

2024

Resumen general =

Disección y conquista

- dividir el problema en sub-problemas
- resolver cada subproblema recursivamente
- combinar las soluciones a cada subproblema

→ teorema maestro:

$$T(n) = AT\left(\frac{n}{B}\right) + O(n^c)$$

$$< T(n) = O(n^c)$$

$$\text{si } \log_B(A) = \frac{\log B}{\log A} = T(n) = O(n^c \log n)$$

$$> T(n) = O(n^{\log_B A})$$

A = cant. de llamados recursivos
B = proporción del tamaño original con el que llamanos recursivamente

O(n^c) = el costo de partir y juntar (todo lo que no son los llamados recursivos)

- ej.
 - Búsqueda binaria
 - MergeSort y Quicksort
 - Árboles y heaps

AVANZADOS.

→ Multiplicación de números muy grandes

- Multiplicamos dos enteros de largos m x n
 - complejidad: O(n x m)

→ nos preguntamos (dada la multiplicación)
¿necesitamos TODOS los PARALES?

$$\begin{array}{r} 1234 \\ \times 56 \\ \hline 7404 \\ 6170 \\ \hline \end{array}$$

→ Separamos una multiplicación como si trabajáramos en base 2: $x \cdot y = (x_1 \cdot 2^{n/2} + x_0) \cdot (y_1 \cdot 2^{n/2} + y_0)$

y de aca vemos que hay 4 multiplicaciones de las subpartes. \Rightarrow no mejoro nada !!
lo que nos da $T(n) = 4T(n/2) + O(n) \rightarrow O(n^2)$

4 multiplicaciones \rightarrow cada una la Parte en 2

→ como sabemos que: $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$

En una multiplicación calculamos 2 términos

→ Solución = Algoritmo Karatsuba-Offman

def multiplicación_bisint(x, y):

 n largo de x e y son pequeños retorno x * y, sino:

$$x := x \cdot 2^{n/2} + x_0$$

$$y := y \cdot 2^{n/2} + y_0$$

$$P := \text{multiplicación_bisint}(x_1 + x_0, y_1 + y_0)$$

$$x_0y_0 = \text{multiplicación_bisint}(x_0, y_0)$$

$$x_1y_1 = \text{multiplicación_bisint}(x_1, y_1)$$

$$\text{return } x_1y_1 \cdot 2^n + (P - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$$

$$T(n) = 3T(n/2) + O(n) \rightarrow O(n^{\log_2 3}) \approx O(n^{1.5})$$

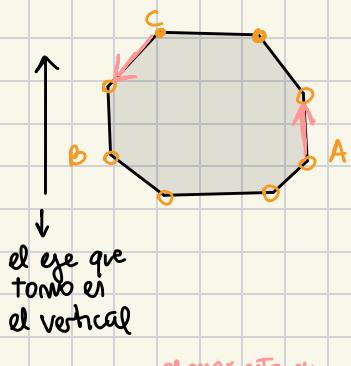
→ Obtener extremo de un polígono

- n vértices en sentido Antihorario
- Problema= obtener el vértice extremo (el maximo) respecto a un eje u
- El polígono debe ser convexo, osea que no se pueda cortar más de dos veces, osea que los ángulos interiores ≤ 180 gr.

→ $e_i = \text{segmento que va de } v_i \text{ a } v_{i+1}$

$$e_{v_i} = v_{i+1} - v_i$$

→ importante = el sentido respecto a la proyección



⇒ convexo: no lo puedo cortar más de dos veces.

⇒ Supongo que el max está entre los vértices A y B

⇒ me fijo el vector director de A que es el anterior con el sentido antihorario - el actual

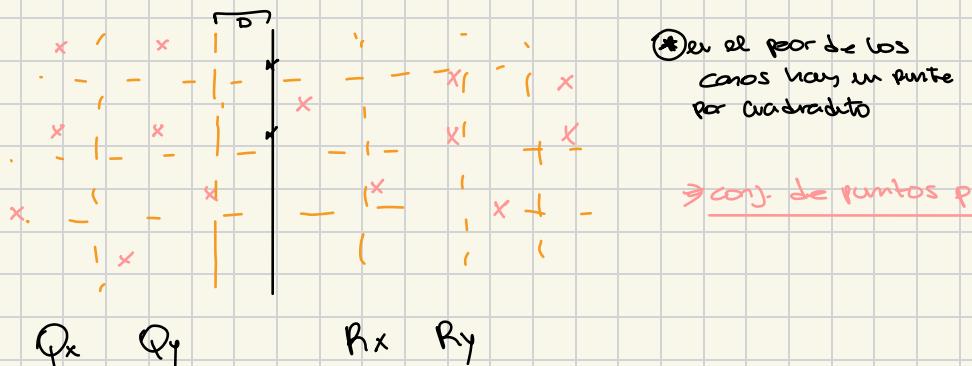
⇒ A va para arriba (max), si C va para abajo (min), entonces el maximo debe estar entre el siguiente de A y C

⇒ claramente debe ser convexo para que lo anterior se cumpla

(+) Por arriba o por debajo de A.
⇒ en con respecto a la proyección

→ Buscando puntos más cercanos en 2 dimensiones

- dado n puntos en un plano, buscar la pareja que se encuentre más cercana
- Sencillo, ver todas las distancias = $O(n^2)$
- Solución mejor: Busco la pareja más cercana del lado derecho, busco la más cercana del lado izquierdo, y luego en tiempo lineal busco los más cercanos $\Rightarrow O(n \log n)$



- 1) voy a ordenar por px y por py
- 2) obtengo los del lado izq y los del lado derecho
- 3) voy al eje medio de px y ahí creo las rectas
- 4) voy elemento a elemento los 2 mas y veo a que cuadrante de x de donde corresponde su otro de iz
- 5) comparo a todos los de un lado todos con todos y lo mismo para la otra mitad pero no entre si
- 6) dividido por cuadrantes de tamaño D, siendo D la distancia mínima de los 2 puntos más cercanos y por lo tanto no vale haber dos puntos de en un mismo cuadrante
- 7) creo (s_x, s_y) con los puntos que estén a $\pm \frac{D}{2}$
- 8) msp el $P_y \geq s_y$ hay 2 puntos a distancia d del la frontera tienen como mucho a 16 posiciones en el arreglo P_y
- 9) comparo este s_x con los siguientes 15, y como 15 es una cte y por lo tanto $O(n)$.

def closet_pairs_rec(px, py):

if len(px) <= 3: return el min de comparar
construir Qx, Qy, Rx, Ry ($O(n)$) cada punto.

$g_0, g_1 = \text{closet_pairs_rec}(Qx, Qy)$

$r_0, r_1 = \text{closet_pairs_rec}(Rx, Ry)$

$d = \min(d_{\text{int}}(g_0, g_1), d_{\text{int}}(r_0, r_1))$

$x^* = \max_{x \in Qx} \text{coordenada } x$ de Qx

S: Puntos de P que están a distancia $\leq d$ de la recta $X = x^*$

construir S_y ($O(n)$)

Por cada punto s de S_y computar distancia contra los siguientes 15 puntos, quedarse con s y s' que minimizan la distancia

if $d_{\text{int}}(s, s') < d$: return s, s'

elif $d_{\text{int}}(g_0, g_1) < d_{\text{int}}(r_0, r_1)$: return g_0, g_1

else: return r_0, r_1

→ complejidad:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

→ Multiplicación de matrices:

- Sencillo = $O(n^3)$

- Lo mejorado =

→ dividir a cada matriz de $n \times n$ en 4 submatrices de $n/2 \times n/2$

... $T(n) = 8T(n/2) + O(1) \rightarrow O(n^3)$, pero si en vez de 8 llamados recursivos hacerlos 7 (con algo similar a Karatsuba-Offman).

$$T(n) = 7T(n/2) + O(1) \rightarrow T(n) = O(n^{\log_2 7}) \approx O(n^{2.8})$$

→ Transformada rápida de Fourier

- 2 vectores A y B, queremos obtener la convolución entre ambos

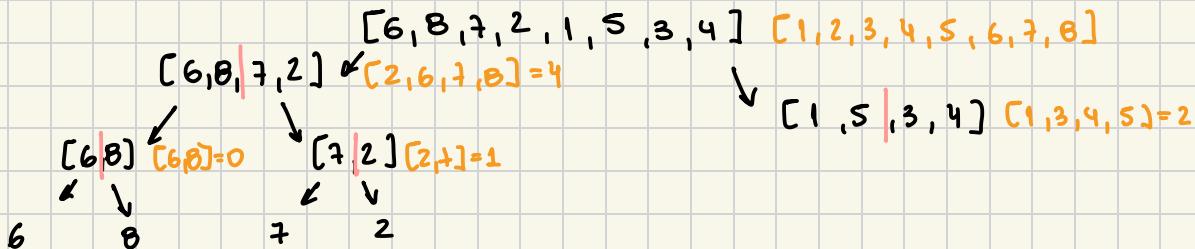
→ muy matemático (x)

→ conteo de inversiones:

- Conjunto de n elementos + 2 arreglos/lentas ordenados por diferentes criterios (A y B)

→ dar una medida de semejanza entre dichas listas

- Solución: mergesort + conteo



$$4 + 2 + (4 \text{ por el } 1, 3 \text{ por el } 3, 3 \text{ por el } 4, 3 \text{ por el } 5) = 19 \text{ (el max rima 28)}$$

Complejidad = $O(n \log n)$, memoria que mergesort p'j' contar no me cuenta (4)

Greedy:

- Se aplica una regla sencilla que nos permite obtener el óptimo local a mi estado actual
- Se aplica iterativamente esa regla, esperando que esto nos lleve al óptimo general

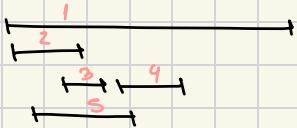
Ventajas vs desventajas:

- no siempre da el óptimo
- Demostrar que da un resultado óptimo es difícil
- + intuitivo de pensar, fácil de entender
- + suelen funcionar rápido
- + para problemas complejos suelen dar buenas aproximaciones

→ Ejercicios:

① Scheduling:

- tengo un aula/sala donde quiero dar charlas.
Las charlas tienen horario de inicio y fin.
Quiero utilizar el aula para dar la mayor cantidad de charlas posibles.



→ lo mejor es ordenar las charlas por tiempo de finalización

$$(2, 3, 5, 4, 1)$$

→ mejoran antes termina la charla, más rápido libera el aula

→ luego saco las que se superponen $(2, 3, 5, 4, 1)$

→ como todos valen lo mismo no importa cual saco

Conclusiones:

- En greedy lg' aplico la regla sencilla \Rightarrow Agarrar siempre la que no colisione y que termine antes
- Ordenar para mejorar la complejidad

Algoritmo:

```
def scheduling(horarios):  
    horarios_ordenados = ordenar_por_horario_fin(horarios)  
    charlas = []  
    for horario in horarios_ordenados:  
        if len(charlas) == 0 or not hay_interseccion(charlas[-1], horario):  
            charlas.append(horario)  
    return charlas  
  
def hay_intersection(anterior, nueva):  
    return anterior[FIN] > nueva[INICIO]:
```

② Árboles de Huffman:

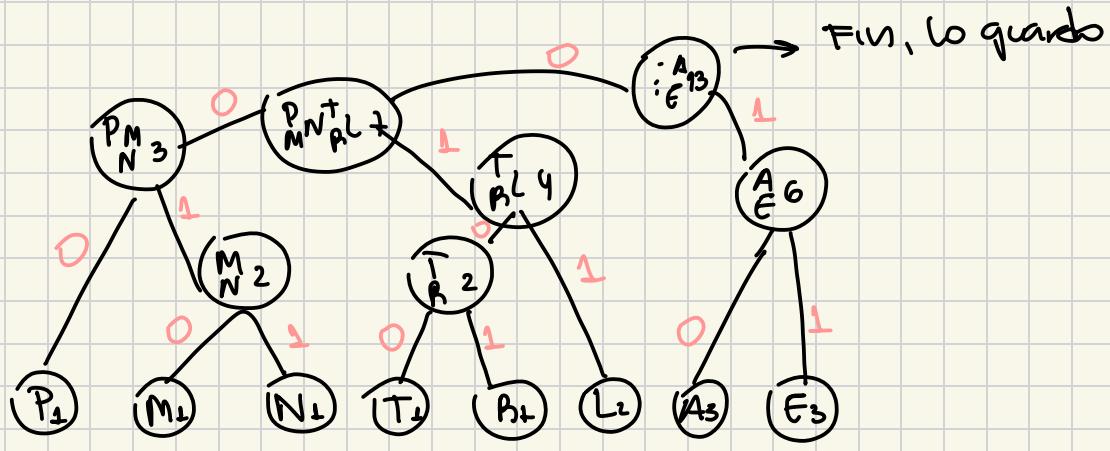
→ Plantea la forma de comprimir un texto en base a la frecuencia de los caracteres en el mismo

→ Utiliza heap de mínimos de forma auxiliar para ir generando el árbol de codigos

Ej. Palabra "PARALELAMENTE"

P A R A L E L A M E N T E

P	1
A	3
R	1
L	2
E	3
M	1
N	1
T	1



① Veo árboles para todos los tetras, cada uno de estos es un árbol

② Meto todos los árboles en un heap de mínimos y ando sacandolos de a 2

③ Con esos 2 que saeo hago un nuevo árbol con la frecuencia de ambos +.

④ Vuelvo a poner ese nuevo ↑ en el heap

⑤ A los ramas derecha les pongo un "1" y a los izquierdos un "0"

⑥ Recorremos el arbol y con eso veo como se encabe cada letra, desde Arriba de todo (seuencia 13) hasta la letra.

Por ej. $P = 000$, $A = 10$, $R = 0101$, $L = 011$,

$E = 11$,

⑦ Armo la palabra con estos bits, importante, veo como A que se repite muchisimo solo tiene 2 bits, mientras que P que aparece una sola vez tiene 3.

... 000-10-0101-10-011-10- ...

⑧ La persona que recibe esto \rightarrow hace lo mismo el arbol para poder leerlo. entregar para leerlo, cada vez que llego a una hoja es una letra.

⑨ Hay que mandarle la tabla de frecuencias tambien.

Algoritmo:

```
def huffman(texto):
    frecuencias = calcular_frecuencias(texto)
    q = heap_crear()
    for caracter in frecuencia:
        q.encolar(Hoja(caracter, frecuencia))
    while q.cantidad() > 1:
        t1 = q.desencolar()
        t2 = q.desencolar()
        q.encolar(Arbol(t1, t2, t1.frecuencia + t2.frecuencia))
    return codificar(q.desencolar())
```

\rightarrow PQ' en greedy? Sacar del heap y volver a meter. Test simple.
 \Leftarrow

Sempre sacar los 2 elementos que acumulan la menor cantidad de apariciones.

\rightarrow es optimo si el numero de bits es natural.

③ Problema del cambio:

→ Se tiene un sistema monetario (ej. el nostro). Se quiere dar "cambio" de una determinada cantidad de plato. Implementar un algoritmo que devuelva el cambio pedido, usando la mínima cantidad de monedas / billetes.

Billetes: 1, 100, 500, 50, 20, 10, 1000, 5

583. \Rightarrow doy el de 500, me falta dar 83 de cambio, repito... doy 50, me falta 33, doy de 20 13, doy de 10, doy 3 de 1

① Ordinamos de mayor a menor

(1000, 500, 200, 100, 50, 20, 10, 5, 1)

② agarré el más alto y veo que puedo hacer,

1: no puedo hacer nada \rightarrow con el siguiente más grande ... \rightarrow Regla sencilla

(13, 9, 7, 5, 2, 1) \Rightarrow sistema monetario

Cambio de 16:

\rightarrow doy 13, me quedan 3

\rightarrow doy una de 2 y una de 1

Pero sería mejor dar uno de 9 y otro de 7

Se puede anotar a $O(n)$ ✓

\Rightarrow conclusión: la optimidad de este algoritmo greedy depende del syst. monetario.

el cambio a dar

\Rightarrow Cada moneda \times la recorre a lo largo m veces entonces sería como mucho $O(n \times m)$ $= O(n^2)$

④ Problema de compra por inflación

Tenemos unos productos dados por un arreglo R , donde $R[i]$ nos dice el precio del producto. Cada día debemos comprar uno (y solo 1) de los productos, pero vivimos en una era de inflación y los precios aumentan todo el tiempo.

El precio del producto i el día j es $R[i]^{j+1}$ (j comenzando en 0)

Implementar un algoritmo greedy que nos indique el precio mínimo al que podemos comprar todos los productos

→ Comprar primero el más caro pq' es el que va a aumentar más

- Orden de mayor a menor por precio y cada dia compro el más caro que haya que no haya comprado ya

⇒ cada día son más baratos?

→ orden de menor a mayor, e igual a este

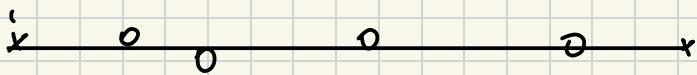
⑤ Problema de la carga de combustible.

Un camión debe viajar desde una ciudad a otra determinada a cargar combustible para poder llegar a destino. El tanque le permite viajar hasta N kilómetros.

Las estaciones se encuentran distribuidas a lo largo de la ruta siendo d_i la distancia desde la estación $i-1$ a la estación i .

1. Implementar un algoritmo que decide en qué estaciones conviene detenerse a cargar combustible, de manera que se detenga la menor cantidad de veces posibles.
2. Indicar N y justificar la complejidad del algoritmo

Ruta



→ Recorra lo máximo que pueda antes de que darse un parón.

→ Por cada estación me fijo si puedo llegar a la siguiente si puedo llegar no cargo, sino si

$N - d_0 \Rightarrow$ con esto puedo llegar a la siguiente?

↓
cantidad de parones

que quanto para llegar
a una estación

$N - d_0 \geq d_1 \downarrow$

distancia de d_0 a d_1

→ Es greedy pq tiene una regla sencilla, "seme banco la estación más cercana que este en mi rango N "

→ óptimo local \Rightarrow ir lo más lejos posible que este dentro de N
 \hookrightarrow así uno iterativamente hasta completar la ruta

→ esto es $O(n)$ pq' me estoy fijando en C/M en llegar a la siguiente y otras cosas más.

⑥ Problema de la mochila

→ Mochila con capacidad W (peso, volumen) N elementos a guardar. Cada elemento tiene un peso y valor. Queremos maximizar el valor de lo que mos llevamos sin pasarnos de capacidad.

Como el valor del objeto es algo positivo y el peso del objeto es algo negativo entonces Se prede utilizar una división matematica (\oplus/\ominus) - Valor/peso.

- mientras mayor peso peor
- orden de mayor a menor

↓
sempre estaban primero los que tengan más valor y menor peso

⑦ problema de la mochila II

→ Ahora si hace falta los elementos pueden fraccionarse, manteniendo la relación Valor/peso

→ utilizo el nuevo criterio de ordenar de mayor a menor por valor/peso
y para cada uno meto el maximo posible

↓

y ahora es optimo.

⑧ Scheduling II - minimizando latencia máxima

→ tareas con deadline (d_i) y duración (t_i), pero pueden hacerse en cualquier momento, siempre que sea antes del deadline.

Si se hacen después del deadline, incrementamos en la latencia.

Se busca minimizar la latencia máxima en el que las tareas se ejecuten. Es decir, si definimos que una tarea i empieza en s_i , entonces termina en $f_i = s_i + t_i$ y su latencia es $l_i = f_i - d_i$ (si $f_i > d_i$, sino 0)

→ lo siguiente es ordenar por deadline

↓
los que se tiene que hacer primero se hagan primero.

→ Pq' es ejecutable?

Inversiones: un schedule tiene una inversión si dentro de él tienen 2 elementos $s[i]$ y $s[j]$ tal que $i < j$, pero $d_i > d_j$.

Entonces todos los schedules sin inversiones, y sus tiempos de respuesta tienen la misma latencia máxima.

Inversión

d_1, d_2 y d_3 valen 3

$t_1 = 1, t_2 = 2, t_3 = 3$

hacer t_1, t_2, t_3 : llega tarde t_3 (por 3), latencia máxima 3

hacer t_1, t_3, t_2 : llegan tarde t_3 (por 1) y t_2 (por 3), latencia máxima 3

hacer t_2, t_1, t_3 : llega tarde t_3 (por 3), latencia máxima 3

hacer t_2, t_3, t_1 : llegan tarde t_3 (por 2) y t_1 (por 3), latencia máxima 3

hacer t_3, t_1, t_2 : llegan tarde t_1 (por 1) y t_2 (por 3), latencia máxima 3

hacer t_3, t_2, t_1 : llegan tarde t_2 (por 2) y t_1 (por 3), latencia máxima 3

⑨ optimal caching

Podemos tener hasta n elementos de memoria bien a mano, el resto tendremos que ir a RAM. Hay que decidir que guardar en la cache.

Tener un conjunto de datos \cup en memoria general (n en total), una memoria cache de $k < n$ elementos. Tener una secuencia de pedidos de datos d_i . Si d_i está en la cache, accedemos muy rápido, si no está hay "cache miss" + ahora hay que traer a la cache (y si la cache está llena tenemos que echar un dato previo). queremos minimizar la cantidad de cache miss.

→ variable a manejar \Rightarrow que evictar en cada caso \Rightarrow soluciones, ordenar por frecuencias de aparición de los elementos en la secuencia y tener en cache los más frecuentes.

⑩ Subcaminos \Rightarrow problema de examen.

Backtracking

→ Una combinación parcial que ya construimos no va a llevar a un resultado válido.



ej grafos:

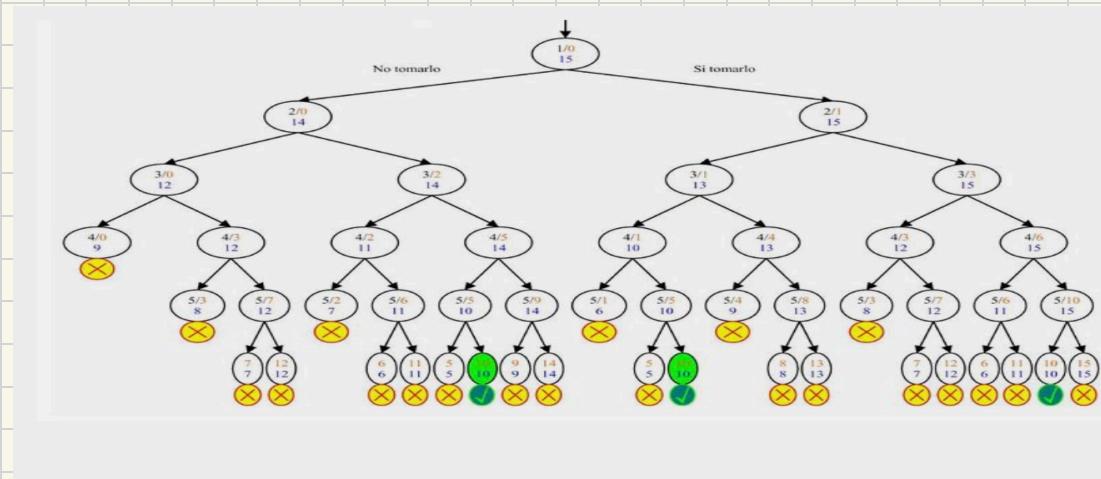
- ir sin recorrido → desmarcar los visitados
- ir por DFS] cuando no hay solución por "one cammino"

→ Segunda receta Backtracking:

- 1) Si ya encontre una solución, la devuelvo y termino.
 - 2) Avanzo si puedo
 - 3) Pruebo si la solución parcial es válida
 - a) Si no lo es, retrocedo y vuelvo a 2)
 - b) Si lo es, llamo recursivamente y vuelvo a 1)
 - 4) Si llegue hasta aca, ya probe con todo y no encontre una solución
- (no válido para todos los casos, pero el esquema suele ser similar)

→ Por cada variante (decisión tomada), se crea una ramaficación en el árbol de decisiones

→ Backtracking no continua sobre un caso que no tiene solución (poda)



① n reinas:

dado un tablero de ajedrez $N \times N$, ubicar (si es posible) a N reinas de tal manera que ninguna pueda comerse con ninguna

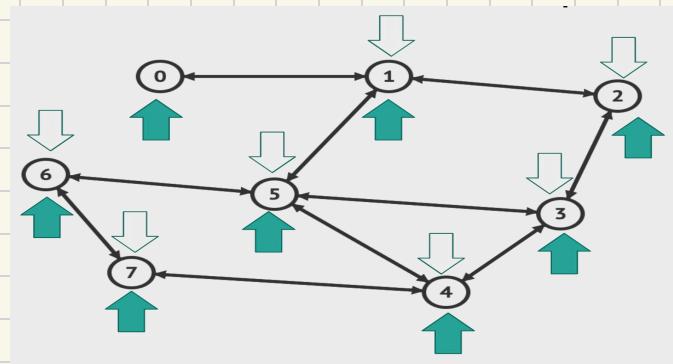


no pueden encontrarse 2 reinas
en la misma fila, columna o diagonal

② Independent set

Quiero guardar en un grafo N elementos. Debo elegir K vértices en los cuales guardar cada uno. Restricción! Queremos ver de ubicar K elementos sin que hayan dos adyacentes con elementos

→ Con fuerza bruta y Backtracking probamos todas las tres opciones en cada paso en el que estoy. En cada vértice tenemos opciones: ¿Guardo el elemento acá o no? Probamos AMBAS opciones.



Por Fuerza Bruta

```
def _ubicacion_FB(grafo, vertices, v_actual, puestos, n):
    if len(puestos) == n:
        return es_compatible(grafo, puestos)
    if v_actual == len(grafo):
        return False

    # Mis opciones son poner acá, o no
    puestos.add(vertices[v_actual])
    if _ubicacion_FB(grafo, vertices, v_actual + 1, puestos, n):
        return True
    puestos.remove(vertices[v_actual])
    return _ubicacion_FB(grafo, vertices, v_actual + 1, puestos, n)
```

Por Backtracking

```
def _ubicacion_BT(grafo, vertices, v_actual, puestos, n):
    if len(puestos) == n:
        return es_compatible(grafo, puestos)
    if v_actual == len(grafo):
        return False

    if not es_compatible(grafo, puestos):
        return False

    # Mis opciones son poner acá, o no
    puestos.add(vertices[v_actual])
    if _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n):
        return True
    puestos.remove(vertices[v_actual])
    return _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n)
```

```
def es_compatible(grafo, puestos):
    for v in puestos:
        for w in puestos:
            if v == w: continue
            if grafo.hay_arista(v, w):
                return False
    return True
```

Por Backtracking

```
def _ubicacion_BT(grafo, vertices, v_actual, puestos, n):
    if v_actual == len(grafo):
        return False
    if len(puestos) == n:
        return es_compatible(grafo, puestos)

    if not es_compatible(grafo, puestos) or ya_no_llego(grafo, vertices, v_actual, puestos):
        return False

    # Mis opciones son poner acá, o no
    puestos.add(vertices[v_actual])
    if _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n):
        return True
    puestos.remove(vertices[v_actual])
    return _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n)
```

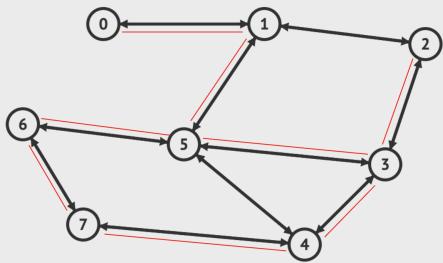
→ Complejidad: $FB = O(2^n)$

$BT =$ La poda lo hace mejor, pero la complejidad sigue siendo la misma

→ Obs: En vez de devolver una opción, devolver todas las posibles. No se puede parar al encontrar una

③ Camino hamiltoniano

→ Un camino de un grafo que visita todos los vértices del grafo una sola vez, si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano.



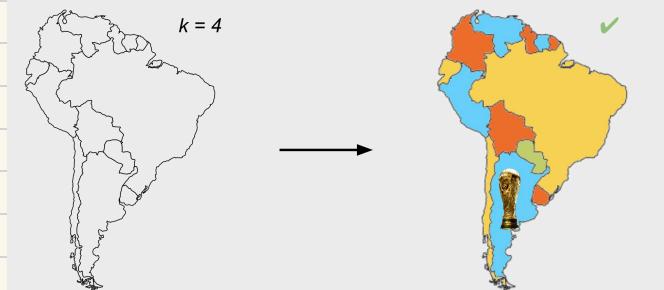
```
def camino_hamiltoniano_dfs(grafo, v, visitados, camino):
    visitados.add(v)
    camino.append(v)
    if len(visitados) == len(grafo):
        return True
    for w in grafo.adyacentes(v):
        if w not in visitados: # Esta es en sí nuestra poda
            if camino_hamiltoniano_dfs(grafo, w, visitados, camino):
                return True
    visitados.remove(v)      # Permitiendo volver a venir a este vértice
    camino.pop()             # por otro camino
    return False
```

No da lo mismo de dónde comencemos, así que...

```
def camino_hamiltoniano(grafo):
    camino = []
    visitados = set()
    for v in grafo:
        if camino_hamiltoniano_dfs(grafo, v, visitados, camino):
            return camino
    return None
```

④ Coloreo de grafos:

dado un grafo M K colores diferentes ¿es posible pintar los vértices de tal forma que ningún par de vértices adyacentes tengan el mismo color?



→ Si $K=2$? Sería el problema de detectar si el grafo es Bipartito.

→ Si $K \geq 3$, ahí ya es más difícil

Coloreo , pasos con receta :



- 1) Si ya encontre una solucion, la devuelvo y termino
- 2) Avanzo si puedo
- 3) Pruebo si la solucion parcial es valida
 - a) Si no lo es, retrocedo y vuelvo a 2)
 - b) Si lo es, llamo recursivamente y vuelvo a 1)
- 4) Si llegue hasta aca, ya probe con todo y no encontre una solucion

→ Pseudocódigo:

- 1) Si todos los países están coloreados, devuelvo True
- 2) Pruebo colorear con un color el siguiente país:
 - 3) Verifico si la solución parcial es válida
 - a) Si no lo es, retrocedo y vuelvo a 2) a probar con otro color
 - b) Si lo es, llamo recursivamente y vuelvo a 1)
- 4) Si llego hasta aca, ya probe con todo y no encontre una solucion

⑤ Sudoku :

Colocar numeros del 1 al 9 dadas las siguientes condiciones:

- llenar una grilla de 9×9
- las celdas estan dispuestas en 9 subgrupos de 3×3
- cada columna y cada fila no puede repetir numero
- cada subgroupo de 3×3 no puede repetir numero

```
def sudoku(cant_elem, M):  
    cant_elem = sig_pos_a_usar(cant_elem, M)  
    if cant_elem >= 9*9: return True  
  
    for num in range (1, 10):  
        if puedo_poner(num, cant_elem, M):  
            M[fila(cant_elem)][columna(cant_elem)] = num  
            if sudoku(cant_elem, M): return True  
            M[fila(cant_elem)][columna(cant_elem)] = 0  
  
    return False
```

1) Si ya encontre una solucion, la devuelvo y termino.

- 2) Avanzo si puedo
- 3) Pruebo si la solucion parcial es valida
 - a) Si no lo es, retrocedo y vuelvo a 2)
 - b) Si lo es, llamo recursivamente y vuelvo a 1)

4) Si llegue hasta aca, ya probe con todo y no encontre una solucion

→ Se puede modelar con un grafo? Si

→ Vertices = celdas individuales

→ Aristas = regiones del juego, conectan por ej. las celdas que no pueden tener el mismo numero, area que esten en la misma fila, columna o subcuadrícula.

6) El caballero en un tablero de ajedrez

dado una pieza de caballo de ajedrez dentro de su tablero, determinar los movimientos a hacer para que el caballo logre pasar por todos los casilleros, una única vez

→ El caballo se mueve en forma de L

```
def caballo(paso = 0):
    if completo(): return True ← encontró solución
    x, y = obtener_posicion_actual_caballo()
    for fila, col in movimientos_caballo(x,y):
        if not dentro_de_tablero(fila, col): continue
        if casillero_ya_marcado(fila, col): continue
        mover_a_posicion(fila, col, paso)
        if (caballo(paso + 1)):
            return True ← encontró solución
        volver_a_posicion(x,y) ← no encontró solución, vuelve para atrás
    return False
```

modelar con un grafo...

→ Vertices: cada uno de los casilleros del tablero

Aristas = los movimientos del caballo entre los casilleros.
cada vértice tendría aristas que lo conectan con los casilleros a los que el caballo puede moverse desde esa posición.

y teniendo el grafo podemos usar el problema del camino hamiltoniano para resolverlo

```
def camino_hamiltoniano_dfs(grafo, v, visitados, camino):
    visitados.add(v)
    camino.append(v)
    if len(visitados) == len(grafo):
        return True
    for w in grafo.adyacentes(v):
        if w not in visitados: # Esta es en sí nuestra poda
            if camino_hamiltoniano_dfs(grafo, w, visitados, camino):
                return True
    visitados.remove(v) # Permitiendo volver a venir a este vértice
    camino.pop() # por otro camino
    return False

def camino_hamiltoniano(grafo):
    camino = []
    visitados = set()
    for v in grafo:
        if camino_hamiltoniano(grafo, v, visitados, camino):
            return camino
    return None
```

7) Materias compatibles:

Se tiene una lista de materias que deben ser cursadas en el mismo cuatrimestre, cada materia está representada con una lista de cursos/horarios posibles a cursar (solo debe elegirse un horario por cada curso). Cada materia puede tener varios cursos.

Implementar un algoritmo de backtracking que devuelva un listado con todas las combinaciones posibles que permitan asistir a un curso de cada materia sin que se solapen los horarios. Considerar que existe una función son_compatibles(curso_1, curso_2) que dados dos cursos devuelve un valor booleano que indica si se pueden cursar al mismo tiempo.

La solución por fuerza bruta implicaría generar todas las posibles asignaciones.

Si para la materia 1 hay k_1 cursos, para la materia 2 k_2 cursos, etc

Consistiría entonces en combinar cada k_1 curso con cada k_2 curso ... con cada k_n para las n materias

El tiempo que tardaría es necesariamente proporcional a la cantidad de combinaciones posibles: $k_1 * k_2 * \dots * k_n$

Si cada curso tiene constante k cursos, la cantidad de asignaciones posibles $O(k^n)$

Ejemplo, si cada materia tiene dos cursos, es $O(2^n)$

```
def horarios_posibles(materias, solucion_parcial):
    # Si no nos quedan materias por ver
    if len(materias) == 0:
        if solucion_posible(solucion_parcial):
            return [solucion_parcial]
        else:
            return []
    # No es solucion total, pero es solucion parcial?
    if not solucion_posible(solucion_parcial):
        return []
    # Caso general, por ahora la solucion parcial es aceptada:
    materia_actual = materias.ver_primer()
    materias.borrar_primer()

    soluciones = []
    for curso in materia_actual:
        # Si es lista con soluciones, se agregan todas. Si devuelve lista vacia, no hará nada
        soluciones.extend(horarios_posibles(materias, solucion_parcial + [curso]))
    # Volver atrás un paso, para volver a poner la materia que sacamos:
    materias.guardar_primer(materia_actual)

    return soluciones

def solucion_posible(horarios):
    ultimo = horarios.ver_ultimo()
    for curso in horarios:
        if curso == ultimo: continue
        if not son_compatibles(curso, ultimo):
            return False
    return True
```

⑧ Sumatoria de n-dados

→ recibe una cantidad de dados n y una suma S .

La función debe devolver todas las tiradas posibles de n dados cuya suma es S . por ej. con $n=2$ y $S=7$, debe devolver $([1,6], [2,5], [3,4], [4,3], [5,2], [6,1])$

→ complejidad temporal?

→ complejidad Espacial?

```
def suma_dados(suma, cant_faltan, solucion_parcial, soluciones):
    if sum(solucion_parcial) == suma and cant_faltan == 0:
        soluciones.append(list(solucion_parcial))
        return

    if cant_faltan == 0 or sum(solucion_parcial) > n: return

    for valor in range(MIN_DADO, MAX_DADO+1):
        solucion_parcial.append(valor)
        suma_dados(suma, cant_faltan-1, solucion_parcial, soluciones)
        solucion_parcial.pop()
    return

def suma_dados(suma, cant_faltan, solucion_parcial, soluciones):
    if sum(solucion_parcial) == suma and cant_faltan == 0:
        soluciones.append(list(solucion_parcial))
        return

    if sum(solucion_parcial) + cant_faltan*MIN_DADO > suma: return
    if sum(solucion_parcial) + cant_faltan*MAX_DADO < suma: return

    for valor in range(MIN_DADO, MAX_DADO+1):
        solucion_parcial.append(valor)
        suma_dados(suma, cant_faltan-1, solucion_parcial, soluciones)
        solucion_parcial.pop()
    return
```

⑨ Subset sum

dada una lista de enteros positivos L y un entero n devuelva todos los subconjuntos de L que suman exactamente n.

```
def subset_sum(L, index, n, solucion_parcial, soluciones):
    # Si encuentro una solucion la agrego a las soluciones
    if sum(solucion_parcial) == n:
        soluciones.append(solucion_parcial[:])
        return

    # Si por esta rama me paso, dejo de probar
    if sum(solucion_parcial) > n or index >= len(L):
        return

    solucion_parcial.append(L[index])
    subset_sum(L, index+1, n, solucion_parcial, soluciones)
    solucion_parcial.pop()

    subset_sum(L, index+1, n, solucion_parcial, soluciones)
return
```

Programación dinámica

→ Utiliza memorización, es decir guardar los resultados previamente calculados.
Permite reducir la complejidad

→ Se prefiere la forma iterativa (Bottom up) porque:

- más fácil de entender
- más fácil calcular complejidad
- más estructura de como pensar la solución → primero ecuación de recurrencia, después programar

→ construye iterativamente la solución a subproblemas hasta llegar a la solución general

"→ si yo puedo asumir que ya tengo calculadas todas las soluciones anteriores (casos más pequeños) ¿cómo puedo usarlos para construir la que necesito?"

① scheduling con pesos:

Tengo un aula/sala donde quiero dar charlas. Las charlas tienen un horario de inicio y fin, y un peso asociado al valor de cada charla. Quiero utilizar el aula para maximizar la sumatoria de pesos de las charlas dadas.

→ ¿Qué era P?

$$P = \begin{cases} \text{dar la charla: } V_j + OPT(p(j)) \\ \text{no dar la charla: } OPT(j-1) \end{cases}$$

P → Arreglo que me dan $\geq m$ si

```
def sche_dinamico(n, p, valor):
    if n == 0:
        return 0
    M_SCHE = [0] * (n+1)
    M_SCHE[0] = 0
    for j in range(1, n+1):
        M_SCHE[j] = max(valor[j] + M_SCHE[p[j]], M_SCHE[j-1])
    return M_SCHE[n]
```

$$\begin{aligned} \text{Valor} &= [0, 2, 4, 4, 7, 2, 1] \\ P &= [1, 0, 0, 1, 0, 3, 3] \end{aligned}$$

Complejidad:

- Ordenar por fin: $O(n \log n)$
- Obtener $P[i]$: $O(\log n)$ → obtener P: $O(n \log n)$
- calcular M_SCHE: $O(n)$
- Final: $O(n \log n)$

d COMO recuperar la Solución?

def sube_Solucion(M_SCHE, valor, p, j, Solucion):

if $j == 0$:

return Solucion

if $valor[j] + M_SCHE[p[j]] \geq M_SCHE[j-1]$:

Solucion.append(j)

return sube_Solucion(M_SCHE, valor, p, p[j], Solucion)

else

return sube_Solucion(M_SCHE, valor, p, j-1, Solucion)

② los 2 escalones.

→ dada 1 Escalera y sabiendo que podemos subir escalones de a 1 o 2 pasos, encontrar cuantas formas diferentes hay de subir la escalera.

- Escalera nivel 0: 1 sola forma, quedarme estatica = 1 forma
- Escalera nivel 1: 1 paso simple = 1 forma
- Escalera nivel 2: dar 2 pasos simples, o dar 1 paso doble = 2 formas
- para una Escalera de nivel n ?

$$\begin{array}{c} \swarrow \\ E[N] = E[N-1] + E[N-2] \\ \downarrow \quad \downarrow \\ \text{opc 1} \quad \text{opc 2} \end{array}$$

OBS: si puedo subir de a 1, 2 o 3 escalones, entonces:

$$E[N] = E[N-1] + E[N-2] + E[N-3]$$

③ Juan El vago.

→ Ofertas de trabajo duran, pero no quiere trabajar dos días seguidos.

→ dado un arreglo con el monto esperado a ganar cada día, determinar por PD el maximo monto a ganar, con las condiciones dadas.

¿ como pensarlo?

→ la forma de los subproblemas:

- trabajos reducidos en función de la ganancia, sabiendo que trabajar un dia no nos permite trabajar el anterior. Los subproblemas se achican a medida que son menores los días

→ como se componen los subproblemas pequeños, solucionarlos para solucionar más grandes.

- Si trabajo hoy, compongo la mejor solución sin considerar el dia de ayer, como contrario compongo la mejor solución al dia de ayer (no trabajar hoy)

$$OPT(n) = \max \begin{cases} \rightarrow \text{no trabajar el dia } n : OPT(n-1) \\ \rightarrow \text{trabajar el dia } n: OPT(n-2) + V_n \end{cases}$$

Para todo $n \geq 2$.

$$n=0: OPT(0) = V_0$$

$$n=1: OPT(1) = V_1$$

```

def juan_el_vago_con_memoria(M, dias):
    G = [0]*dias
    G[0] = M[0]
    G[1] = max(M[0], M[1])

```

for d in range(2, dias):

$$G[d] = \max(M[d] + G[d-2], G[d-1])$$

return G

```
def construir_ejecuciones(G, M):
```

ejecuciones = []

d = len(G) - 1

while (d >= 0):

OPT_ayer = G[d-1] if d > 0 else 0

OPT_anterior = G[d-2] if d > 1 else 0

valor_hoy = M[d]

if valor_hoy + OPT_anterior >= OPT_ayer:

ejecuciones.append((0, d))

d -= 1

else

d -= 1

return ejecuciones

OBS: → Juan está teniendo un problema de scheduling.

→ Cada trabajo de juan es como una charla, con comienzo en el dia i y con fin en el dia $i+1$, ya que juan no esta dispuesto a trabajar en dia, ni trabajo el dia anterior

→ En particular: para cada dia i hay una "charla", con comienzo i y fin $i+1$

→ Otra forma de resolución: adestrar esto a un set acorde al de scheduling con pesos

③ Caminos posibles en un laberinto:

- Laberinto representado por una grilla
- calcular la cantidad de caminos posibles que \exists para llegar desde la posición $(0,0)$, hasta la $N \times M$.
- Movimientos permitidos: desde la esquina superior izquierda $(0,0)$, hacia abajo o hacia la derecha

f.c. de recurrencia: $P[i][j] = P[i-1][j] + P[i][j-1]$



con P = probabil.

→ Y si ahora pasar por cada casillero (i,j) nos da una ganancia V_{ij} ?

→ calcular la ganancia máxima desde $(0,0)$ hasta $N \times M$, con los movimientos permitidos hacia abajo ó hacia la derecha.

$$G[i][j] = \max(G[i-1][j], G[i][j-1]) + V[i][j]$$

OBS: si hubiera obstáculos entonces lo chequeo, voy por el otro camino.

④ Combinaciones a través del teclado del teléfono:

→ tengo:

- teclado del teléfono
- número inicial N

1	2	3
4	5	6
7	8	9
0		

→ debo:

- Encontrar la cantidad de números de longitud N , empezando por cierto botón N .

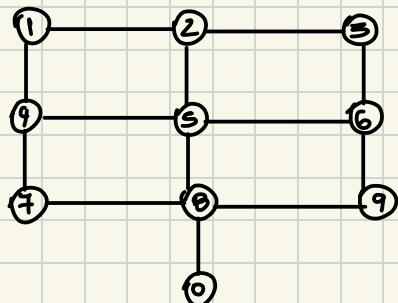
→ restricción:

- Solo se puede presionar un botón si está arriba, abajo, a la derecha o a la izquierda del botón actual.

→ soluciones:

- $N=1$, digitamos el número del inicio
- $N=2$, el inicial y el de arriba, de abajo, derecha e izquierda
 - ej: para $N=0$: 00, 08 (cont=2)
 - para $N=1$: 11, 12, 14 (cont=3)
 - para $N=2$: 22, 21, 23, 25 (cont=4)
 - para $N=5$: 55, 52, 54, 56, 58 (cont=5)

→ utilizamos un grafo:



- Forma de subproblemas: Cont de combinaciones posibles, conociendo la cont de pasos y el numero inicial

- COMO SE SOLUCIONAN LOS SUBPROBLEMAS:- Sabemos que si digito un numero estoy dando un paso, para considerar todos los combinaciones tenemos un paso menos, hay que referirnos a la solución con un paso menos para todos los numeros conectados

$$\text{F.c. de recurrencia} = C[\text{pasos}=i][\text{dnde}=v] = \sum_{\substack{\text{para todo} \\ \text{vecino}}} C[\text{pasos}=i-1][\text{dnde}=v\text{ vecino}]$$

Construir Solución Bottom-up.

o necesitamos:

- Cont. de pasos que quedan por hacer
- tecla de inicio

↳ Si elegimos bien los casos base siempre despondremos de esa info.

dado que los subproblemas tienen un orden, estos crecen a medida que se incrementa la cant. de pasos

o casos base:

- Pasos = 1 , para cada boton del teclado

$$C[\text{pasos} = i][\text{dnde} = v] = C[\text{pasos} = i-1][\text{dnde} = v] + \sum_{\substack{\text{Para todo} \\ \text{vecino}}} C[\text{pasos} = i-1][\text{dnde} = \text{vecino}]$$

def cont-combinaciones (grafo, pasos, tecla-inicial):

cont = [] []

for tecla in range (teclas del 0 al 9):

cont [0][tecla] = 0

cont [1][tecla] = 1

for i in range (2, Pasos+1):

for tecla in range (teclas del 0 al 9):

contador = 0

for vecino in grafo.adyacentes(tecla):

contador += cont[i-1][vecino]

cont[i][tecla] = contador // + cont[i-1][tecla] en grafo sin bucles

return cont[pasos][tecla_inicial]

⑤ Problema de la mochila:

- tenemos mochila con capacidad W. tenemos elementos a guardar. Cada elemento tiene una capacidad y un valor. Queremos maximizar el valor de lo que llevamos, sin excedernos

→ Pensar los subproblemas:

- llenar mochila de capacidad W , con 0 items. \Rightarrow ganancia = 0 .

- llenar mochila de capacidad W , con 1 item. \Rightarrow incluirlo o no incluirlo

→ conclusiones:

- o Peso menor a W

- o Si se incluye , la ganancia total sera = V

- Con 2 items:

- si no se incluye, disponemos de la solución de tamaño 1
 - si se incluye tenemos una ganancia = item 1 + item 2
 - el item debe poder entrar pero $item\ 1 + item\ 2 \leq W$

∴ Maximizar la ganancia para la modula de cap = W: Agregar o no agregar un elemento

- Si no se agrega: Mejor solución: la que maximiza sin el
→ si se agrega: la que se considera

1. parte de la gama nica sera Vi
 2. la mejor sub-solucion posible con menor capacidad:
una capacidad de W-pi

Fc. de recurrencia:

$$\text{OPT}(n, w) = \max \begin{cases} \text{no usar el elem. } \text{OPT}(n-1, w) \\ \text{usar el elem. } \text{OPT}(n-1, w - p_i) + v_i \end{cases}$$

↓
Cuando w sea $\geq p_i$

Definir modula iterativa(valor, peso, N, w)

$$df = [c_0] * (w+1) \text{ for } w \in \text{range}(N+1)$$

for i in range(1, N+1):

for w in range(1, $w+1$):

→ si el peso del elemento actual es $>$ que W , no quedamos con el anterior.

If $\rho_{\text{eff}}(L) > w$:

$$dp[i][w] = dp[l-1][w]$$

ele:

→ max utre marks on

$$dp[i][w] = \max (dp[i-1][w], valor[i] + dp[i-1][w - peso[i]])$$

return $\text{dp}[W][W]$

Valores = {0, 60, 100, 120} ↳ comparen en elemento

$$Perido = [0, 10, 20, 30]$$

$$N = \text{len}(\text{valor}) - 1$$

$$W = 50$$

Cont.
de
elmentos
(n)

Cap. de la modulación (W)

A 6x6 grid of black lines on white paper, consisting of five horizontal rows and five vertical columns, creating a total of 25 smaller squares.

Comp. $O(n \times m)$

⑥ El problema del cambio

se tiene un snt monetario.

se quiere dar "cambio" de una cant. de plata

devolver el cambio pedido con la menor cantidad de monedas/billetes.

→ forma de los subproblemas: dar cambio para una cant. de dinero, se podra dar cualquier moneda y quedaria menos cambio a dar. El tamaño en la cant. de cambio a dar.

→ como se componen para solucionar mas grandes: tengo \neq monedas que puedo utilizar. La mejor sera aquella que minimice la cant. total utilizada. probar todas y sacar el minimo

fc. de recurrencia: $C[\text{dinero}] = 1 + \min (\forall \text{moneda} \leq \text{dinero}: C[\text{dinero} - \text{moneda}])$
 $C[0] = 0$

def cant_monedas (snt_monetario, dinero):

```
comt = [0] * (dinero + 1)
for i in range (1, dinero + 1):
    minimo = i # usar todas monedas de 1
    for moneda in snt_monetario:
        if moneda > i
            continue
        cantidad = 1 + comt[i - moneda]
        if cantidad < minimo:
            minimo = cantidad
```

comt[i] = minimo

```
if comt[dinero] < minimo:
    minimo = comt[dinero]
```

comt[i] = minimo

return comt[dinero]

⑦ Bellman-Ford: Camino minimo desde un vertice, para grafos dirigidos, en caso de que haya aristas negativas, mientras no hayan ciclos negativos

Proceso:

- 1) inicializa vertices en infinito y padres en none. el orden lo tenemos de trasciende 0
- 2) iteramos V veces sobre todas las aristas, se busca la posibilidad de actualizar los pesos, buscando mejorar el camino minimo
- 3) finalizadas las V iteraciones, se verifica si aun se actualizan los pesos, en ese caso entramos ante un ciclo negativo.

Porque en PD?

- Plantea la forma de los subproblemas como: Encontrar el camino mínimo sabiendo que solamente podemos bajar por n vértices
- Se suele iterativamente de $n=0$ hasta $n = \# \text{Vértices}$
- Ningún camino óptimo puede atravesar más de V vértices

codigo:

```
def obtener_aristas (grafo):  
    result = []  
    for v in grafo:  
        for w in grafo.adyacentes:  
            result.append ((v,w, grafo.peso_arista(v,w)))  
    return result
```

```
def camino_minimo_bf (grafo, origen):  
    distancia = {}  
    Padre = {}  
    for v in grafo:  
        distancia[v] = 0  
    distancia[origen] = 0  
    Padre[origen] = None  
    aristas = obtener_aristas(grafo)  
    for i in range (len(grafo)):  
        for v, w, peso in aristas:  
            if distancia[v] + peso < distancia[w]:  
                Padre[w] = v  
                distancia[w] = distancia[v] + peso  
  
    for v, w, peso in aristas:  
        if distancia[v] + peso < distancia[w]:  
            return None # hay un ciclo Θ, lanzar excepciones  
  
    return Padre, distancia
```

⑧ Distancia de edición:

→ Alineamiento \Rightarrow Alineamiento

- La mejor forma de alinear dos cadenas es minimizar la distancia de edición.

Criterio:

- Por cada par de letras que coinciden, no hay costo
- Existe la penalidad S que refiere a no alinear las letras (gap)
- Por cada par de letras que no coinciden hay un costo $a(x_i, y_j)$ según la diferencia entre las letras x_i y y_j . Por ej. el costo de alinear V y B puede ser bajo, por considerar su parecido y cercanía en el teclado
- El costo total del alineamiento es la suma total de todos los costos pagados por brechas y por costo de reemplazo (cambiar letras)

Entonces calcular la distancia de edición significa encontrar el alineamiento óptimo tal que se minimice el valor de la distancia de edición, en decir, minimizar la sumatoria de costos de brechas y de reemplazos

→ El algoritmo trata las cadenas de atrás hacia adelante.

- Si el último carácter coincide, entonces no hay costo y puedo eliminarlo.

→ ej. lobos y OROS == lobo y ORO

→ ej: CASO y casa, se paga el costo de A y O y luego se calcula el de cas y cas

→ ej: cosa y cana (cana - my cases), tiene que poner una brecha en casa

Finalmente hay 4 opciones:

- 1) Los caracteres coinciden: se calcula con -1 ($i-1$ en X y $j-1$ en Y)
- 2) Reemplazar caracteres: sumar un costo y calcular la solución óptima para $i-1$ en X y $j-1$ en Y
- 3) Brecha en la 1ra cadena: sumar un costo de brecha y calcular la solución óptima para i en X y j en Y
- 4) Brecha en la 2da cadena: sumar un costo de brecha y calcular la solución óptima para $i-1$ en X y j en Y

- o los subproblemas se definen por el numero de caracteres de la cadena x que se pueden utilizar y el numero de caracteres en la cadena y que se puede utilizar

f_c de recurrencia:

si son iguales: $\text{OPT}(i-1, j-1)$

$$\text{OPT}(i, j) = \min \begin{cases} \alpha_{x_i, y_j} + \text{OPT}(i-1, j-1) \\ j + \text{OPT}(i, j-1) \\ j + \text{OPT}(i-1, j) \end{cases}$$

⑨ Subset-Sum:

- Conjunto de numeros (v_1, v_2, \dots, v_n) y queremos obtener un subconjunto de todos estos numeros cuya suma sea igual a un valor V , o approximarlo lo maximo posible.

↓ Maximizar la suma de los elementos sin superar V

$$\text{OPT}(n, V) = \max \begin{cases} \text{no usar el elemento: } \text{OPT}(n-1, V) \\ \text{usar el elemento: } \text{OPT}(n-1, V-v_i) + v_i \end{cases}$$

↓
cuando V sea mayor o igual a v_i

⇒ Similar al problema de la mochila

↳ se pierde usar para resolver sub-set sum

- 1) Convertir la entrada en una valida para el problema de la mochila $O(n)$
- 2) usar la solucion del problema de la mochila $(O(n \cdot 2^m))$
- 3) Verificar. $O(1)$

→ Si uno encuentra un valor igual a $V \Rightarrow$ resultado
sino, habra que encontrar el mas cercano al superior V

⑩ Tu a Londres y yo a California

- negocio con clientes en Londres y en California
- Arreglaron cada mes si comienza operar en una o en otra ciudad.
- los costos para cada mes pueden variar y son L_i y C_i para todo i
- Si un mes se opera en una ciudad y el siguiente en la otra habrá un costo fijo M por mudanza.
- dados los costos de operación en L y en C , indican la secuencia de las n localizaciones en las que operan durante n meses. Sabiendo que queremos minimizar los costos de operación. Se prede empezar en cualquier ciudad.
- Para un plan de n meses, el plan termina en L o en C .
- para un plan de $n-1$ meses, el lugar de finalización impacta en la posibilidad de optimizar el plan del mes n .
- Entonces:

Para un mes n que termina en L

- $L_n + \text{costos op. anteriores en } L$
- $L_n + \text{costos op. anteriores en } C, + \text{mudanza.}$

Para un mes n que termina en C .

- $C_n + \text{costos op. anteriores en } C$
- $C_n + \text{costos op. anteriores en } L + \text{mudanza.}$

Ec. de recurrencia:

$$\text{OPT Londres}[n] = L[n] + \min(\text{OPT Londres}[n-1], M + \text{OPT California}[n-1])$$

$$\text{OPT California}[n] = C[n] + \min(\text{OPT California}[n-1], M + \text{OPT Londres}[n-1])$$

Programación Lineal:

→ Permite resolver problemas de un sistema de ecuaciones lineal.

en decir: $c_0 + c_1 x_1 + \dots + c_n x_n \geq c$

EVITAR $x_0 x_1 \leq 10$

→ Componentes de PL

1) Variables

2) Ecuaciones / inecuaciones lineales: $\sum a_i x_i \geq b$

3) Buscamos maximizar / minimizar una función objetivo: $\max \sum c_i x_i$

4) Aplicamos un algoritmo que resuelve el modelo lineal (Simplex)

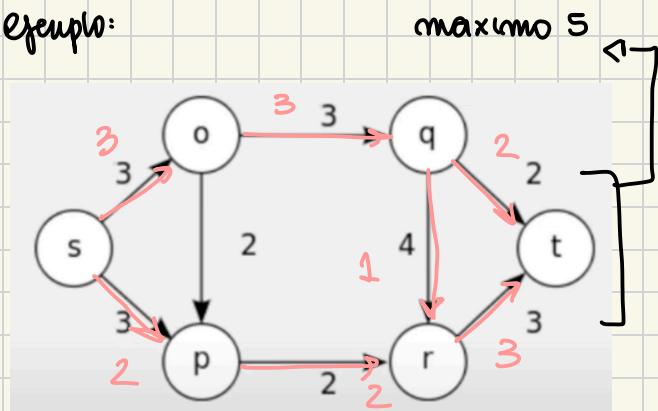
OBS: las restricciones crean un dominio de soluciones válidas, no siempre existen

Problema de flujo:

- Modelar con grafos flujos de materiales (ej. cloacas, info en redes)
- el grafo es dirigido
- Unica Fuente y unico Sumidero (\Rightarrow una una componente fuertemente conexa) Fuente: vértice con grado de entrada 0. Sumidero: vértice con grado de salida 0.
- cada vértice intermedio simplemente transfiere lo que le pasan. no produce, ni consume.
- cada arista tiene un peso que refleja la capacidad de transporte por esa vía.

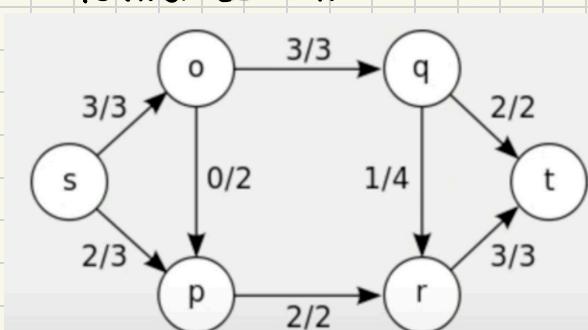
objetivo: obtener la máxima cantidad de flujo que puede transferirse de la fuente al sumidero

Ejemplo:



Flujo máximo

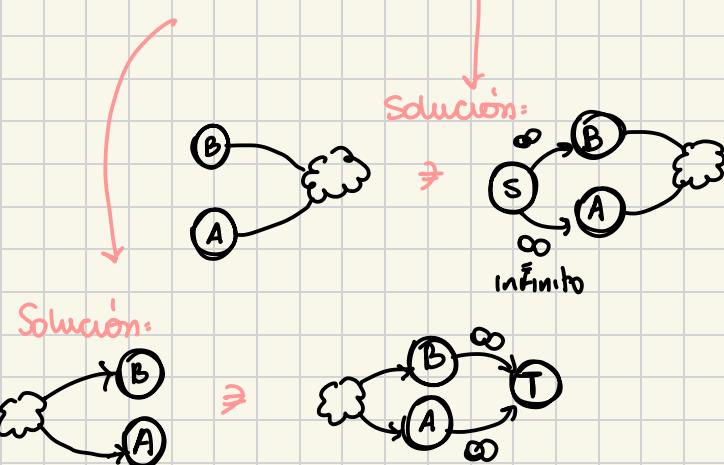
forma de representarlo:



Condiciones generales:

- no se aceptan bucles \Rightarrow 0
- no se aceptan ciclos de dos vértices (anillas antiparalelas) \Rightarrow 0
- todos los vértices deben poder llegar al sumidero
- solo una fuente
- solo un sumidero

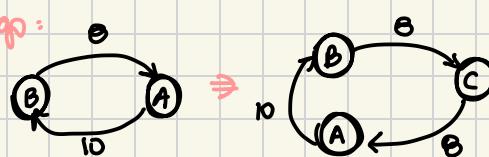
Solución:



Solución:

Solución:

tempo:



→ Red residual:

- Grafo con los mismos vértices, pero tiene como aristas:

1. las mismas que el original, al que aun les quede capacidad por utilizar. El peso es esa capacidad restante

2. la arista opuesta, con peso la capacidad utilizada
3. si alguno de los pesos anteriores es 0, no hay arista

○ En el peor de los casos el grafo residual tiene el doble de aristas

○ Si encontramos un camino de la fuente (s) al sumidero (t) en el grafo residual, entonces encontramos un camino por el que podemos aumentar el flujo.

- Buscamos aumentar el flujo total, pero puede reducirse el que pasa por una arista particular

○ Algoritmo Ford-Fulkerson: $O(E \cdot v)$

```
def flujo(grafo, s, t):
    flujo = {}
    for v in grafo:
        for w in grafo.adyacentes(v):
            flujo[(v, w)] = 0
    grafo_residual = copiar(grafo)
    while (camino = obtener_camino(grafo_residual, s, t)) is not None:
        capacidad_residual_camino = min_peso(grafo_residual, camino)
        for i in range(1, len(camino)):
            if grafo.hay_arista(camino[i-1], camino[i]):
                flujo[(camino[i-1], camino[i])] += capacidad_residual_camino
                actualizar_grafo_residual(grafo_residual, camino[i-1], camino[i], capacidad_residual_camino)
            else:
                flujo[(camino[i], camino[i-1])] -= capacidad_residual_camino
                actualizar_grafo_residual(grafo_residual, camino[i-1], camino[i], capacidad_residual_camino)

    return flujo
```

```
def actualizar_grafo_residual(grafo_residual, u, v, valor):
    peso_anterior = grafo_residual.peso(u, v)
    if peso_anterior == valor:
        grafo_residual.remover_arista(u, v)
    else:
        grafo_residual.cambiar_peso(u, v, peso_anterior - valor)
    if not grafo_residual.hay_arista(v, u):
        grafo_residual.agregar_arista(v, u, valor)
    else:
        grafo_residual.cambiar_peso(v, u, grafo_residual.peso(v, u) + valor)
```

- Complejidad: ○ depende de como elegimos buscar el camino \Rightarrow nosotros lo hacemos por BFS pg' siempre funciona (teorema de Edmonds-Karp)

o Seguimiento:

Seguimiento



Arista	Flujo
0, 1	11
0, 2	13
1, 3	13
3, 5	19
4, 5	5
2, 4	11
2, 1	2
4, 3	6

1

Seguimiento



Arista	Flujo
0, 1	11
0, 2	0
1, 3	11
3, 5	11
4, 5	0
2, 4	0
2, 1	0
4, 3	0

2

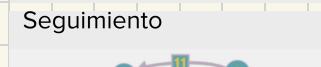
Seguimiento



Arista	Flujo
0, 1	11
0, 2	4
1, 3	11
3, 5	11
4, 5	4
2, 4	4
2, 1	0
4, 3	0

3

Seguimiento



Arista	Flujo
0, 1	11
0, 2	4
1, 3	11
3, 5	11
4, 5	4
2, 4	4
2, 1	0
4, 3	0

4

Seguimiento



Arista	Flujo
0, 1	11
0, 2	4
1, 3	11
3, 5	11
4, 5	4
2, 4	4
2, 1	0
4, 3	0

5

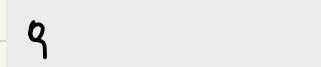
Seguimiento



Arista	Flujo
0, 1	11
0, 2	4
1, 3	11
3, 5	11
4, 5	4
2, 4	4
2, 1	0
4, 3	0

6

Seguimiento



Arista	Flujo
0, 1	11
0, 2	11
1, 3	11
3, 5	18
4, 5	4
2, 4	11
2, 1	0
4, 3	7

7

Seguimiento



Arista	Flujo
0, 1	11
0, 2	11
1, 3	11
3, 5	18
4, 5	4
2, 4	11
2, 1	0
4, 3	7

8

Seguimiento



Arista	Flujo
0, 1	11
0, 2	12
1, 3	12
3, 5	19
4, 5	4
2, 4	11
2, 1	1
4, 3	7

9

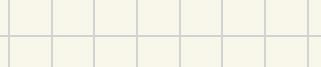
Seguimiento



Arista	Flujo
0, 1	11
0, 2	12
1, 3	12
3, 5	19
4, 5	4
2, 4	11
2, 1	1
4, 3	7

10

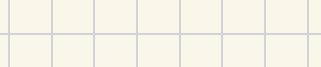
Seguimiento: red original



Arista	Flujo
0, 1	11
0, 2	12
1, 3	12
3, 5	19
4, 5	4
2, 4	11
2, 1	1
4, 3	7

11

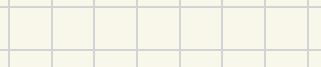
Seguimiento



Arista	Flujo
0, 1	11
0, 2	12
1, 3	12
3, 5	19
4, 5	4
2, 4	11
2, 1	1
4, 3	7

12

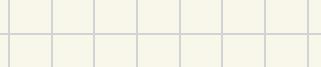
Seguimiento



Arista	Flujo
0, 1	11
0, 2	12
1, 3	12
3, 5	19
4, 5	4
2, 4	11
2, 1	1
4, 3	7

13

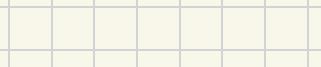
Seguimiento



Arista	Flujo
0, 1	11
0, 2	13
1, 3	13
3, 5	19
4, 5	5
2, 4	11
2, 1	2
4, 3	6

14

Seguimiento



Arista	Flujo
0, 1	11
0, 2	13
1, 3	13
3, 5	19
4, 5	5
2, 4	11
2, 1	2
4, 3	6

Corte mínimo.

→ en el peso total (mínimo) que se necesita desconectar para que el grafo deje de estar conectado.

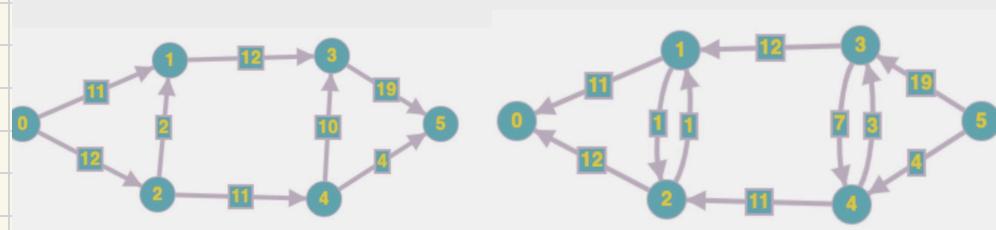
→ teorema max flow min cut:

Si el grafo corresponde a una red de flujo, entonces el corte mínimo tiene capacidad igual que el flujo máximo

¿Cómo lo obtenemos?

- 1) Agarramos el grafo residual
- 2) Vemos todos los vértices a los cuales llegamos desde la fuente
- 3) Vemos los aristas (del grafo original) que vayan de un vértice al que podemos llegar (en el residual) a uno que no (en el residual), son parte del corte

En nuestro ejemplo...

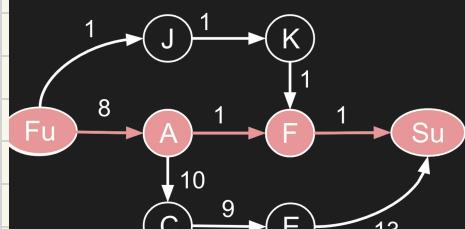


Integer value Flows:

- Si la red solo tiene aristas con capacidades enteras, entonces existe un flujo máximo en el que la capacidad utilizada de cada arista es un número entero

Segundo seguimiento:

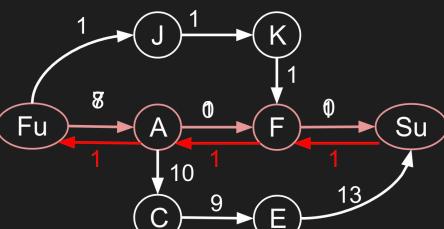
APLICO BFS Y CONSIGO CAMINO MÁS CORTO DE FUENTE(Fu) a SUMIDERO (Su)



BFS
Mínimo: 1

1

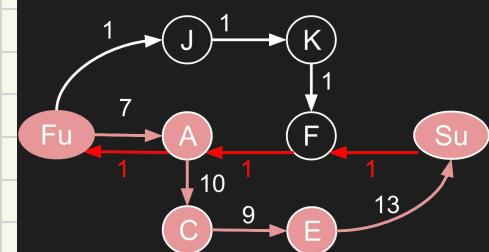
ACTUALIZO RED RESIDUAL (usando el BFS y el mínimo)



BFS
Mínimo: 1

2

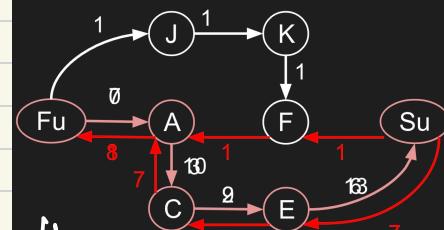
APLICO BFS Y CONSIGO CAMINO MÁS CORTO DE FUENTE(Fu) a SUMIDERO (Su)



BFS
Mínimo: 7

3

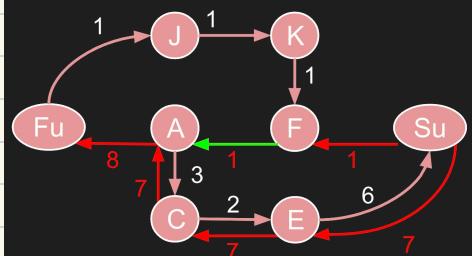
ACTUALIZO RED RESIDUAL (usando el BFS y el mínimo)



BFS
Mínimo: 7

4

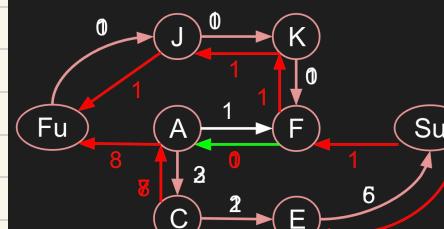
APLICO BFS Y CONSIGO CAMINO MÁS CORTO DE FUENTE(Fu) a SUMIDERO (Su)



BFS
Mínimo: 1

5

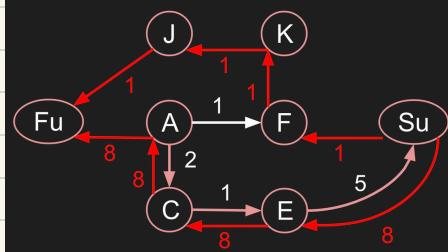
ACTUALIZO RED RESIDUAL (usando el BFS y el mínimo)



BFS
Mínimo: 1

6

Como no hay camino BFS de la fuente al sumidero termina



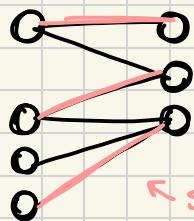
7

FU-J	1
FU-A	8
J-K	1
K-F	1
F-Su	1
A-C	8
C-E	8
E-Su	8

Aplicaciones de las redes de flujo:

① Perfect Bipartite Matching:

Dado un grafo no dirigido, un match es un subconjunto de las aristas en el cual para todo vértice V a lo sumo una arista del match incide en V (en el match, tienen grado a lo sumo 1). Decimos que el vértice V está matched si hay alguna arista que incide en él (sino, está unmatched). El matching máximo es aquel en el que tenemos la mayor cantidad de aristas (matchear los la mayor cant. posible).



Un vértice no puede elegir a 2 amistos.

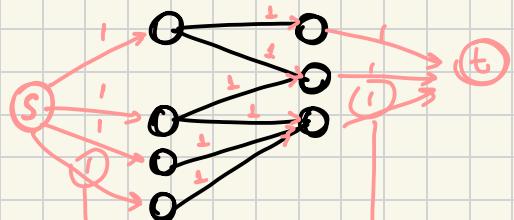
→ quiero maximizar la cantidad de vértices agarrados (que en 2 veces los amigos agarrados)

← Selección en 3 por q.

¿Cómo usamos redes para esto?

- 1) le ponemos 1 como peso a todas las aristas.
- 2) le agrego dirección
- 3) agrego (S) y (t) y veo sus pesos

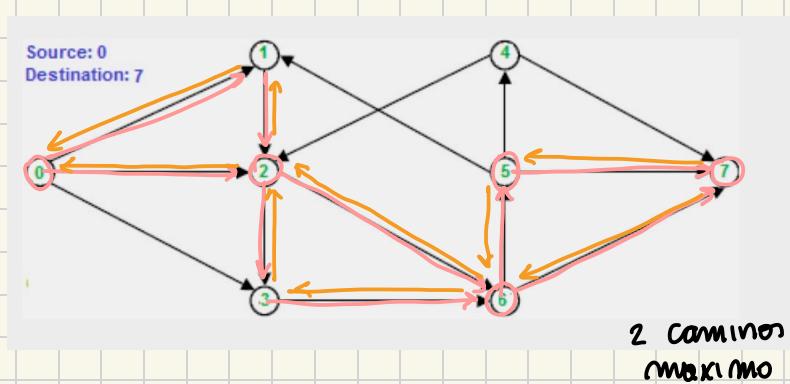
Complejidad: $O(EV)$ → Por pesos = 1, amistades mutuas



→ le doy una unidad pq' cada vértice se puede utilizar una sola vez
M así es lo que pone

② Disjoint Paths:

- Son disjuntas si no comparten aristas (pueden compartir nodos).
- dado un grafo dirigido y dos vértices s y t , encontrar el máximo número de caminos disjuntos $s-t$ en G .
- grafo no perdido : Peso = 1
- Si s, t fuente y suministro
- Usamos FF, busca en camino, el flujo total será la cantidad de caminos que encuentre, dado que una vez que pase por una arista, ésta se da vuelta, por $P=1$, y por lo tanto deja de ser utilizable.

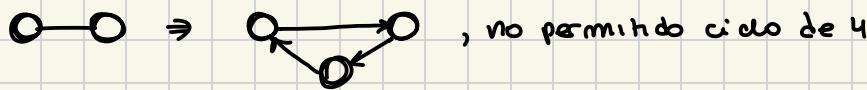


$O(E^*V)$

③ disjoint path para grafos no dirigidos

→ Convierto a dirigido.

→ le agrego un \bar{z} vértice y genero un ciclo por cada z



④ circulaciones con demanda

→ tenemos valores "fuentes" con suministro y valores "sumideros" con demandas.

Ahora cada nodo tiene una demanda ($\oplus, \ominus \neq 0$)

→ condiciones:

1) capacidad. $0 \leq f(e) \leq$ capacidad

↳ el flujo en e , con e una arista

2) demanda $f_{in}(v) - f_{out}(v) = dv$

3) la suma de los sumideros es igual a la suma de las demandas

Ej.

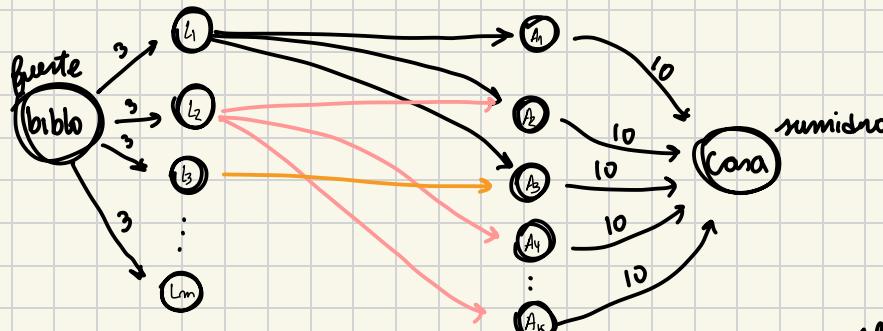
Sistema de la facultad donde cada alumno pide pedir hasta 10 libros de la biblioteca. La biblioteca tiene 3 copias de cada libro.

Cada alumno desea pedir libros diferentes.

Implementar un algoritmo que nos permita obtener la forma de asignar libros a alumnos de tal forma que la cantidad de préstamos sea máxima.

i) forman el grafo.

Vértices = libros vs alumnos. Grafo bipartito.



se dan libros →
↳ suelen tener
 $Perio = 1$

2 entidades:
algo da y algo recibe

⑤ mismo problema, pero con cotas mínimas.

Condiciones:

1) $l_e \leq f(e) \leq c_e$ debe tener mínimo l_e

2) $f_{in}(v) - f_{out}(v) = d_v$

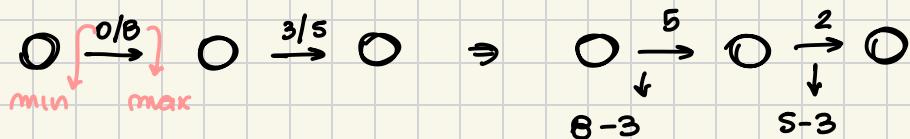
Solución:

lo transformamos en uno sin cotas inferiores:

1) definimos un hijo que cumpla las capacidades (incluyendo cotas)

2) creamos un nuevo grafo con nuevos vértices y aristas, con capacidad = $c_v - \text{consumido}$ (salvo que la demanda se cumpla, entonces no pones al vértice y sus aristas)

$$\Rightarrow \text{paso de } l_e \leq f(e) \leq c_e \text{ a } 0 \leq f(e) \leq c_e - l_e$$



Hago un DFS y me doy cuenta cuánto es lo que debo cumplir



chequeo cuál es el mínimo para que



Porque tengo la restricción
de si o si 1/1



No intenta darme el flujo máximo, sino simplemente uno válido.
una vez lo tengo genero el nuevo grafo con ese flujo ya contemplado

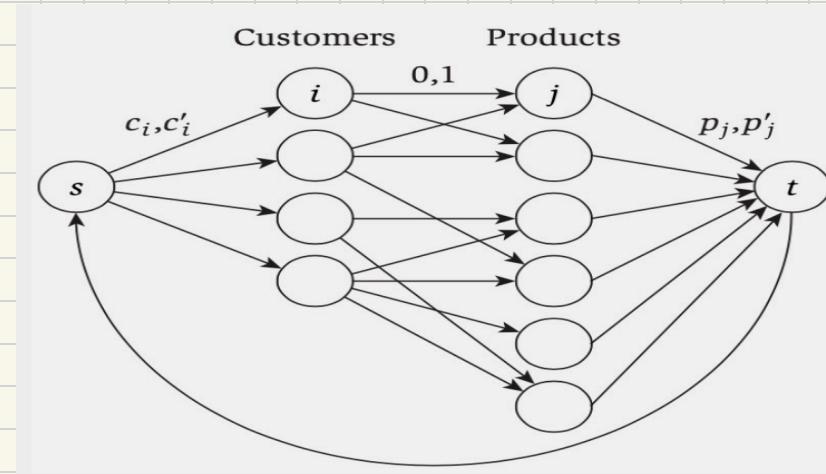


Ejemplo: diseño de encuestas.

Empresa que vende n productos, tenemos el historial de compras de cada cliente. Queremos enviar encuestas a n clientes para ver cuales son los productos que más les gustan.

Consideraciones:

- 1) Cada cliente sera consultado por un subset de productos (y siempre que el/ella hayan comprado)
- 2) La cantidad de preguntas a un cliente i debe estar entre c_i y c'_i .
- 3) Para cada producto j deben haber entre p_j y p'_j consultas.



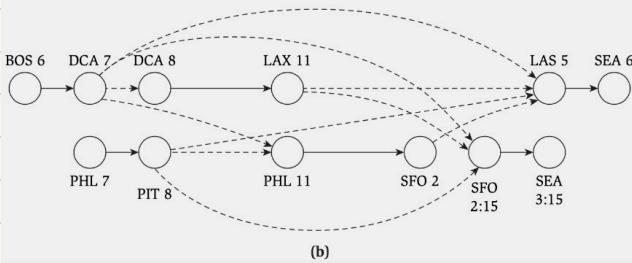
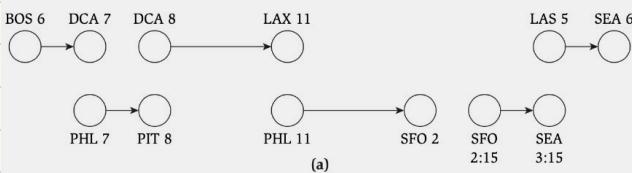
DFS se utilizará

Ejemplo: Airliner Scheduling

- queremos schedulear como los aviones van de un aeropuerto a otro para cumplir horarios y demás. Minimizar la cant. de aviones
- podemos decir que podemos unir un avión para un segmento/vuelo i y luego otro j si:
 - a) el destino de i y el origen de j son el mismo
 - b) podemos agregar un vuelo desde el destino de i al origen de j con tiempo suficiente

decirnos que el vuelo j es alcanzable desde el vuelo i si es posible unir el avión del vuelo i y despues para j .

¿Podemos cumplir con los m vuelos usando a lo sumo k aviones?



→ Vertice por aeropuerto y por horario

→ fijo, literalmente los aviones

→ los vuelos que queremos cumplir si o si \Rightarrow cota mínima y capacidad = 1 (forzamos que se usen)

→ si otro vuelo es alcanzable por las reglas anteriores, ponerle otra cinta de capacidad 1

→ la fuente tiene cintas de capacidad 1 a los origenes

→ el sumidero tiene cintas de capacidad 1 a los destinos

→ la fuente tiene demanda $-N$ y el sumidero N

Ejemplo: Project Selection

→ tenemos proyectos que podemos realizar. Algunos dan ganancia $(+)$ y otros $(-)$. Hay dependencia. prede que para hacer un proyecto i tambien tengamos que hacer un proyecto j. Sin ciclos.

→ obviamente un proyecto con ganancia $(+)$, sin otro que tenga $(+)$, se hará.

→ proyecto con ganancia $(+)$, se hará

Creamos el grafo:

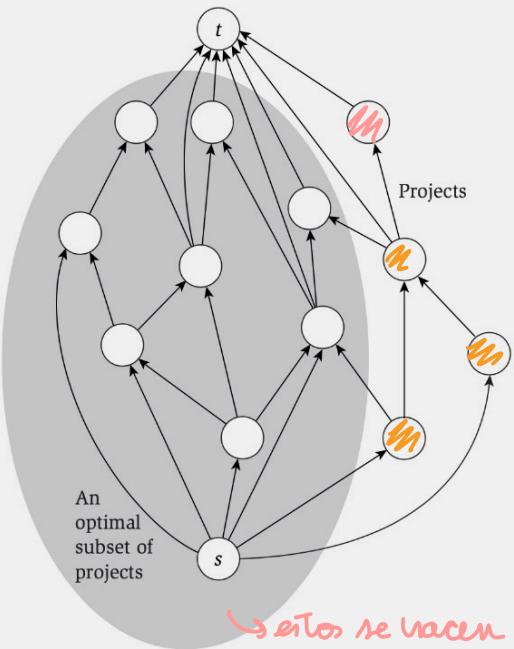
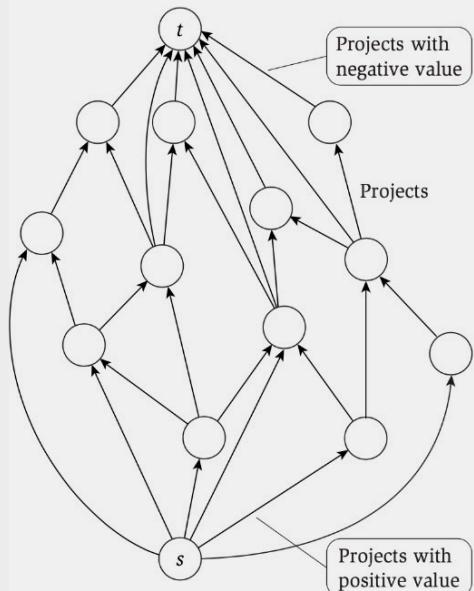
→ Vértices = Proyectos. Arista (i, j) cuando i depende de j

→ ponemos fuente, unirlo a proyectos $(+)$ con capacidad (∞) al beneficio del proyecto.

→ ponemos sumidero, unirlo los proyectos $(-)$ con capacidad - beneficio del proyecto

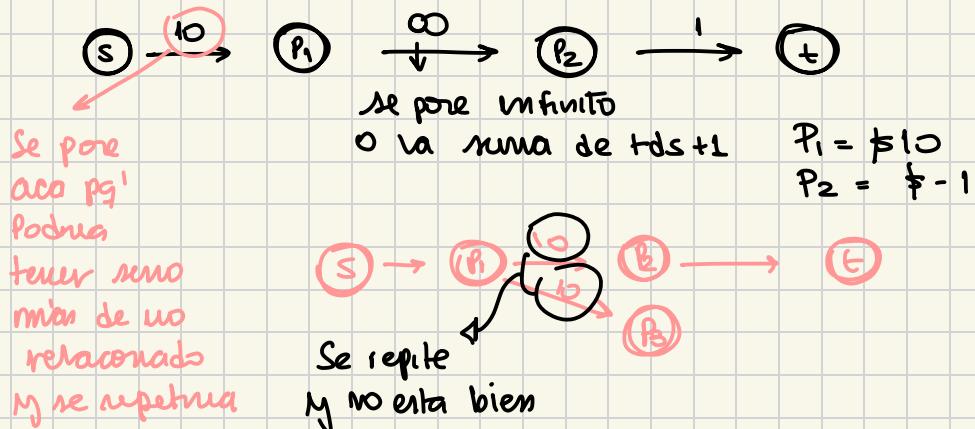
→ a las cintas de las dependencias le ponemos "capacidad infinita" (o la suma de los proyectos ponemos $+1$)

obtenemos el corte mínimo (A', B') y nos quedamos con $A' - 1$ fuente?

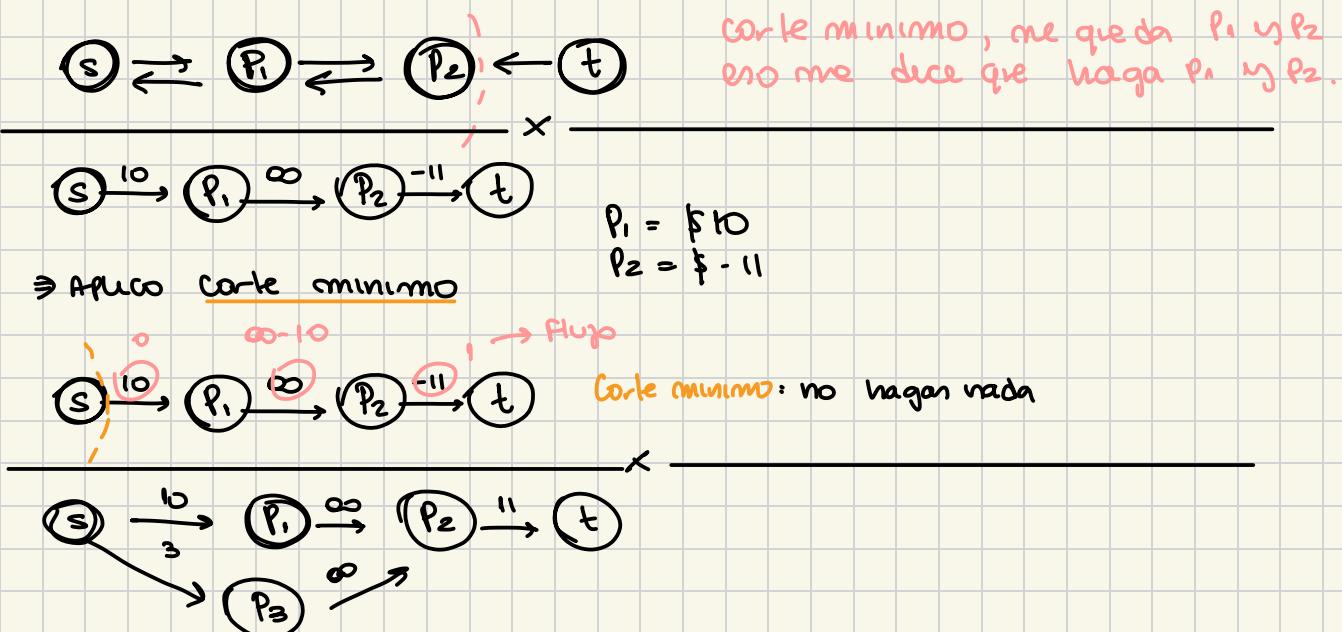


los ganancias
de estos no
compensan
la perdida
de P_1

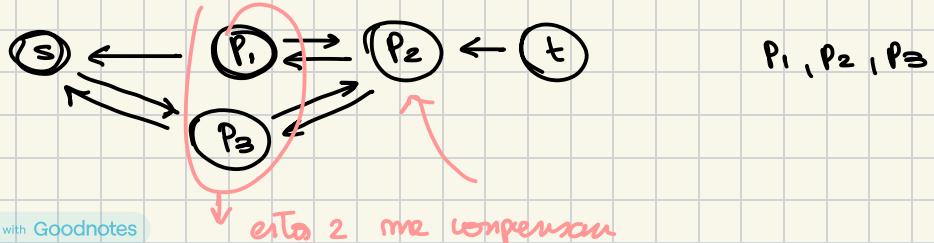
→ estos se hacen



grafo residual (voy a aplicar corte mínimo)



grafo residual



Reducir:

- Sirve para comparar la dificultad entre problemas
- Asumimos que tenemos una caja negra que resuelve x , y vamos a probar que por tener esa caja ya podemos resolver y .
la dificultad de resolver y se reduce a usar la caja que resuelve x .
y podemos decir que y es reducible polynomialmente a x

Ejemplo 1: Ordenar vs encontrar el maximo

¿ Se puede reducir encontrar el maximo de un arreglo a ordenar ?

→ Si no sabemos encontrar el maximo, ese problema se reduce a ordenar y ver una posición.

→ Entonces, ordenar es al menos tan difícil como encontrar el maximo de un arreglo

$$\begin{array}{l} \text{y} \rightarrow \boxed{x} \rightarrow \text{acceso al} \\ \text{encontrar} \quad \text{ordenar} \quad \text{ultimo} \\ \text{el max} \quad \quad \quad O(1) \\ O(n) \quad \quad \quad O(n \log n) \end{array}$$

$\max \leq_p \text{ordenar}$

- reduci encontrar el max a ordenar
- Encontrar el max, en lo sumo, tan difícil como ordenar
- =
- ordenar es al menos tan difícil como encontrar el max

¿ $\text{Ord} \leq_p \max$?

$$y \rightarrow \boxed{x} \rightarrow \max$$

le paso el arreglo con n elementos, retorna el max.

le paso el arreglo con $n-1$ elementos (saco el maximo anterior), retorna nuevo max.

y así ...

Hago $O(n)$ veces y swaps. A lo sumo sera $O(n^2)$

Como: $\max \leq_p \text{Ord}$ y $\text{Ord} \leq_p \max$ esto significa que

$$\boxed{\text{Pol}} \max \leq_p \text{Ord} \boxed{\text{Pf}} \quad \text{y} \quad \boxed{\text{Pf}} \text{Ord} \leq_p \max \boxed{\text{Pf}}$$

Ej. 2: Bipartite matching

$BM \leq_p$ maximizar reden de flujo

Ej. 3: Problema del caballo y Problema del camino hamiltoniano

$PC \leq_p CH$

↓

- construimos el grafo
- lo pasamos a la caja negra CH

¿ $CH \leq_p PC$?

No se puede convertir cualquier grafo en un tablero de ajedrez.

CH : más general

PC : recibe n : cumple cierto patrón

⇒ usualmente Específico \leq_p General

Ej. 3. Independent set y n reinas

n reinas = tablero de ajedres

n reinas ⇒ transformo en un grafo, conecto las posibles

grafo, $n \rightarrow$ $IS \rightarrow$ SOL

Ej. 4. Subset sum y Problema de la mochila

$SS(\text{elem}, w) \rightarrow$ $[\text{Mochila}] \rightarrow$ SOL

con

valores = elementos]
Personas = elementos] → mismo anegro

$w = W$

Subset sum \leq_p mochila

Cuando $x \leq_p y$ y $y \leq_p x$, entonces los problemas son lo mismo $x =_p y$

Ej 5. búsquedas I

¿ podemos reducir "Búscar un elemento en un arreglo ordenado" a "búscar un elemento en un arreglo desordenado?

$$BO \longrightarrow \text{BD} \longrightarrow \text{SOL}$$

$$BO \leq_p BD$$

se predece al reverso

$$BD \longrightarrow BO \rightarrow \text{SOL}$$

\downarrow
ordenar el
arreglo

$$BD \leq_p BO$$

otra forma:

BD - hago $n \rightarrow$ BO $\rightarrow \text{SOL}$
llamadas
a BO, con
un arreglo
de un elemento

otra forma:

BD - agarré el \rightarrow BO $\rightarrow \text{SOL}$
 \downarrow
arreglo y
busco el
elemento
"0" veces
 $O(n)$

Siempre predece hacer $P_1 \leq_p P_2$, ¿cómo?

$P_1 -$ lo resuelvo $\rightarrow P_2 \rightarrow \text{SOL}$

\downarrow
por su
solución
original
llamo una
cantidad
Polinomial de
Voces a P_2 ,
"0" Voces.

OJO: solo n la fila
Original es polinomial

Ej 6. Búsquedas II

¿ podemos reducir polinomialmente la búsqueda del máximo de un arreglo a la búsqueda de un elemento en un arreglo?

Si.

$$\max \longrightarrow \boxed{\text{Busq desord}} \longrightarrow \text{SOL}$$

\Rightarrow si llavo por cada numero posible a la caja esto sera hasta 2^{bits} y no sera polinomial.

Ej 7.

Tenemos una caja negra que eleva al cuadrado

¿Cómo podemos reducir " $a \cdot b$ " al problema de elevar al cuadrado?

$$(a+b) \rightarrow \boxed{x^2} \rightarrow (a+b)^2 = a^2 + b^2 + 2ab \Rightarrow \\ (a+b)^2 - a^2 - b^2 = 2ab \Rightarrow \\ a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$$

le monto a pasar

$$a \rightarrow \boxed{x^2} \rightarrow a^2$$

$$b \rightarrow \boxed{x^2} \rightarrow b^2$$

Mi entonces ya tengo todo lo necesario,
sabe \ominus y sabe \oplus

Detalle \Rightarrow Problema de decisión

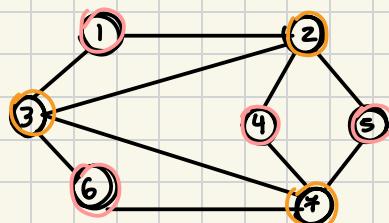
\rightarrow plantear los problemas de forma más Booleana. Por ej. independent set, el problema original es encontrar el set independiente más grande, pero vamos a plantear si existe un set de (al menos) tamaño K .

$$\text{IS}(\text{grafo}, K) \xrightarrow{\text{True}} \text{False}$$

\downarrow true de 1?, true de 2?, true de V? (con V igual a cantidad de vértices)

OBS: Con esto podemos resolver el problema más general y una búsqueda binaria para hallar el máximo K

Independent set vs Vertex cover



la \oplus de los conjuntos de IS y VC
o la totalidad del grafo.

Independent set: Conjunto de vértices que no son adyacentes entre sí

$$\text{IS}(\text{grafo}, 2) = V$$

$$\text{IS}(\text{grafo}, 5) = F$$

$$\text{IS}(\text{grafo}, 4) = V$$

n

$O(n^n)$

IS MAX = 4

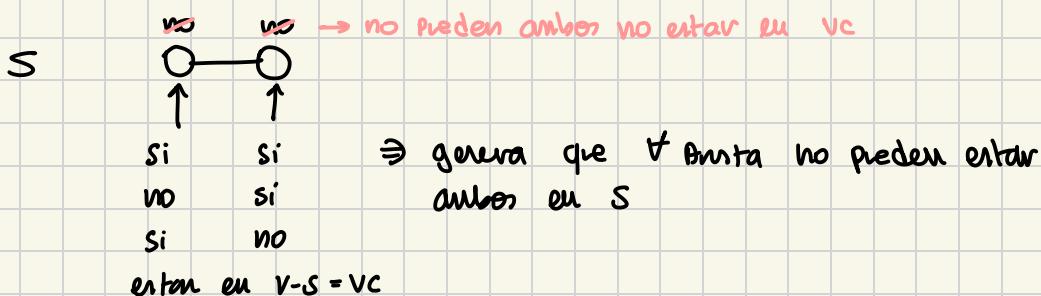
Vertex cover: Conjunto de vértices que cubren todos los aristas a lo más 2

$O(n^n)$

$$\begin{array}{ll} \text{VC}(\text{grafo}, 2) = F & \text{VC MIN} = 3 \\ \text{VC}(\text{grafo}, 4) = V & \\ \text{VC}(\text{grafo}, 5) = V & \\ \text{VC}(\text{grafo}, 1) = F & \end{array}$$

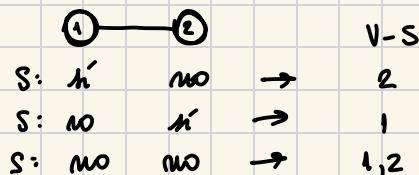
Sabemos que no podemos resolver ambos en tiempo polinomial, pero ¿y su dificultad relativa?

Teorema: S es un set independiente de g si $V-S$ es VC de g .



Devolución: Si tengo un IS S , y agarro una arista (v, w) , entonces al menos uno de ellos tiene que estar en $V-S$ pq' no pueden estar ambos en S (por ser IS) \rightarrow para para todos los aristas $\rightarrow V-S$ es VC

Suponemos que $VC(V-S)$ y tomamos un par de vértices cualquiera en S . Supongo que están unidos por una arista, pero entonces habrá una arista no cubierta por $V-S \rightarrow$ mi suposición genera el absurdo



Reducimos usando el teorema:

¿ $IS \leq_p VC$?

$IS(g, n) \rightarrow \boxed{VC} \rightarrow \text{Sol}$



no puedo pedirle
que me de todo
el VC. Le debo
Preguntar $VC(1)$
tiene soluc?o?
Y retorna VOF

Si quiero saber si $\exists IS(g, 6)$ para un grafo de 7 vértices, debo ponerte a la caja negra $\exists VC(g, 1)$

Porque $\begin{matrix} S \\ 7 \xrightarrow{\quad} V-S \end{matrix}$

gracias al teorema

Veamos:

- conocer la relación, tener el teorema
- demostrar el teorema

¿ $VC \leq_p IS$? , ni es exactamente lo mismo

$$VC(g, r) \rightarrow \boxed{IS}$$
$$IS(g, |V| - r)$$

Entonces:

cuando $IS \leq_p VS$ y $VS \leq_p IS$, son igual de difíciles.

¿Para qué sirve? $y \leq_p x$

→ Si x se puede resolver en tiempo polinomial → y también $y_{pol} \leq x_{pol}$

→ Si x NO se puede resolver en tiempo polinomial, x tampoco puede
 y no es pol. $\leq_p x$ no es pol.

Porque x es igual o más difícil

→ Si encontramos una sol polinomial para VC , entonces tenemos también para IS ($IS \leq_p VC$) y viceversa para ($VC \leq_p IS$)

• SAT y 3-SAT (satisfacción)

se tienen variables, en verdad que se cumplen ciertos postulados?

¿vera verdad que \exists un conjunto de $x_1, x_2, x_3 \dots$ (siendo T o F) que hacen que las condiciones se cumplan (todas pq's se cumplan con ands)?

ej.	Variable	Postulados	
	$x_1 = F$	- $\bar{x}_1 \vee x_4 \quad \wedge \quad \checkmark$	$\wedge = \text{and}$
	$x_2 = T$	- $x_1 \vee x_2 \quad \wedge \quad \checkmark$	$\vee = \text{or}$
	$x_3 = F$	- $x_2 \vee \bar{x}_3 \quad \wedge \quad \checkmark$	
	$x_4 = T$	- $\bar{x}_2 \vee x_3 \vee x_4 \quad \checkmark$	

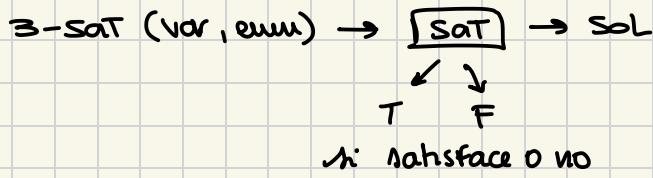
el problema para 3-SAT es que se pueden tener n variables y k postulados pero los postulados deben ser que estén compuestos por 3 variables

ej. variables Postulados

x_1	$x_1 \vee x_3 \vee x_4 \quad \wedge$
x_2	$x_1 \vee x_2 \vee \bar{x}_3 \quad \wedge$
x_3	$x_2 \vee x_3 \vee x_4 \quad \wedge$
x_4	$\bar{x}_1 \vee \bar{x}_2 \vee x_3$
x_5	

¿ $\exists\text{-SAT} \leq_p \text{SAT}$? Reducir $\exists\text{-SAT}$ a SAT

Se supone que si $p \leq p'$ $\exists\text{-SAT}$ es un caso particular de SAT



Demstración equivalente:

$\exists\text{-SAT} \leq_p \text{SAT} \Rightarrow$ trivial

$\text{SAT} \leq_p \exists\text{-SAT}?$ \Rightarrow

- cláusula con 2 términos (w, y): reemplazar $(y \vee w \vee z)$ y $(y \vee w \vee \bar{z})$. Inventar z , por cada cláusula (w, y) creamos 2, una con z y otra con \bar{z}
 - cláusula de 1 término (y): reemplazamos $\text{era}(y)$ por $(y \vee z_1 \vee z_2)$ y dos variables: $(y \vee z_1 \vee \bar{z}_2)$, $(y \vee \bar{z}_1 \vee z_2)$, $(y \vee \bar{z}_1 \vee \bar{z}_2)$
 - cláusula de 3 términos: OK
 - cláusula con +3 términos: Crean tantas variables no reales como sean necesarias
- $$(l_1 \vee l_2 \vee z_1), (l_3 \vee \bar{z}_1 \vee z_2), (l_4 \vee \bar{z}_2 \vee z_3), \dots,$$
- $$(l_{n-2} \vee \bar{z}_{n-4} \vee z_{n-3}), (l_{n-1} \vee l_n \vee \bar{z}_{n-3})$$

dato:

la notación \leq_p es transitiva.

$\exists\text{-SAT} \leq_p \text{IS}$

$\text{IS} \leq_p \text{VC}$

\Downarrow entonces

$\exists\text{-SAT} \leq_p \text{VC}$

y no necesitamos probarlo

Problemas vs algoritmos vs Cheques

- un problema puede tener \neq soluciones (algoritmos) \rightarrow tomamos el más eficiente
- dado un problema un algoritmo nos da una solución
- dado un problema y una solución, debemos poder también tener un validador.
- decimos que un certificador eficiente, si es correcto y efectúa en tiempo polinomial.

ej.

IS (g, k)

te doy S , es solución?

$S \rightarrow$ vértices \Rightarrow no pueden ser adyacentes \Rightarrow lo chequeo
└─ el tamaño debe ser $\geq k$.
└─ vértices \neq grafo

Clares de complejidad:

P: Problemas que pueden resolverse en tiempo polinomial (de forma eficiente)

NP: Problemas para los que \exists un certificador eficiente (se pueden validar en tiempo polinomial)

¿Están en NP?

→ todos los que son P son NP ($P \subseteq NP$)

¿ $P \supseteq NP$? No sabemos.

Aclaración:

¿ $3\text{-SAT} \leq_p IS$? $\Rightarrow 3\text{-SAT}$ se puede reducir a IS.

- 1) Ponemos nodos = términos ($3k$ nodos) de cada cláusula
 \downarrow
 $N =$ cláusulas
- 2) creamos triángulos por cada cláusula
- 3) Ponemos aristas en posibles conflictos (variables opuestas)
- 4) Se puede demostrar que g tiene un IS de tamaño k si el 3-SAT es satisfacible

Si hay 3-SAT hay IS

- Si es satisfacible, al menos 1 nodo de cada triángulo es True. Agarramos un vértice de cada triángulo tal que valga True. \rightarrow este es un IS (no puede generar conflictos)

Si hay IS, hay 3-SAT:

- Por cada x_i , si está en el set, va 1, si está el complemento va 0, si no hay ninguno, pone 0 alguno en 0 y alguno en 1. Si hay IS de tamaño k \rightarrow hay al menos 1 por cada triángulo
 \Rightarrow Se cumple 3-SAT

Problemas NP-completos: \rightarrow los más difíciles

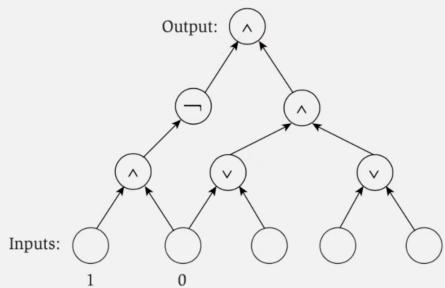
\rightarrow un problema $X \in \text{NP-Completo}$ si: (todos los problemas NP son equivalentes)
• $X \in \text{NP}$
• $\forall Y \in \text{NP}, Y \leq_p X$

\hookleftarrow 1. Si X es NP-completo, solo se puede resolver en tiempo polinomial si $P = NP$

2. Si al menos un problema NP-completo puede resolverse en tiempo polinomial, entonces todos pueden y $P = NP$

Ejemplo:

teorema de Cook y Levin: Circuit Satisfiability



Si tengo una secuencia de inputs, el output va a ser T o F.

\rightarrow Es NP-completo

Si reduzco este a otro, ese otro sera NP-completo (ese otro era NP)

\hookleftarrow Reduzco circuit satisfiability a 3-SAT

Se puede reducir circuit satisfiability a 3-SAT transformando el problema a SAT con tritos 3 terminos y luego pasara a 3-SAT como vimos antes

Así lo hacemos

- Si el nodo v es \neg con entrada desde $u \rightarrow x_v = \neg x_u \rightarrow$ agregamos cláusulas $(x_u \vee x_v) \wedge (\neg x_u \vee \neg x_v)$
- Si el nodo v es \vee con entradas $u \wedge w \rightarrow x_v = x_u \vee x_w \rightarrow$ agregamos $(x_v \vee \neg x_u), (x_v \vee \neg x_w) \wedge (\neg x_u \vee \neg x_w)$.
- Si el nodo v es \wedge con entradas $u \wedge w \rightarrow x_v = x_u \wedge x_w \rightarrow$ agregamos $(\neg x_v \vee x_u), (\neg x_v \vee x_w) \wedge (x_u \vee \neg x_w)$.
- Si el nodo v es una constante \rightarrow Agregamos (x_v)

y entonces como sabemos: $3-\text{SAT} \leq_p \text{Independent-set} \leq_p \text{Vertex cover}$

\hookleftarrow Independent-set y vertex cover tambien son NP-completos

(NP)

NP-completo

\hookrightarrow N-reducción (que se puede reducir a independent-set) ¿es en NP-completo?

\hookleftarrow No porque debemos poder reducir independent-set a N-reducción

#

N-reducción es NO NP-completo si lo hacemos por independent-set, pero si no (por otra distinción) si lo es

Problema

Reducemos ver si un número es múltiplo de otro
a ver si un elemento está en una lista (ordenada?)

\leq_p

Caja negra

Hago una lista con todos los múltiplos, chequeo si uno que no le pase
esta en la lista

Reducir independ-set a K-clique

↓ Problema

Para ver si K-clique es NP-completo

Siempre tiene que haber un si entre
ambos, pq' si redi-
dan en falso \ominus

↳ Ver si en un grafo
hay un subgrafo
completo de tamaño K

→ caja
negra

① Obtengo el grafo "complemento" → Anoto que no entran en G

② de finitos: hay un independ set de al menos K vértices si hay un clique
de al menos K vértices en el grafo complemento

¿Es K-clique un problema NP-completo?



en cuadrático, se verifica el tiempo polinomial \rightarrow es NP



al reducir un problema NP-completo \hookrightarrow reducido a K-clique (por lo menos NP)
entonces le pongo un peso a K-clique \Rightarrow el techo es que es NP

↳ que de NO ser NP-completo pq' es más que NP-completo

Estrategias para reducir en general:

↳ 3 formas:

- 1- Por equivalencias (independ-set y vertex cover, o K-clique)
- 2- de caso especial a caso general (N remas a independ-set)
- 3- Por encodes de características (3-sat \Rightarrow simple y general)

→ Ver si K -clique es NP-completo con IS.

K -clique: Ver si en un grafo,

hay un subgrafo

Completo de tamaño k

1) veo si K -clique está en NP, ¿cómo? \Rightarrow verificador polinomial

en este caso: me dan K vértices y debo chequear si forman un clique. Voy para c/u de los vértices para c/u de los demás y veo si están todos entre si ($O(K^2) \Rightarrow$ polinomial)

2) ¿Qué reducimos a qué? \Rightarrow IS $\leq_p K$ -clique: reducimos IS a K -clique

IS \rightarrow K -clique \rightarrow SOL

Recibe: ¿Este grafo tiene al menos un K -clique de tamaño K ?

3) Pensar las similitudes/diferencias entre IS y K -clique

→ Si tengo un grafo completo, el max IS es Δ y el K -clique es completo

→ Si tengo el cuadro del grafo completo (no tengo más) el IS es de todos, tamaño (V), y el K -clique es de 1

¿Cómo lo resuelvo finalmente?

① Obtengo el grafo "complemento" \Rightarrow Anular que no están en G

② de nuevo: hay un independ set de al menos K vértices si hay un clique de al menos K vértices en el grafo complemento

→ ¿Es K -clique un problema NP-completo?



en cuadrático, se verifica en tiempo polinomial \Rightarrow es NP



al reducir un problema NP-completo y reducirlo a K -clique (por lo que entonces le pasamos un paso a K -clique \Rightarrow el hecho es que es NP)

↳ que de NO ser NP-completo pq' es más que NP-completo

Estrategias para reducir en general:

↳ 3 formas:

- 1- Por equivalencias (independent-set y vertex cover, o K-clique)
- 2- De caso especial a caso general (N remas a independent-set)
- 3- Por encodar las características (3-sat \Rightarrow simple y general)

Veamos Programación Lineal

→ es la PL en NP? \Rightarrow Si Pg' reemplaza Minto

→ es NP-completo? no.

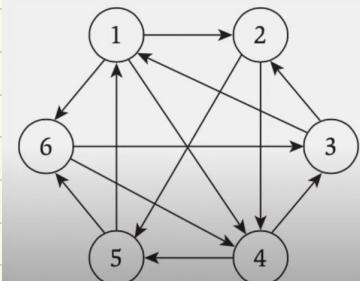
→ M PL entera?, si Pg' puedo usar PL entera para resolver, por ej. LS.

Otro tipo de problemas:

→ Vertex cover e independent-set \Rightarrow problemas de selección de subsets

~~Nuevo~~
→ Problema de permutaciones: elegir un orden (ej. Scheduling)

Problema del viajante (TSP)



→ no debe ser necesariamente no dirigido.

→ no debe cumplir si o si la desigualdad triangular

→ lo planteamos como un problema de decisión: ¿Existe un camino de costo/distancia como máximo D?

Ciclo Hamiltoniano: = CH

↳ ¿Existe algún camino que pase por todos los vértices?

Reducir ciclo hamiltoniano al problema del viajante (TSP)

TSP es NP

Le doy el grafo y el costo a TSP y

se devuelve True o False y es

Respuesta que te da que ayudan

a saber si existe o no un ciclo hamiltoniano



A los aristas que no existen en G le tengo que agregar los aristas faltantes como "2" p₁
esa no la va a tocar

↳ Existe un ciclo hamiltoniano en el grafo original si existe un camino TSP en el nuevo grafo de a lo sumo (y exactamente) n = cantidad de vértices

↳ no puede ser menor que n

↳ n es mayor que n y haber usado una arista de peso 2 que no existe en el original \rightarrow no hay ciclo tf

⇒ Pero esto es NP completo? \Rightarrow No se, pero se puede reducir un problema NP-completo a TSP entonces TSP es NP-completo, pero aun así no se si el ciclo hamiltoniano lo es.

- TSP es al menos tan difícil como CH
- CH es a lo sumo tan difícil como TSP
- debes convertir CH a algo de la forma TSP

$CH \leq_p TSP$

CH más particular
de TSP

CAJA NEGRA

↳ no repite vértices

↳ supones que tiene todos los aristas, esta cumplido

PERO: Si Reduzco un Problema NP-completo A un Ciclo Hamiltoniano, entonces Ciclo Hamiltoniano es NP-completo y TSP también por el mismo proceso que de NP-completo a Ciclo Hamiltoniano

- CH es al menos tan difícil como NP-comp.
- NP-comp. es a lo sumo tan difícil como CH

NP-completo \leq_p CH

CH \leq_p TSP

Reducimos 3-SAT a CH.

Pq' 3-SAT \Rightarrow cuando no se hace bien 3-SAT es buena idea pq' sus combinaciones son bastante simples

¿ 3-SAT \leq_p CH? \rightarrow 3-SAT es a lo sumo tan difícil como CH
 \rightarrow CH es al menos tan difícil como 3-SAT

¿ Común Hamiltoniano es NP-completo?

• Ciclo H \leq_p Camino H \rightarrow reduzco ciclo a camino

- 1) agarramos un vértice al azar y lo reemplazamos por V' y V'' . todos los amigos salientes de V ahora salen de V' , todos los amigos entrantes a V , ahora entran a V'' .
- 2) \exists ciclo H. en el original si \exists camino H. en el nuevo (ese camino necesariamente arranca en V' y termina en V'')

Ej III.

Dado 3 números a, b y c . Escribir un algoritmo que determine si $a+b=c$

1) Es NP? \rightarrow Pq¹ se puede resolver y verificar en tiempo polinomial
es P? \rightarrow

2) Cómo reducirlo a TSP? $\text{ento} \leq_p \text{TSP}$

2 vértices (a y b), M con TSP me fijo el costo en a lo sumo C , si es así entonces me fijo M ($a, b+1$) da mayor o igual a $C \Rightarrow$ si da falso no es

No dice nada esta reducción

Reducir TSP a este problema:

TSP \leq_p este: no.

Problemas de particionamiento

↓
donde buscamos dividir un conjunto de obj/datos/líneas.

Coloreo de grafos

\rightarrow Con $K=2 \rightarrow$ Grafo bipartido

\rightarrow Con $K=3 \rightarrow$ NP-comp.

① Esto es NP, si, vértice por vértice v_j veo que los colores de los adyacentes sean diferentes

② Vamos a la confiable 3-SAT. $3\text{-SAT} \leq_p \text{Coloreo}(3)$

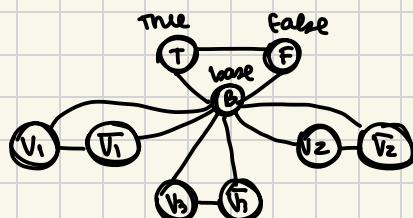
$3\text{-SAT} \rightarrow \boxed{\text{Coloreo}(3)} \rightarrow \text{SOL}$

Paso:

→ Por cada variable pongo un vértice, uno los vértices con sus complementos. Si o si colores \neq para vértice v_j su complemento, igual a T o F, T = color 1, F = color 2 me queda un color libre \Rightarrow agrego un vértice artificial (base/cable a tierra) que tenga este color

→ Unes a cada variable v_j complemento con base, para formar triángulos.

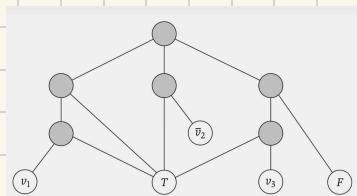
Por ahora tengo así



Varón por las expresiones:

Terror $x_1 \vee \bar{x}_2 \vee x_3$

→ al menos 1 debe ser T → al menos uno debe tener el color T



Con esta configuración → es satisfacible si el grafo es coloreable por 3 colores

dominated set: dado un grafo, un subconjunto en el que todos los vértices o bien, están en el conjunto, o son adyacentes a alguno de los vértices del conjunto

Problema de dominación: dado un grafo y un número k. ¿Existe un conjunto de a lo más k vértices que sea dominante?

en un grafo completo tengo como mínimo un dominating set

dúctiles que es NP-completo:

1) está en NP? ✓ Agarramos los del dominating set, le agregamos adyacentes y agregamos que estén todos los del grafo $O(V+E)$

2) parecido a vertex cover.

Reducir vertex cover a dominating set

$VC \leq_p DS$ VC no parece ser más difícil que DS

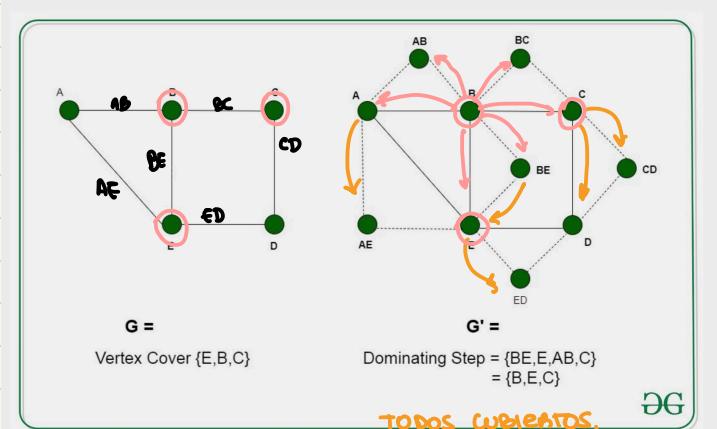
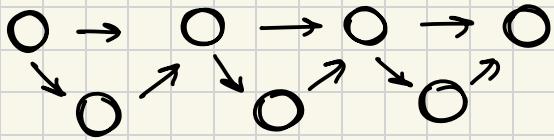
$VC \rightarrow DS \rightarrow SOL$

Por cada arista tengo que tomar a lo sumo un vértice
ej:

$O \rightarrow O \rightarrow O \rightarrow O \Rightarrow$ 2 para ds (el primero y el último), pero para VC valido necesitamos los 2 del medio.

↓

ds debe elegir alguno de los 2 extremos
¿Cómo? ⇒ por cada arista agregamos un vértice que esto unido a los 2 que usaba originalmente
⇒ op! dejar la ruta igual



1. Tenemos nuestro grafo original G , y vamos a construir otro grafo. Este nuevo grafo va a tener los mismos vértices que el original.
2. Aparte, tenemos un vértice por cada arista del grafo original. Unimos los vértices adicionales con los vértices que originalmente unían (como poner un vértice en el medio).
3. Ahora resulta que si el grafo original tiene un vertex cover de tamaño K , entonces esos mismos vértices forman un dominant set en el nuevo grafo \rightarrow si hay un dominant set de tamaño k , hay un vertex cover de tamaño $k \rightarrow$ pudimos reducir VC a DS \rightarrow DS es NP Completo.

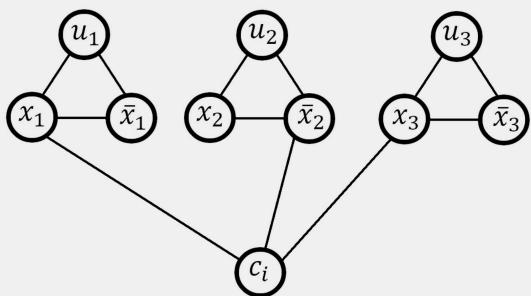
demostración que dominat set es NP-completo por 3-SAT

$$3\text{-SAT} \rightarrow \boxed{\text{DS}} \rightarrow \text{sol}$$

$$3\text{-SAT} \leq_p \text{DS}$$

→ convierto 3-SAT en un grafo

- 1) un vértice por cada variable y cada complemento
- 2) Ahora suento un vértice más y con este y \nearrow hago un triángulo
- 3) Entonces tengo n variables = triángulos
- 4) Si quiero un DS de tamaño $W = n$ -variables, va a tener que elegir un vértice de cada triángulo. Sea n como cota superior para seras si o si 1 por triángulo. Si o si debe ser n pq' si no no hay DS.
- 5) Agregar un vértice por cláusula
- 6) unes los vértices de tipo cláusula a las variables que la integran
- 7) hay DS de a lo sumo $W = n$ vértices si el problema de 3-SAT se puede satisfacer



Hay Dominating Set → Hay 3-SAT

Tenemos n triángulos donde un vértice (extra) sólo se une a 2 de estos \rightarrow al menos uno de esos 3 debe quedar seleccionado (similar a 3-SAT \leq_p Independent Set), pero no puede haber más de 1 por triángulo porque podemos seleccionar hasta n , sino es imposible dominar las variables extra.

Luego, para que los vértices de cláusulas queden dominados, alguna de sus variables deben quedar en el conjunto \rightarrow si hay Dominating Set, esta asignación permite que todas las cláusulas del 3-SAT sean satisfechas.

Hay 3-SAT → Hay Dominating Set

Si tenemos una asignación de variables/complementos valiendo true/false, puedo poner los que están en true como parte del dominating set, y eso va a tener que dominar uno por triángulo, y necesariamente va a haber al menos una variable por cláusula dominándola.

Problema con NP:

¿Cómo demostramos que no hay una solución exacta?

Complemento a su problema: todo lo que sea solución en el problema, no sera solución en su complemento y viceversa
 P = Polinomial
 X = Problema

si $x \in P \rightarrow \bar{x} \in \bar{P}$: en el not del problema
 y si $x \in NP \rightarrow \bar{x} \in NP$?

¿ $NP = CO-NP$? CO = complemento

no sabemos, y se cree que no $NP \neq CO-NP \rightarrow P \neq NP$

se demuestra por método inducción ...

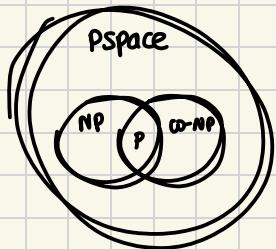
PSPACE:

problemas que se resuelven con un algoritmo que consume una cantidad de espacio polinomial

- 1)imergenont està en Pspace? \Rightarrow si, mas o menos $O(n)$
- 2)fibonacci " \Rightarrow si, a lo sumo $O(n)$
- 3)P està en Pspace! \Rightarrow si, pq' si fuera Exponencial debería pedirlo. (maderado)

"El espacio se puede neutralizar, el tiempo no"

$NP \subseteq Pspace$: NP està contenido en P-space



Problemas en Pspace que "no son NP".

\rightarrow Vamos a buscar Pspace-completos

Cuantificación (\forall SAT)

Volveremos a 3-SAT: teneremos n variables, y una formula booleana que es una disyunción de conjunciones de 3 términos

3-SAT pregunta. $\exists x_1 \exists x_2 \dots \exists x_{n-2} \exists x_{n-1} \exists x_n \Phi(x_1, \dots, x_n) ?$

planteamos Quantified 3-SAT = (\forall SAT) : $\exists x_1 \forall x_2 \dots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \dots, x_n) ?$

\nwarrow en pspace-completo

Resolución:

si tenemos 2 amigos en cada paso de la recursividad: $s(n) = 2 s(n-1) + p(n)$

pero podemos neutralizar $\rightarrow s(n) = s(n-1) + p(n) \leq n p(n) \rightarrow \forall$ SAT està en Pspace

Generalizaciones:

- Tengo un conjunto C_0 de condiciones iniciales
- Tengo un conjunto C^* de condiciones finales al que queremos llegar
- Tengo un conjunto de operaciones/operadores O_1, \dots, O_k , con cada Operador i con pre-requisitos P_i , una "add-list" A_i y una "delete list" D_i .
- ¿Es posible aplicar una secuencia de operaciones desde C_0 tal que lleguemos a C^* ?

Juego de la fana:

→ tengo un grafo implícito para el pasaje.

(x, y) :

- $(5, y) \rightarrow$ lleno la 1^a de SL
 - $(x, 3) \rightarrow$ lleno la 2^a de BL
 - $(0, y) \rightarrow$ vaciar la 1^a
 - $(x, 0) \rightarrow$ Vaciar la 2^a
 - $(\max(x - (3-4), 0), \min(y + x, 3))$
 - $(\min(x + y, 5), \max(y - (5-x), 0))$
- Pasajes de una a la otra

→ Quiero pasar de $(0, 0)$ a $(4, 0)$.

uso BFS

$(0, 0) \rightarrow$

$(1, 0); (0, 1) \rightarrow (2, 1); (1, 1); (0, 2) \rightarrow \dots \rightarrow (4, 0)$

Consecuencias:

$(0, 0) \rightarrow (1, 0) \rightarrow (2, 0) \rightarrow (3, 0) \rightarrow (4, 0)$

Podemos decir que hay un grafo (dirigido) **implícito** de nodo = todas las 2^n configuraciones posibles (todos los posibles subsets de C), con arista de C' a C'' si existe un operador que nos lleve de C' a C'' .

En el marco de nuestro problema, ... ¿Existe un camino de C_0 a C^* ?

En el peor de los casos tenemos que pasar por todos los estados posibles ($\rightarrow O(2^n)$)

[Si fueran de un tamaño polinomial, estaría en NP porque validarlos se hace en tiempo polinomial]

→ es exponencial \Rightarrow no está en NP, no está en Pspace tampoco