COLORADO - GRAFO
- Dada una instancia  $i$  de 3SAT con  $k$  cláusulas y  $n$  variables
- Crear :
  - 1 gadget para las variables
  - 1 gadget por cláusula
    - ↑
    - grafo
- Definimos los nodos  $v_i$  y  $\bar{v}_i$  correspondientes a cada  $x_i$  y su negado  $\bar{x}_i$ , y los nodos especiales  $T$  (true),  $F$  (false) y  $B$  (bar)
- Unimos  $v_i$  y  $\bar{v}_i$  entre ellas y  $B$ , y  $T$  y  $F$  entre ellas y con  $B$



Si  $v_i$  es True tiene el mismo color que T

# Complejidad

- Creamos 1 gadget para cada cláusula
  - La cláusula  $c_i = (a, b, c)$  con  $a, b, c \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$
  - Las variables de la cláusula corresponden a los nodos creados para las variables

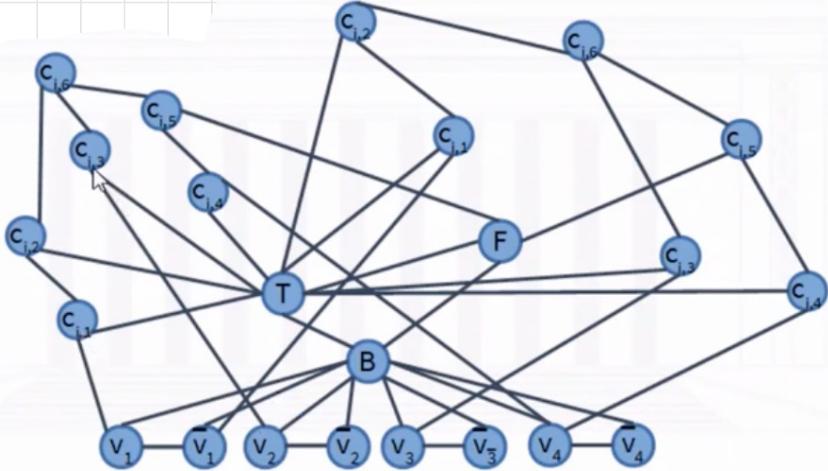
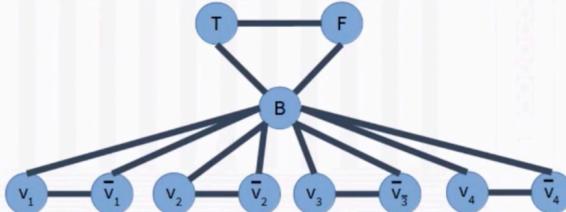


# Complejidad

Ejemplo

Tengo:  $E = (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4)$

- 4 variables
- 2 clausulas



# Complejidad

## 3 Dimensional Matching

- Dados 3 sets disjuntos  $X, Y, Z$  de tamaño  $n$  cada uno.
- Un set  $C \subseteq X, Y, Z$  de triples ordenadas que señalan relaciones entre los elementos de los distintos conjuntos
- Determinar si existe un subset de  $n$  triples en  $C$  tal que cada elemento  $x \in X \cup Y \cup Z$  sea contenido exactamente en una de esas triples

Ejemplo : asignar un chofer, auto y pasajeros según preferencias

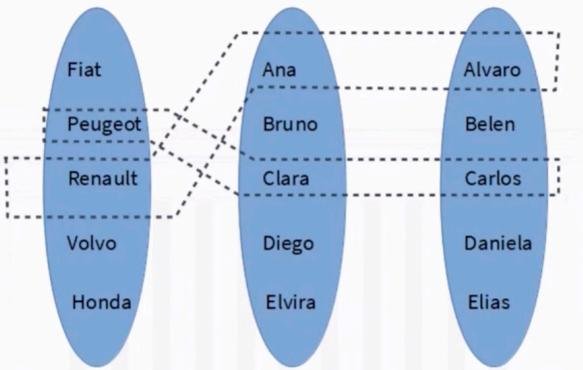
Autos={Fiat, Peugeot, Renault, Volvo, Honda}

Choferes={Ana, Bruno, Clara, Diego, Elvira}

Pasajeros={Alvaro, Belen, Carlos, Daniela, Elias}

Posibles equipos = { (Fiat,Ana,Belen), (Fiat,Bruno,Daniela),  
(Peugeot,Clara,Carlos), (Peugeot,Diego,Elias), (Peugeot,Elvira,Alvaro),  
(Renault,Bruno,Daniela), (Renault,Ana,Alvaro), (Renault,Clara,Elias),  
(Volvo,Diego,Elias), (Honda,Clara,Carlos), (Honda,Clara,Daniela),  
(Honda,Diego,Alvaro) }

# Complejidad



Existe una posible asignación donde todos quedan con pareja?

Si! pero como lo encuentro?

## 3DM : Variante de 2DM

- Existe un algoritmo polinomial que lo resuelve (se puede reducir a un problema de redes de flujo).

## ¿ 3DM es NP ?

- Si! En la clase dan un ejemplo que lo resuelve, es decir, un T certificado
- Podemos certificar en tiempo polinomial que  $|T|$  es igual a  $n$ .

# Complejidad

¿ 3DM es NP-HARD? ?

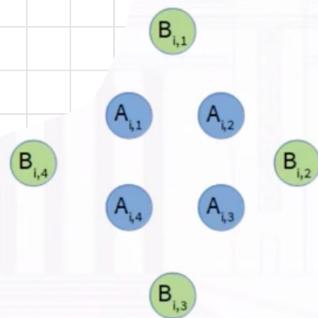
- Probaremos que  $3SAT \leq_p 3DM$
- Sea  $i$  instancia de  $3SAT$  con  $n$  variables  $= \{x_1, \dots, x_n\}$  y  $k$  clausulas  $= \{c_1, \dots, c_k\}$ , reducir en tiempo polinomial

Reducción de 3SAT a 3DM

- Por cada variable  $x_i$  creamos un gadget con los elementos:

$$A_i = \{\alpha_{i,1}, \alpha_{i,2}, \dots, \alpha_{i,2k}\} \rightarrow \text{núcleo}$$

$$B_i = \{\beta_{i,1}, \beta_{i,2}, \dots, \beta_{i,2k}\} \rightarrow \text{puntas}$$



Variable  $i$ : ejemplo para  $k=2$  clausulas

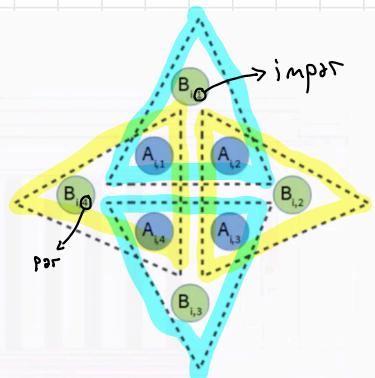
# Complejidad

- También por cada variable  $x_i$  creamos triples:

$$t_{i,j} = \{ \exists_{i,j}, \exists_{i,j+1}, \exists_{i,j} \}$$

● Tripla par  $\rightarrow$  si  $j$  es par

● Tripla impar  $\rightarrow$  si  $j$  es impar



- Por cada cláusula  $c_j$  creamos un set de elementos núcleo:

$$c_j = \{ p_j, p_j' \}$$

- Por cada variable  $x_i$  en la cláusula  $c_j$

- Si contiene la variable  $\bar{x}_i$

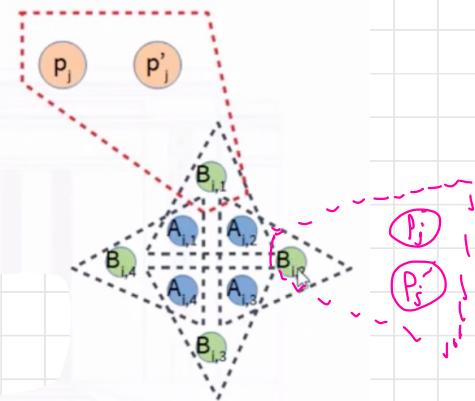
$\hookrightarrow$  creamos una tripla  $(p_j, p_j', b_i, 2_{j-1})$

- Si contiene la variable  $x_i$

$\hookrightarrow$  creamos una tripla  $(p_j, p_j', b_i, 2_j)$

$\Rightarrow$  Cada cláusula tendrá 3 triples

# Complejidad



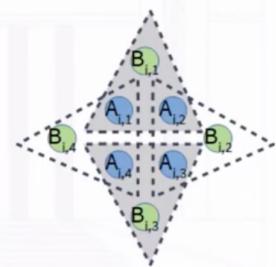
Clausula 1: con la variable  $\bar{x}_i \rightarrow$  Como  $\bar{x}_i$ , entonces agarro la punta donde está el elemento  $p_i$

- Si hubiese fijo  $x_i$  si negar hubiese agarrado  $B_{i,2}$

¿Como funciona?!

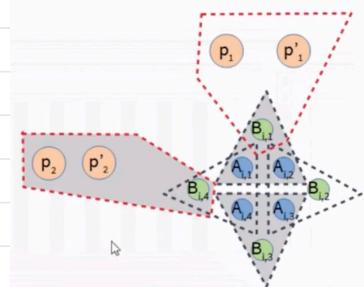
- Si una variable  $i$  en la solución está en 1 ( $x_i = 1$ )

Ls Las puntas del gadget  $i$  correspondientes a su valor 0 estarán cubiertas por los triplas de su núcleo



# Complejidad

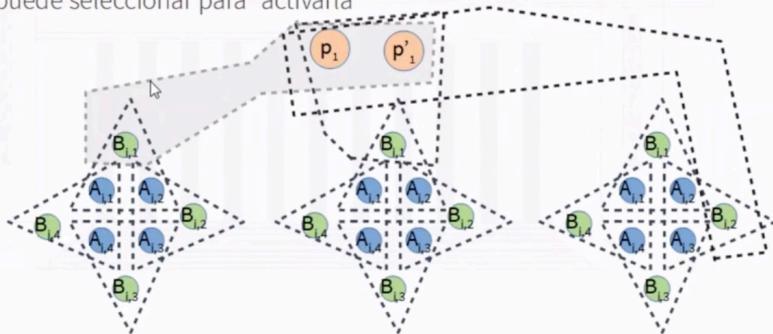
↳ Los puntos corren ponderados a su valor 1 pueden usarse para activar cláusulas



## Cada cláusula

Tiene 3 triples asociadas

Solo 1 se puede seleccionar para "activarla"



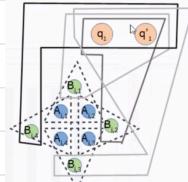
• En total tenemos  $2 \cdot n \cdot k$  puntos

- Las triples de las cláusulas cubren  $k$  de ellos
- Las triples de los gadgets cubren  $n \cdot k$  puntos
- Faltan cubrir  $(n-1) \cdot k$  puntos

# Complejidad

## Cleanup gadgets

- Construimos  $(n-1) \cdot k$  gadgets
- Cada uno tendrá  $Q_i = \{q_i, q_i'\}$
- Agruparemos las triples  $\{q_i, q_i', b\}$  con  $b$  son todas las puertas de los gadgets



Ejemplo de cleanup gadget  
(solo mostrando las triples  
en 1 gadget de variable)

## Terminando...

- Para terminar, necesitamos convertir los conjuntos  $X, Y, Z$

### Conjunto X

$a_{ij}$  con  $j$  par (de los widgets variables)  
 $\rightarrow nk$

$p_j$  (de los widget clausula)  $\rightarrow k$

$q_j$  (de los widget cleanups)  $\rightarrow (n-1)k$

### Conjunto Z

Todos los  $b_{ij}$  (de los widgets variables)  $\rightarrow 2nk$

### Conjunto Y

$a_{ij}$  con  $j$  impar (de los widgets variables)  $\rightarrow nk$

$p'_j$  (de los widget clausula)  $\rightarrow k$

$q'_j$  (de los widget cleanups)  $\rightarrow (n-1)k$

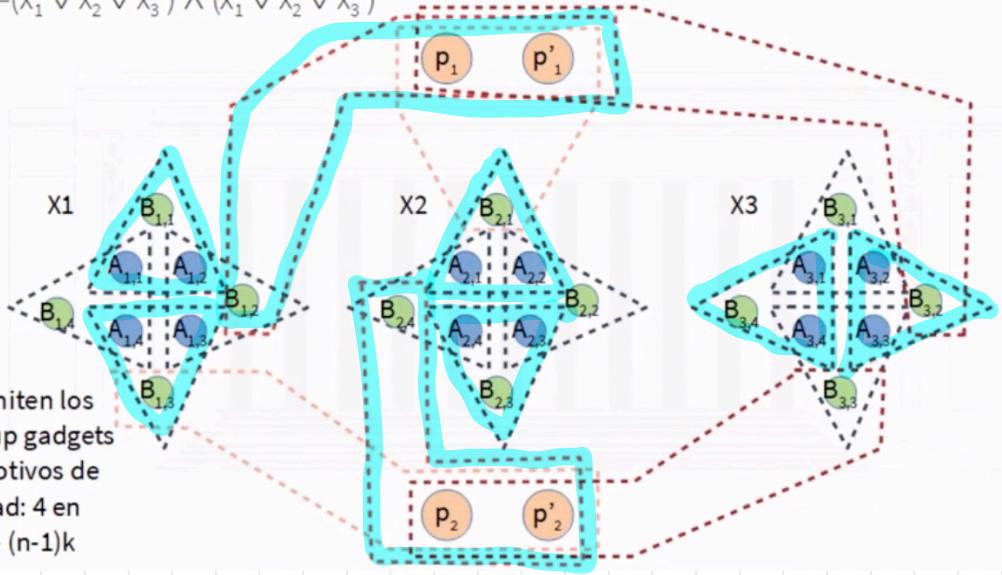
### Triplas "C"

serán todas las triples definidas

# Complejidad

## Ejemplo

$$E = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$



possible solución  $\rightarrow x_1=1, x_2=1, x_3=0$

- Con esto la reducción está hecha  $\rightarrow$  3DM es NP-C

# Complejidad

## Subset Sum

- Sea un conjunto de números naturales  $C = \{w_1, w_2, \dots, w_n\}$
- Determinar si existe un subconjunto de  $C$  que suma exactamente  $W$

Está relacionado con el problema de la mochila

¿ SS es P? ?

- Utilizando programación dinámica se puede resolver en tiempo pseudo-polinomial  $\mathcal{O}(Wn)$
- Si representamos  $W$  en bits, el algoritmo crece exponencialmente a la cantidad de bits de  $W$

¿ Existe una polinomial?  $\rightarrow$  No se conoce

# Complejidad

¿SS es NP?

- $C$ : conjunto de  $n$  naturales
- $W$ : número a sumar
- $T$  certificado con subconjunto de  $C$

Podemos verificar polynomialmente

$$\sum_{i \in T} t_i = W \quad \rightarrow \quad \text{todo } t_i \in C$$

→ SUBSET-SUM ∈ NP

# Complejidad

¿SS es NP-HARD?

- Utilizaremos 3DM  $\rightarrow$   $3DM \leq_p SUBSET-SUM$

Reducción 3DM a SUBSET-SUM

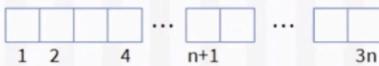
- Si instancia de 3DM, con  $X, Y, Z$  conjuntos de  $n$  elementos
- Podemos representar cada ejemplo como un vector de bits

**Utilizaremos vectores de  $3^n$  bits**

Los primeros  $n$  bits representan los elementos del conjunto  $X$

Los siguientes  $n$  bits representan los elementos del conjunto  $Y$

Los últimos  $n$  bits representan los elementos del conjunto  $Z$



**A cada elemento de los conjuntos X (y similarmente para Y, Z)**

Les asignaremos un orden arbitrario de 1 a  $n$

Llamaremos  $pos_x(x)$ ,  $pos_y(y)$ ,  $pos_z(z)$  a las funciones que dado el elemento nos retorna su posición en el set

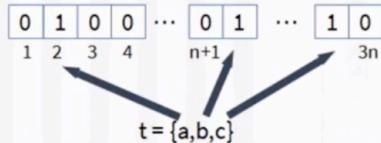
# Complejidad

A cada tripleta de T

$$t = \{x_i, y_j, z_k\}$$

La representaremos como un vector  $V_t$  de  $3n$  bits

Con todos los bits en cero



Pondremos los siguientes bits en 1

$$\text{pos}_x(x_i)$$

$$\text{pos}_y(y_j) + n$$

$$\text{pos}_z(z_k) + 2n$$

Cada vector  $v_t$

Se puede interpretar como un numero  
 $w_t$  en el subconjunto de subset Sum

El Valor W lo formaremos como

el vector con todos los bits en 1

Para sumar W

Tenemos que encontrar aquellos  $w_t$  que sumados den W

$$\begin{array}{r} 0 \ 1 \ 0 \ \dots \ 1 \ 0 \ 0 \ \dots \ 0 \ 1 \ 0 \\ 0 \ 1 \ 0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 1 \ 0 \ 0 \\ + \vdots \qquad \vdots \\ 1 \ 0 \ 0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0 \ 0 \ 1 \\ \vdots \qquad \vdots \\ 0 \ 0 \ 1 \ \dots \ 0 \ 0 \ 1 \ \dots \ 1 \ 0 \ 0 \\ \hline W = 1 \ 1 \ 1 \ \dots \ 1 \ 1 \ 1 \ \dots \ 1 \ 1 \ 1 \end{array}$$

Pero! puedo tener un problema de overflow.

Para evitarlo podemos redefinir como expresar  $w_t$  en base al vector

# Complejidad

## Transformación en un número entero

### Seleccionaremos una base d

Y representaremos cada tripla como un número de la manera:

$$w_t = \sum_{i=1}^{3n} v_t[i] * d^{i-1}$$

Como todos los elementos del vectores estarán en 0 excepto 3 se puede ver como:

$$w_t = d^{pos_s(x_i)-1} + d^{pos_s(y_i)+n-1} + d^{pos_s(z_i)+2n-1}$$

### Para la base d

Utilizaremos un valor mayor a m (cantidad de triples)  $\rightarrow d = m+1$

(con eso, aunque las m triples contengan un mismo elemento no será posible el overflow)

### Ejemplo

$$X = \{x_1, x_2, x_3\}$$

$$Y = \{y_1, y_2, y_3\}$$

$$Z = \{z_1, z_2, z_3\}$$

$$T = \{(x_1, y_1, z_1), (x_2, y_2, z_3), (x_3, y_3, z_1), (x_1, y_1, z_2), (x_3, y_3, z_2), (x_1, y_3, z_2), (x_3, y_3, z_3)\}$$

t	v <sub>t</sub>		w <sub>t</sub>
(x <sub>1</sub> , y <sub>1</sub> , z <sub>1</sub> )	001001001	262144+512+1	262657
(x <sub>2</sub> , y <sub>2</sub> , z <sub>3</sub> )	100010010	16777216+4096+8	16781320
(x <sub>3</sub> , y <sub>3</sub> , z <sub>1</sub> )	001100100	262144+32768+64	294976
(x <sub>1</sub> , y <sub>1</sub> , z <sub>2</sub> )	010001001	2097152+512+1	2097665
(x <sub>3</sub> , y <sub>3</sub> , z <sub>2</sub> )	010100100	2097152+32768+64	2129984
(x <sub>1</sub> , y <sub>3</sub> , z <sub>2</sub> )	010100001	2097152+32768+1	2129921
(x <sub>3</sub> , y <sub>3</sub> , z <sub>3</sub> )	100100100	16777216+32768+64	16810048
w	11111111		19173961

$$n = 3, m = 7$$

$$d = 7 + 1 = 8$$

$$3 \cdot n = 9$$

Ahora nuestro problema se reduce a ver si hay una suma de **tres** que me dé un valor de **8** W

$$\begin{array}{r} 262657 \\ + 16781320 \\ 2129984 \\ \hline 19173961 \end{array}$$

# Complejidad

SUBSET-SUM es NP-C

- Como SUBSET-SUM  $\in$  NP y acabamos de mostrar que  $3DM \leq_p$  SUBSET-SUM

# Complejidad

## Ciclo Hamiltoniano

- Sea  $G = (V, E)$  un grafo dirigido
- Un ciclo  $C$  en  $G$  es hamiltoniano si visita cada vértice 1 vez, comenzando y terminando en el mismo vértice
- Queremos encontrar el ciclo

¿ HAM-CYCLE es NP ?

- T certificado :  $\{t_0, \dots, t_{|V|}\}$  secuencia ordenada de vértices

Puedo verificar en tiempo polinomial que

- $|T| = |V|$
- Todos los vértices de  $V$  están en  $T$
- $\forall t_i, t_{i+1} \in T, (t_i, t_{i+1}) \in E$

HAM-CYCLE  $\in$  NP

# Complejidad

¿ HAM-CYCLE es P?

- No se conoce un algoritmo...
- Si probamos que  $3SAT \leq_p HAM-CYCLE$  entonces  $HAM-CYCLE \in NP-C \rightarrow HAM-CYCLE \in P \Leftrightarrow P=NP$

Reducción de 3SAT a HAM-CYCLE

- Si instanciamos 3SAT con  $n$  variables y  $k$  cláusulas
- Construimos un grafo

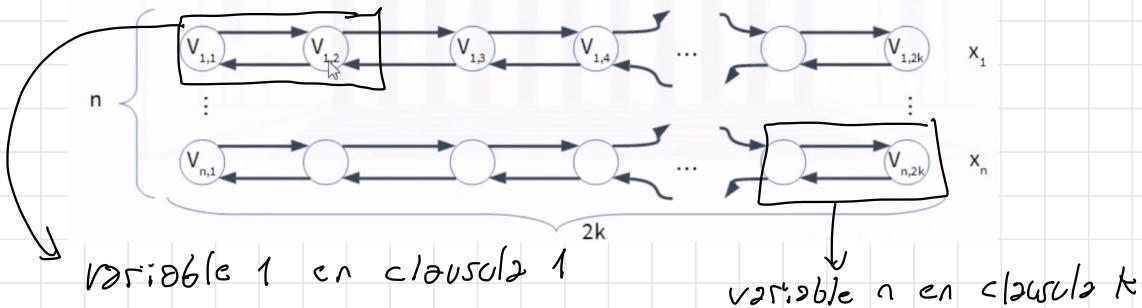
**Construimos  $n$  caminos  $p_1, \dots, p_n$**

cada uno representa a una variable

**Cada camino estará conformado por  $2^k$  nodos**

Unidos entre sí por aristas de ida y vuelta

(cada 2 nodos corresponden a la variable en una cláusula)



# Complejidad

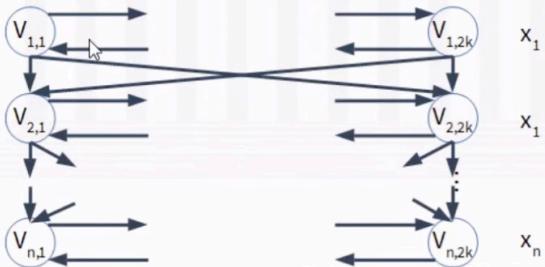
## Uniremos cada camino

Desde su nodo inicial hasta el nodo inicial del camino siguiente

Desde su nodo inicial hasta el nodo final del camino siguiente

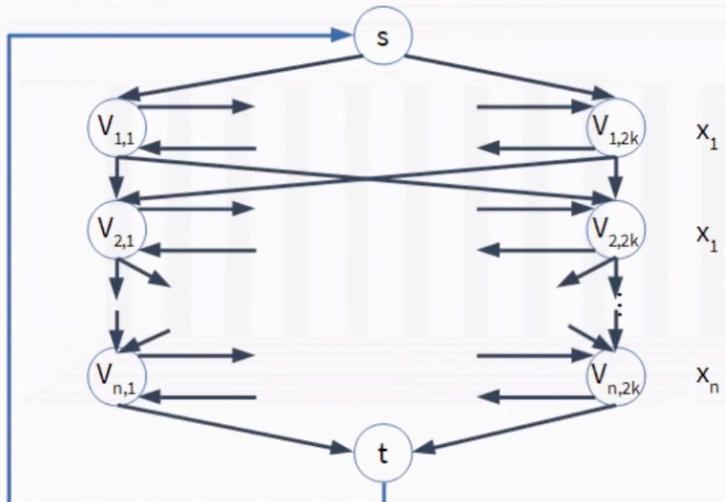
Desde su nodo final hasta el nodo inicial del camino siguiente

Desde su nodo final hasta el nodo final del camino siguiente



## Creamos nodos: s y t

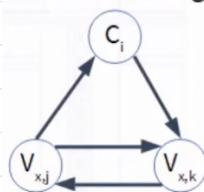
Unimos con los ejes  $s-V_{1,1}$ ,  $s-V_{1,2k}$ ,  $V_{n,2k}-t$ ,  $V_{n,2k}-t$ ,  $t-s$



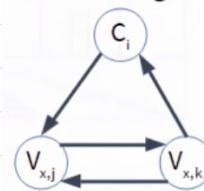
# Complejidad

- Para cada cláusula, vamos a crear 1 nodo, con 3 pasos de ejes
- Para cada variable  $x$  de las cláusulas  $i$  se va a unir con 2 nodos del camino de la variable  $x$  con el nodo de la cláusula  $i$  en la posición  $j \in i.2$  y  $k \in i.2 + 1$
- Para que exista el camino hamiltoniano tienen que poder satisfacerse la expresión booleana

Variable sin negar



Variable negada



# Complejidad

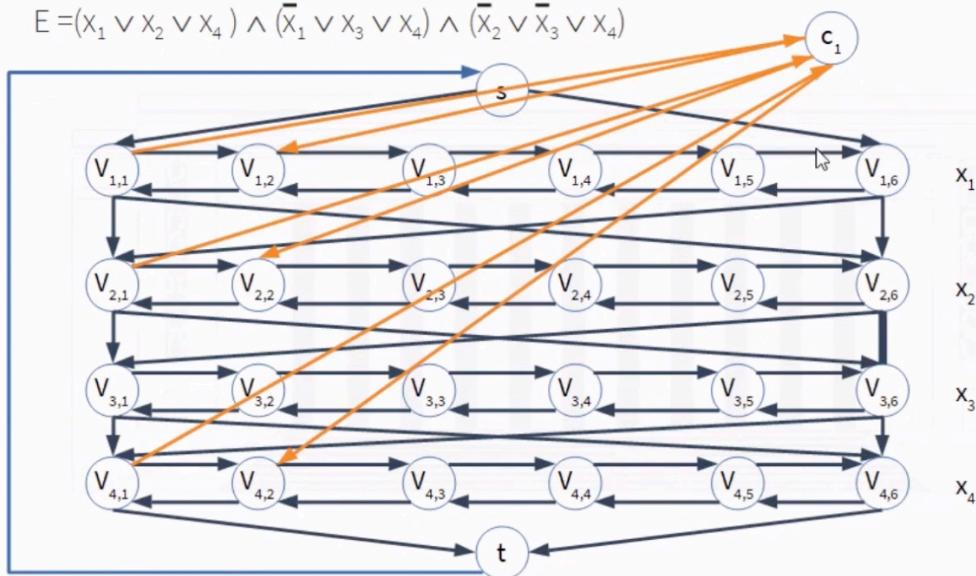
Ejemplo

$$E = (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4)$$

$$n = 4$$

3. 2 = 6 nodos por camino

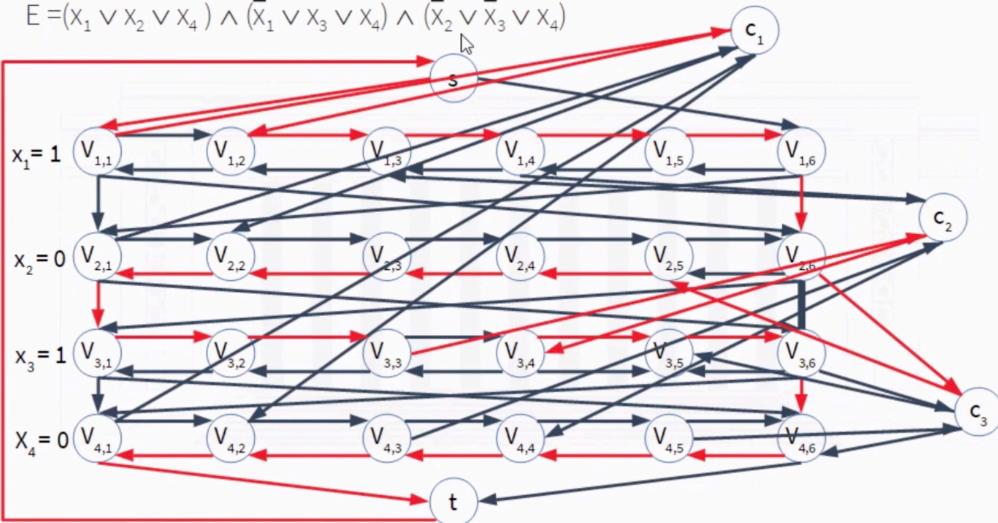
3 nodos clausula



Falta agregar los otros dos clausulas

# Complejidad

$$E = (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4)$$



HAM-CYCLE ∈ NP-C

# Complejidad

## Viajante de comercio

Un viajante debe

Recorrer  $n$  ciudades  $v_1, v_2, \dots, v_n$

Partiendo de  $v_1$  se debe construir un tour visitando cada ciudad una vez y retornar a la ciudad inicial

Para cada par de ciudades  $v_x, v_y$

Se especifica una distancia  $d(v_x, v_y)$

No necesariamente hay simetría:  $d(v_x, v_y)$  puede ser diferente a  $d(v_y, v_x)$

No necesariamente se cumple la desigualdad triangular:  $d(v_i, v_j) + d(v_j, v_k) > d(v_i, v_k)$

¿ Es NP?

• T certificado = tour de ciudades

•  $k$  distancias como límite

Se puede verificar en tiempo polinomial

- T contiene todas las ciudades solo 1 vez
- T comienza y termina con la misma
- La suma es menor a  $k$

# Complejidad

¿Es P?

- No se conoce

Reducción HAM-CYCLE

Por cada

Vértice  $v_i \in V \rightarrow$  creamos una ciudad  $v'_i$

Arista  $e_{i,j} \in E \rightarrow$  definiremos la distancia  $d(v'_i, v'_j) = 1$

Aquellas distancias que no están definidas (no tienen aristas) las crearemos con valor 2

Ponemos como valor  $k = |V|$  (numero de vértices).



**Solucionamos viajante con k definido**

Si existe camino con longitud k, entonces existe ciclo hamiltoneano.

# Programación Lineal

## Introducción

- Dado :
  - un conjunto de " $n$ " variables de decisión de valor real
  - " $m$ " restricciones expresadas como inequaciones lineales utilizando las variables de decisión
  - una función lineal utilizando las variables de decisión cuyo valor debe ser evaluado
- Queremos :
  - Encontrar una asignación que cumpla las restricciones
  - Maximizar el resultado

# Programación Lineal

## Ejemplo:

Una empresa informática brinda dos tipos de paquetes de consultoría para sus clientes. El primero llamado "paquete 1" corresponde a asignar por el lapso de un año a 4 desarrolladores backend y 2 desarrolladores frontend. El segundo, "paquete 2" asigna 1 desarrollador backend y 3 desarrolladores frontend por el mismo intervalo de tiempo. Actualmente cuenta con 30 desarrolladores backend y 20 desarrolladores frontend. Por convenio con una universidad local por cada "paquete 2" deben asignar 4 pasantes. La cantidad máxima de pasantes disponibles es 12. Sin embargo por cada contrato "paquete 1" que realizan pueden adicionar un pasante a la disponibilidad. El "paquete 1" brinda un beneficio económico de 900 por mes y el "paquete 2" 400 por mes. Desean conocer cómo podrían maximizar sus ganancias con sus recursos disponibles.

Variabiles de decisión  $\rightarrow x_1 = 2$        $x_1$ : paquete 1  
 $x_2 = 2$        $x_2$ : paquete 2

Función objetivo  $\rightarrow \sum_{i=1}^n c_i \cdot x_i$   
 $Z = 900 \cdot x_1 + 400 \cdot x_2$

Vales objetivo  $\rightarrow$  si  $x_1 = 1$  y  $x_2 = 2 \rightarrow Z = 1300$

Restricción lineal  $\rightarrow \sum_{i=1}^n a_{ij} \cdot x_i \{ \leq, =, \geq \} b_j$  para  $1 \leq j \leq m$

$$4x_1 + 1x_2 \leq 30 \quad (\text{backend})$$

$$2x_1 + 3x_2 \leq 20 \quad (\text{frontend})$$

$$-1x_1 + 4x_2 \leq 12 \quad (\text{pasantes})$$

# Programación Lineal

No negatividad  $\rightarrow x_1 \geq 0, x_2 \geq 0$

## Soluciones

- Podemos representar una solución de un problema de  $n$  variables como un vector de " $n$ " posiciones  $(x_1, \dots, x_n) = (v_1, \dots, v_n)$
- Factible: cumple con las restricciones
- Óptima: maximiza (o minimiza) el resultado

$$(x_1, x_2) = (1, 1)$$

Solución  
factible  $\rightarrow$

$$4x_1 + 1x_2 = 4 \cdot 1 + 1 \cdot 1 = 5 \leq 30 \text{ (d. backend)} \quad \checkmark$$

$$2x_1 + 3x_2 = 2 \cdot 1 + 3 \cdot 1 = 5 \leq 20 \text{ (d. front end)} \quad \checkmark$$

$$-1x_1 + 4x_2 = -1 \cdot 1 + 4 \cdot 1 = 3 \leq 12 \text{ (pasantes)} \quad \checkmark$$

$$z = 1300$$

# Programación Lineal

## Interpretación geométrica

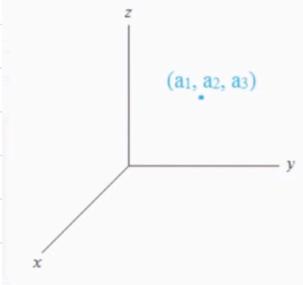
### Espacio de soluciones

- Cada solución es un vector de  $n$  posiciones:

$$(x_1, x_2, \dots, x_n)$$

Por ejemplo:  $(a_1, a_2, a_3)$ ,  $(b_1, b_2, b_3)$ , etc

- Podemos pensar cada punto como perteneciente a un espacio " $n$ " dimensional

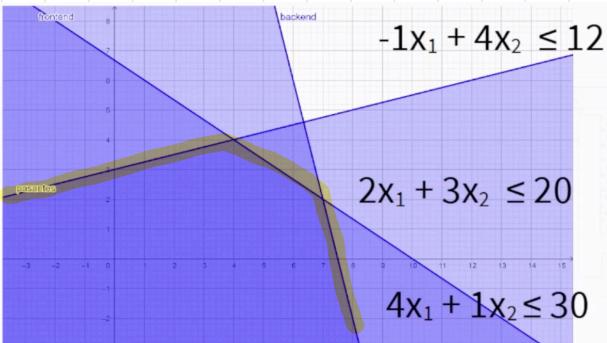


# Programación Lineal

## Polítopo Convexo

- Cada restricción es un hiperplano
- Queremos considerar la intersección de todos los restricciones
  - Si el problema es limitado, esa región es un polítopo convexo

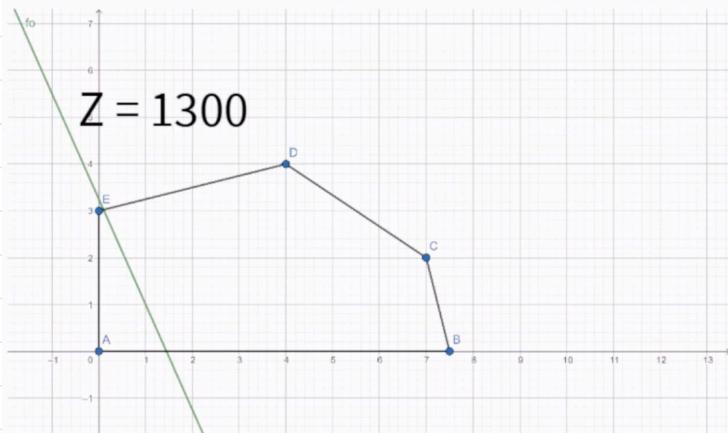
## Ejemplo



# Programación Lineal

Agregamos la función objetivo

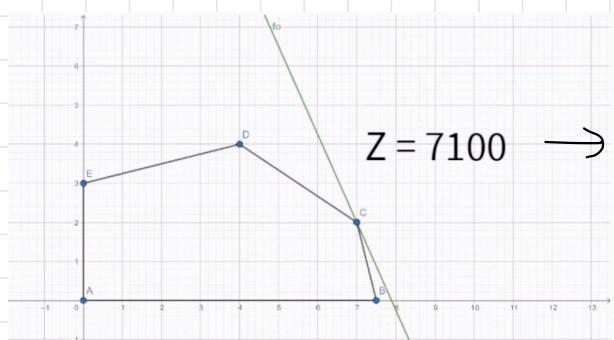
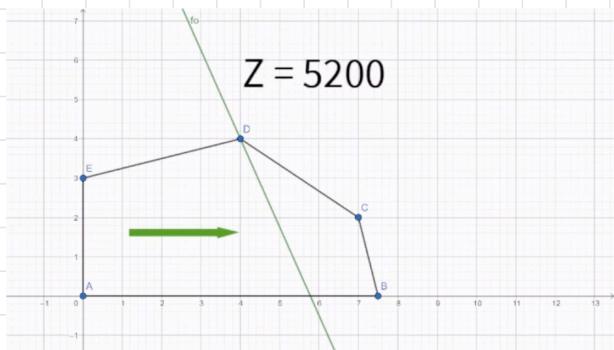
- Veremos que un hiperplano identifica todos los puntos con el mismo valor objetivo
- La intersección determina el conjunto de soluciones factibles con ese valor objetivo
- Un punto del conjunto debe corresponder con la frontera del polítopo



# Programación Lineal

## Búsqueda por deslizamiento

- En algún momento no habrá puntos del polítopo que intersecten con el hiperplano
- Un instante antes se va a cruzar con el último punto de la frontera
- Esta será la solución OPTIMA



# Programación Lineal

## Puntos extremos

. Podemos estimar el límite superior de puntos extremos como

$$\binom{n+m}{n} \approx \frac{(n+m)!}{n!m!}$$

# Programación Lineal

## Forma Estándar y de holgura

### Problemas equivalentes

Para cada factible  $s_1$  de  $P_1$  con valor objetivo  $z$

$\uparrow \downarrow$  existe una

Para cada factible  $s_2$  de  $P_2$  con valor objetivo  $z$

Para cada factible  $s_1$  de  $P_1$  con valor objetivo  $(-z)$

$\uparrow \downarrow$  existe una

Para cada factible  $s_2$  de  $P_2$  con valor objetivo  $-z$

# Programación Lineal

## Forma Estándar

- La función objetivo se debe maximizar
- Todas las restricciones deben ser menor o igual ( $\leq$ )
- Todas las variables deben ser positivas ( $x_i \geq 0$ )

## Transformación en problema estandar

Minimizar :

$$Z = -2x_1 + 4x_2 - x_3$$

Sujeto a :

$$x_1 - x_2 - x_3 \leq 24$$

$$x_2 + x_3 = 0$$

$$-2x_1 + 4x_2 \geq 8$$

$$x_1, x_2 \geq 0$$

# Programación Lineal

## Transformación: Igualdad

$$x_2 + x_3 = 10 \Leftrightarrow \begin{array}{l} x_2 + x_3 \leq 10 \\ x_2 + x_3 \geq 10 \end{array}$$

## Transformación: Mayor o igual

- Negamos ambos lados para que sea con  $\leq$

$$-2x_1 + 4x_2 \geq 8 \rightarrow 2x_1 - 4x_2 \leq -8$$

$$x_2 + x_3 \geq 10 \rightarrow -x_2 - x_3 \leq -10$$

## Transformación: No negatividad

- Si una variable  $x_i$  no tiene esa restricción / se reemplaza por  $x_i'$  con  $x_i' \geq 0$  y  $x_i''$  con  $x_i'' \geq 0$
- Reemplazamos  $x_i$  por  $x_i' - x_i''$

Como le faltó a  $x_3$ , agrego  $x_3'$  y  $x_3''$

$$4x_1 - x_2 - x_3' + x_3'' \leq 24$$

$$x_2 + x_3' - x_3'' \leq 10$$

$$-x_2 - x_3' + x_3'' \leq -10$$

# Programación Lineal

Transformación: Minimización a maximización

$$z = \sum c_i \cdot x_i$$

↓

$$-z = -\sum c_i \cdot x_i$$

$$z = -2x_1 + 4x_2 - x_3$$

↓

$$-z = 2x_1 - 4x_2 + x_3$$

# Programación Lineal

## Ejemplo completo

Maximizar

$$(-z) = 2x_1 - 4x_2 + x_3' - x_3''$$

Sujeto a:

$$4x_1 - x_2 - x_3' + x_3'' \leq 24$$

$$x_2 + x_3 - x_3'' \leq 10$$

$$-x_2 - x_3 + x_3'' \leq -10$$

$$2x_1 - 4x_2 \leq -8$$

$$x_1, x_2, x_3', x_3'' \geq 0$$

(6, 8, 4, 2) es factible

## Reconstrucción

$x_1$  y  $x_2$  son iguales

$$\rightarrow (6, 8, 2)$$

$$x_3 \text{ es } (x_3' - x_3'') = 2$$

# Programación Lineal

## Forma de holgura

- Para cada restricción  $j \sum_{i=1}^n a_{ij} \cdot x_i \leq b_j$
- Agregaremos una nueva variable Slack  $x_{n+j} \geq 0$
- $x_{n+j} = b_j - \sum_{i=1}^n a_{ji} \cdot x_i$

## Ejemplo

Maximizar

$$z = 900 x_1 + 400 x_2$$

Sujeto a

$$4x_1 + 1x_2 \leq 30 \text{ (d. backend)}$$

$$2x_1 + 3x_2 \leq 20 \text{ (d. front end)}$$

$$-1x_1 + 4x_2 \leq 12 \text{ (pasantes)}$$

$$X_1 \geq 0, X_2 \geq 0$$



Maximizar

$$z = 900 x_1 + 400 x_2$$

Sujeto a

$$X_3 = 30 - 4x_1 - 1x_2 \text{ (d. backend)}$$

$$X_4 = 20 - 2x_1 - 3x_2 \text{ (d. front end)}$$

$$X_5 = 12 + 1x_1 - 4x_2 \text{ (pasantes)}$$

$$X_1 \geq 0, X_2 \geq 0, X_3 \geq 0, X_4 \geq 0, X_5 \geq 0$$

# Programación Lineal

## Simplex

- Requiere estar en forma de holgura
- Pasar:
  - 1 - Seleccionamos un punto extremo del polítopo inicial
  - 2 - Reemplazamos el punto por otro extremo adyacente que aumenta la función objetivo
  - 3 - Repetimos el paso 2 hasta que sea imposible aumentar
  - 4 - Retomamos como solución el último punto encontrado

### Variabiles básicas > simples

- Básicas: se encuentran multiplicadas por coeficientes
- Libres: en el primer término de las igualdades de las restricciones

$$x_{n+j} = b_j - \sum_{i=1}^n a_{ji} \cdot x_i \text{ para } 1 \leq j \leq m$$

Variables libres                          Variables básicas

- Van a ir cambiando

# Programación Lineal

Paso 1: Selección de punto extremo inicial

$$( \underbrace{x_1, \dots, x_n}_{\text{básicas}}, \underbrace{x_{n+1}, \dots, x_{n+m}}_{\text{libres}} ) \rightarrow \text{Vector}$$

- Asignamos a todas las variables de decisión en cero
- Las variables libres toman como valor los  $b_i$  correspondientes a la ecuación de restricción en la que están.

$$(0, \dots, 0, b_1, \dots, b_m)$$

Paso 2: Selección de variable a aumentar

- Seleccionar una variable básica que en la función objetivo esté multiplicada por un coeficiente positivo
- Si hay más de una agarrar el de mayor coeficiente, y si empiezan el que esté más a la izquierda
- Mantenemos el resto de las variables fijas (en cero)

# Programación Lineal

## Problema ilimitado

- Si una variable tiene un coeficiente positivo y no tiene restricciones, puede ser aumentada infinitamente
- No hay un máximo en este caso

## Variable ajustada

- Cuando una restricción impide que sigamos aumentando la variable

$$x_{n+j} = b_j - \sum_{i=1}^n a_{ji} \cdot x_i \quad x_i \geq 0 \quad x_{n+j} \geq 0$$

$$x_{n+j} = b_j - a_{ji} \cdot x_i = 0 \Rightarrow x_i = \underbrace{b_j / a_{ji}}_{\text{lo máximo que se puede aumentar}}$$

- Este es un nuevo punto extremo

# Programación Lineal

Pivot

- Pero continuas tenemos que hacer un pivot
    - Intercambiaremos la variable aumentada y la libre ajustada
  - 
  - ¿Cómo se aplica?
    - Se debe despejar la saliente en la ecuación de restricción que aparece y reemplazar el resultado en el resto de las restricciones función objetivo.

# Programación Lineal

Passo 3 : Repetir el passo 2!

- Hasta que no se pueda aumentar
- . No se puede aumentar cuando alguna de las variables en la función objetivo está multiplicada por un coeficiente negativo

Solución óptima

- Tenemos que recolectar el valor de las variables en la última ejecución

Representar el problema de forma Slack

Seleccionar como solución inicial a la solución básica

Repetir mientras existe un coeficiente positivo en la función objetivo

    Seleccionar una variable de salida "vs" entre las variables básicas con coeficiente positivo en la función objetivo

Determinar la variable de entrada "ve" cuya restricción es la que ajusta a "vs"

Si la variable "vs" no se ajusta

    Retornar 'solución ilimitada'

Realizar el proceso de pivot entre "vs" y "ve"

Retornar valor de las variables de decisión y valor de función objetivo

# Programación Lineal

## Complejidad

- Asumimos que viene en forma estándar

- En el peor de los casos tenemos que transformar las  $n$  variables en  $m$  restricciones  $\mathcal{O}(m.n)$  (temp)  $\rightarrow$  Transformación a slack
- Agregamos  $m$  variables  $\mathcal{O}(m)$  (esp)
- Llevamos un registro de variables libres y básicas que es  $\mathcal{O}(m+n)$  (esp)
- Matriz de coeficientes  $\rightarrow \mathcal{O}(n.m)$  (esp)
- Obtener coeficiente pivote y variable saliente requiere  $\mathcal{O}(n)$  (temp) y  $\mathcal{O}(1)$  (esp)
- Ajustar las variables en  $m$  restricciones  $\rightarrow \mathcal{O}(m)$  (temp)
- pivot  $\rightarrow \mathcal{O}(n.m)$

Hay  $\binom{n+m}{n}$  todos extremos  $\rightarrow$  No es polinomial

# Programación Lineal

Ejemplo:

Maximizar

$$z = 900x_1 + 400x_2$$

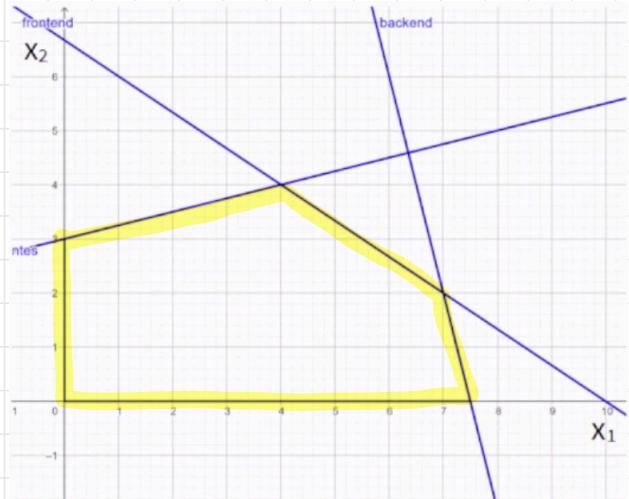
Sujeto a

$$4x_1 + 1x_2 \leq 30 \text{ (d. backend)}$$

$$2x_1 + 3x_2 \leq 20 \text{ (d. front end)}$$

$$-1x_1 + 4x_2 \leq 12 \text{ (pasantes)}$$

$$x_1 \geq 0, x_2 \geq 0$$



Solución:  $(\underbrace{x_1, x_2}_{6 \text{ sijos}}, \underbrace{x_3, x_4, x_5}_{1 \text{ libres (3 restricciones)})}$

Holgura

Maximizar

$$z = 900x_1 + 400x_2$$

$$\rightarrow \text{con } x_1 = 0 \quad y \quad x_2 = 0$$

$$z = 0$$

$$x_3 = 30$$

$$x_4 = 20$$

$$x_5 = 12$$



Sujeto a

$$x_3 = 30 - 4x_1 - 1x_2 \text{ (d. backend)}$$

$$x_4 = 20 - 2x_1 - 3x_2 \text{ (d. front end)}$$

$$x_5 = 12 + 1x_1 - 4x_2 \text{ (pasantes)}$$

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$$

$(0,0, 30, 20, 12)$  es solución bás: c2

# Programación Lineal

Variante entrante y saliente (Simplex)

- Seleccionamos  $x_1$
- Buscamos cual es el menor valor de  $x_1$

$$x_3: 0 = 30 - 4x_1 - 0 \rightarrow x_1 = 7,5 \leftarrow \text{Queda ajustada}$$

$$x_4: 0 = 20 - 2x_1 - 0 \rightarrow x_1 = 10$$

$$x_5: 0 = 12 + x_1 - 0 \rightarrow x_1 = 12$$

- La primera variable libre que queda ajustada es  $x_3$
- La nueva solución es

$$(7,5, 0, 0, 20, 12) \rightarrow z = 6750$$



# Programación Lineal

## Proceso Pivot

- Despejamos  $x_1$  donde lo ajustamos

$$x_3 = 30 - 4x_1 - x_2 \rightarrow x_1 = 7.5 - \frac{x_3}{4} - \frac{x_2}{4}$$

- Reemplazamos  $x_1$  en todos lados

$$\text{Maximizar } z = 6750 - 225x_3 + 175x_2$$

Sujeto a:

$$x_1 = 7.5 - \frac{1}{4}x_3 - \frac{1}{4}x_2$$

$$x_4 = 5 + \frac{1}{2}x_3 - 2.5x_2$$

$$x_5 = 21.5 - \frac{1}{4}x_3 - 4.25x_2$$

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$$

# Programación Lineal

Vuelvo a iterar

La única variable básica que podemos seleccionar es  $x_2$

única con coeficiente positivo en la función objetivo.

$$z = 6750 - 225x_3 + 175x_2$$

Buscamos el máximo valor que puede tomar hasta ajustar a alguna variable libre.

$$(x1): 0 = 7,5 - \frac{1}{4} * 0 - \frac{1}{4}x_2 \rightarrow x_2 = 30$$

$$(x4): 0 = 5 + \frac{1}{2} * 0 - 2,5x_2 \rightarrow x_2 = 2 \quad \leftarrow \text{Variable entrante } x_4 \text{ y saliente } x_2$$

$$(x5): 0 = 19,5 - \frac{1}{4} * 0 - 4,25x_2 \rightarrow x_2 = 38/17$$

La nueva solución corresponde a  $(x_1, x_2, x_3, x_4, x_5) = (7, 2, 0, 0, 11)$ .

Con esta solución obtenemos como valor objetivo 7100

En el problema original corresponde a la solución (7,2)

Realizamos el proceso Pivot entre  $x_2$  y  $x_4$  y obtenemos el siguiente problema equivalente:

$$\text{Maximizar } z = 7100 - 190x_3 - 70x_4$$

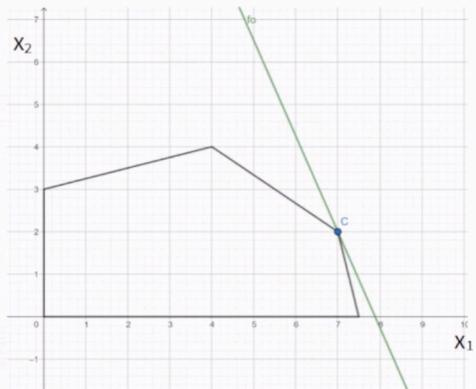
Sujeto a:

$$x_1 = 7 - 3/10x_3 + 1/10x_4$$

$$x_2 = 2 + 1/5x_3 - 2/5x_4$$

$$x_5 = 13 - 11/10x_3 - 17/10x_4$$

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$$



Terminamos en  $(7, 2) \rightarrow z = 7100$   
 $\downarrow \swarrow$   
 $x_1 \quad x_2$

# Programación Lineal

## Resolución de problemas

### Cambio mínimo

#### Dado

un grafo  $G = (V, E)$

Cada eje  $e$  tiene asociado un peso  $w_e \geq 0$ .

Contamos con un nodo "s" desde el que partimos y un nodo "t" al que queremos llegar.

#### Queremos

calcular la distancia mínima de "s" a "t"

- Cada nodo es una variable de decisión  $d_v$  que representa la distancia mínima de  $s$  a  $v$ .
- Tendremos  $|V|$  variables
- Queremos calcular  $d_t$  la distancia mínima de "s" a "t"
- Por cada eje  $e = (v, w)$  creamos una restricción
$$d_w \leq d_v + w_e$$
- $d_s$  es la distancia de "s" a si mismo  $d_s = 0$
- $d_v$  no pueden ser negativas

# Programación Lineal

Función objetivo: contendrá la variable de

Queremos maximizar  $d_t$ , las restricciones nos van a impedir sobreestimar

**Maximizar**

$$d_t$$

**Sujeto a:**

$$d_v \leq d_u + w_e \text{ para todo eje } e = (u, v)$$

$$d_s = 0$$

$$d_x \geq 0 \text{ para todo nodo } x \text{ del grafo}$$

# Programación Lineal

## Problema del flujo máximo

### Dado

un grafo  $G = (V, E)$

cada eje  $e$  tiene asociado una capacidad  $c_e \geq 0$ .

Contamos con un nodo fuente "s" y un nodo "t" sumidero.

### Queremos

calcular la cantidad máxima de flujo que enviamos desde "s" a "t".

- . Variable de decisión:  $f_e$  por cada eje  $e$  del grafo, representa la cantidad de flujo que se envía por eje  $\rightarrow |E|$  variables
- . Restricciones:
  - $f_e \leq c_e$
  - $\sum_{e=(x,n)} f_e = \sum_{e=(n,x)} f_e$
  - $f_t \geq 0$
$$\left. \begin{array}{l} \\ \\ \end{array} \right\} O(|V| + |E|)$$

# Programación Lineal

- Función objetivo: maximizar el flujo saliente de la fuente

$$\sum_{e \in (s, v)} f_e$$

Nos queda:

Maximizar

$$z = \sum_{e=(s, v) \in E} f_e$$

Sujeto a:

$f_e \leq c_e$  para todo eje  $e=(u, v)$  del grafo

$\sum_{e=(x, v) \in E} f_e = \sum_{e=(v, x) \in E} f_e$  para todo nodo  $v$  del grafo excepto la fuente y sumidero

$f_e \geq 0$  para todo eje  $e$  del grafo

- Luego de ejecutar Simplex tendremos:

- en cada variable la representación del flujo

- el valor objetivo corresponde a la cantidad de flujo transportado

# Programación Lineal

## Problemas de flujo de múltiples productores

### Dado

un grafo  $G = (V, E)$

Cada eje  $e$  tiene asociado una capacidad  $c_e \geq 0$ .

Contamos con "n" productos que deben ser transportados.

Cada producto  $p$  tiene demanda  $d_p$ , parte de un nodo  $s_p \in V$  y debe llegar a un nodo  $t_p \in V$ .

### Queremos

determinar si es posible realizar el transporte de todos los productos demandados.

- Variable de decisión:  $f_{ep}$  (flujo de producto  $p$  que pasa por  $e$ )  
 $\hookrightarrow |V| \cdot n$  variables
- Restricciones:
  - $f_{ep} \geq 0$
  - $\sum_{p=1}^n f_{ep} \leq c_e$
  - $\sum_{e=(x,y)} f_{ep} - \sum_{e=(y,x)} f_{ep} = 0$
  - $\sum_{e=(x,y)} f_{ep} - \sum_{e=(y,x)} f_{ep} = d_p \rightarrow$  nodos productores

# Programación Lineal

- Función objetivo: queremos saber si se puede satisfacer la demanda. No es maximización o minimización.

$$Z = 0$$

No queda:

Maximizar

$$Z = 0$$

Sujeto a

$$f_{ep} \geq 0 \text{ para todo eje y todo producto}$$

$$\sum_{p=1}^n f_{ep} \leq f_e \text{ para todo eje } e$$

$$\sum_{e=(x,v) \in E} f_{ep} - \sum_{e=(v,x) \in E} f_{ep} = 0 \text{ Para todo nodo } v \text{ (no productor ni consumidor)}$$

$$\sum_{e=(x,v) \in E} f_{ep} - \sum_{e=(v,x) \in E} f_{ep} = d_p \text{ para los nodos productores del producto } p$$

Luego de ejecutar Simplex tendremos

En cada variable la representación del flujo por producto que recorre cada eje

Debemos verificar que

la suma de los ejes que salen de los nodos productores corresponde a  $d_p$

# Programación Lineal

## Programación entera y 0-1

0-1: los valores de las variables solo pueden ser 0 o 1

### Entera

- turco que el conjunto de las soluciones factibles tomen en sus variables números enteros
  - Podríamos no obtener el valor óptimo o peor, uno no factible
  - Si: todos los puntos extremos son enteros, simplex sirve

### Ejemplo:

El punto extremo que nos brinda una mayor ganancia

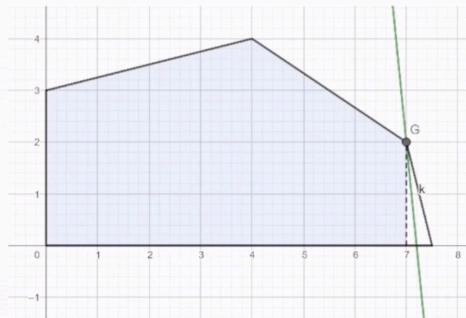
corresponde a  $(x_1, x_2) = (7, 5)$

con un valor objetivo de 750.

No podemos ejecutar fracciones de proyectos de consultoría.

Simplex no es óptimo para este caso.

El resultado óptimo es  $(7; 2)$  que corresponde a otro punto extremo.



# Randomizados

## Algoritmos Randomizados

### Monte Carlo

- Resultados: probabilísticamente correctos
- Probabilidad grande de obtener un valor correcto
- Ejecución en tiempo polinomial

### Las Vegas

- Resultados: correctos
- No tiene cota de tiempo de ejecución
- Ejecución probabilísticamente rápida

# Randomizados

## Clase RP

A aquellos problemas de decisión

Para los que existe un programa “M” randomizado

Que se ejecuta en tiempo polinomial

Tal que para toda instancia I del problema

Si I es “si”, entonces  $\text{pr}(M(I,r) = \text{"si"}) \geq \frac{1}{2}$

Si I es “no”, entonces  $\text{pr}(M(I,r) = \text{"si"}) = 0$  ← No hay falsos positivos

por lo menos en el 50% me da un resultado

		Respuesta Producida	
		SI	NO
Respuesta Correcta	SI	$\geq \frac{1}{2}$	$\leq \frac{1}{2}$
	NO	0	1

# Randomizados

## Clase co-RP

### Se conoce

Como "co-RP" (o "co-R")

### A aquellos problemas de decisión

Para los que existe un programa "M" randomizado

Que se ejecuta en tiempo polinomial

### Tal que para toda instancia I del problema

Si I es "si", entonces  $\text{pr}(M(I,r) = \text{"no"}) = 0$   $\leftarrow$  No hay falsos negativos

Si I es "no", entonces  $\text{pr}(M(I,r) = \text{"no"}) \geq \frac{1}{2}$

		Respuesta Producida	
		SI	NO
Respuesta Correcta	SI	1	0
	NO	$\leq \frac{1}{2}$	$\geq \frac{1}{2}$

# Randomizados

## Clase ZPP

Se conoce

Como zero-error probabilistic P (ZPP)

A aquellos problemas de decisión

Que pertenecen a RP y co-RP

Para toda instancia I del problema

Podemos ejecutar el algoritmo en RP y co-RP

En tiempo polinomial tendremos 3 respuestas posibles

Si, No y No Se sabe

La repetición de un numero no determinado de ejecuciones

Nos asegura obtener el resultado correcto

RP	co-RP	ZPP
NO	NO	NO
NO	SI	NO SE
SI	NO	(imposible)
SI	SI	SI

Corresponden a los  
algoritmos  
conocidos como Las  
Vegas  
 $RP \cap co-RP$

# Randomizados

## Clase BPP

Se conoce como

bounded-error probabilistic P (BPP)

A aquellos problemas de decisión

Para los que existe un programa "M" randomizado

Que se ejecuta en tiempo polinomial

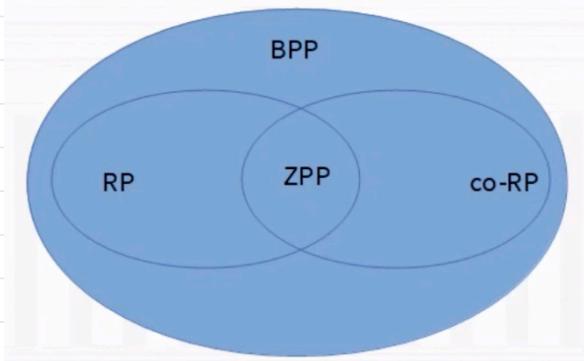
Tal que para toda instancia I del problema

Si I es "sí", entonces  $\text{pr}(M(I,r) = \text{"sí"}) \geq 2/3$

Si I es "no", entonces  $\text{pr}(M(I,r) = \text{"sí"}) \leq 1/3$

No podemos estar seguros si el resultado es correcto, pero tiene una alta probabilidad

		Respuesta Producida	
		SI	NO
Respuesta Correcta	SI	$\geq 2/3$	$\leq 1/3$
	NO	$\leq 1/3$	$\geq 2/3$



Randomizados

Randomizados

Randomizados

Randomizados