Teoría de Algoritmos (TB024, 75.29, 95.06) – Curso Buchwald - Genender

1.er recuperatorio de examen parcial – 24/06/2024

1. Juan es ambicioso pero también algo vago. Dispone de varias ofertas de trabajo diarias, pero no quiere trabajar tres días seguidos. Se tiene la información de la ganancia del día i (G_i), para cada día. Implementar un modelo de **programación lineal** que maximice el monto a ganar por Juan, sabiendo que no aceptará trabajar tres días seguidos.

2. Implementar un algoritmo greedy que permita obtener el mínimo del problema del viajante: dado un Grafo pesado G y un vértice de inicio v, obtener el camino de menor costo que lleve a un viajante desde v hacia cada uno de los vértices del grafo, pasando por cada uno de ellos una única vez, y volviendo nuevamente al origen. Se puede asumir que el grafo es completo. Indicar y justificar la complejidad del algoritmo implementado.

   ¿El algoritmo obtiene siempre la solución óptima? Si es así, justificar detalladamente, sino dar un contraejemplo. Indicar y justificar la complejidad del algoritmo implementado. Justificar por qué se trata de un algoritmo greedy.

3. Coty cumplió años ayer y está organizando su festejo. En dicho festejo, va a dar unos regalos. Son regalos geniales, que van a dar que hablar luego del festejo. Eso es justamente lo que desea ella: que todos aquellos invitados que se conozcan entre sí, luego de terminado el evento hablen del regalo que recibió uno, o bien el otro. ¿El problema? Coty está invitando a n personas, pero no tiene presupuesto para comprar n regalos, sino tan sólo k.

   El problema del cumpleaños de Coty puede enunciarse como: Dada la lista de n invitados al cumpleaños de Coty, un número k, y conociendo quién se conocen con quién (ej: una lista con los pares de conocidos), ¿existe una forma de asignar a lo sumo k personas para dar los regalos, de tal forma que todos los invitados, al hablar luego con quienes se conozcan, puedan hablar del regalo que obtuvo uno o bien el otro?

   Demostrar que *el problema del cumpleaños de Coty* es un problema NP-Completo.

4. Implementar un algoritmo `potencia(b, n)` que nos devuelva el resultado de b^n en tiempo $\mathcal{O}(\log n)$. Justificar adecuadamente la complejidad del algoritmo implementado. *Ayuda*: recordar propiedades matemáticas de la potencia. Por ejemplo, que $a^h \cdot a^k = a^{h+k}$. ojo con exp impares, no?

5. Dado un número n, mostrar la cantidad más económica (con menos términos) de escribirlo como una suma de cuadrados, utilizando programación dinámica. Indicar y justificar la complejidad del algoritmo implementado (cuidado con esto, es fácil tentarse a dar una cota más alta de lo correcto). Implementar un algoritmo que permita reconstruir la solución.

   Aclaración: siempre es posible escribir a n como suma de n términos de la forma 1^2, por lo que siempre existe solución. Sin embargo, la expresión $10 = 3^2 + 1^2$ es una manera más económica de escribirlo para $n = 10$, pues sólo tiene dos términos.

6 (SFIS)

1	2	3	4	5
B	B-	B	M.	B=

IOAKI LLORENS

1er Recuperatorio de Parcial

24/6/24

1) Juan no quiere trabajar 3 días seguidos, tenemos ganancia de cada día G_i

Para definir el modelo de programación lineal primero definimos las variables:

$\forall i : Y_i \leq 1, \geq 0$, Y_i variable entera (Es booleano.)

→ ∴ tenemos n variables booleanas y representan si Juan trabaja ese día o no

Luego necesitamos aplicar las restricciones de que no quede trabajar 3 días seguidos, definimos:

$Y_i + Y_{i+1} + Y_{i+2} < 3$ → hacemos que no se puedan elegir tres variables seguidas

Y por último, como queremos maximizar el monto que gana Juan, definimos el problema como:

$$\text{problema} = \max \sum_{i=0}^{n} Y_i \cdot G_i$$

∴ Si un día Juan decide trabajar la variable Y_i se marca como 1 y se ve reflejado en la suma para el máximo.

Entonces, para resumir, el modelo de PL queda:

- $\forall i : Y_i \leq 1$, $Y_i \geq 0$, Y_i variable entera (Booleana)
- Para todas las variables Y_i definimos: $Y_i + Y_{i+1} + Y_{i+2} < 3$
- Definimos el problema como $\max \sum_{i=0}^{n} Y_i \cdot G_i$ ✓

B

IÑAKI
LLORENS

2) Algoritmo Greedy para el problema del viajante en grafo pesado G y vertice de inicio V

B.

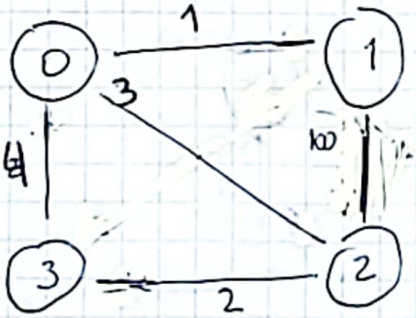
El algoritmo que voy a definir tiene la siguiente estructura:
- me paro en el vertice V
- miro las aristas de mis adyacentes
- elijo moverme al vertice que no haya visitado que me cuesta menos

```
def problema_del_viajante(grafo, Vinicio):
    Visitados = set()  , camino = set()
    vactual = Vinicio
    while (len(visitados) < len(grafo)):
        Vpeso minimo = None , peso minimo = None   deberas inicializarlo en INF no en None.
        for w in grafo.adyacentes(vactual):
            if pesoarista(vactual, w) < pesominimo AND w not in visitados:
                peso minimo = pesoarista(vactual, w)
                Vpeso minimo = w
    Visitados.add(Vpeso minimo), camino.add((vactual, Vpeso minimo))
    vactual = Vpeso minimo
    return camino
```

Hay que agregar vértice para el camino. cerrar

Es Greedy porque en cada paso buscamos un optimo local, esto es moverme al vertice y suponer un menor costo, para conseguir la solucion optima general.
La complejidad del algoritmo es O(V+E) (es grafo completo) porque siempre vamos a estar recorriendo todos los vertices y viendo todas sus aristas.

El algoritmo no siempre consigue la solucion optima, contraejemplo:



Lo que haria el algoritmo en esta situacion, partiendo por el 0 seria moverse al (1), ya que la arista a (2) vale (3), la del (3) vale 4 y la del (1) vale (1), pero despues no encontraria con que el proximo coste para ir al (2) es 100. Una mejor solucion optima iria al (2) o al (3) aunque cueste 3, y asi se ahorraria pasar por la arista de coste 100.

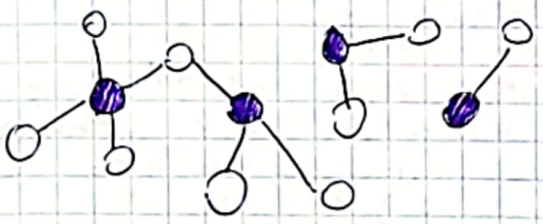
∴ mi algoritmo da: [(0,1),(1,2),(2,3),(3,0)], coste = 107

cuando el optimo es: [(0,2),(2,3),(3,0),(0,1)], coste = 10

IJAKI
LLORENJ

3) PROBLEMA DEL CUMPLEAÑOS DE COTY

Invito a n personas y tiene K regalos, $K \leq n$
Quiero una forma de asignar a los invitados de modo que todos los invitados tengan con quien hablar luego



con $K = 4$

Tenemos lista de quienes se conocen con quien, transformo en grafo.

a) Para demostrar que el problema del cumpleaños es NP-Completo, primero tengo que demostrar que está en NP. Que un problema esté en NP significa que existe un verificador que en tiempo polinomial me dice si una respuesta es válida o no.

```
def verificador(grafo_personas, solucion):
    visitados = set()
    for v in solucion:
        for w in grafo_personas.adyacentes(v):
            visitados.add(w)
        visitados.add(v)
    return len(visitados) == len(grafo)
```

MAL. El problema de CC no recibe un grafo sino una lista de pares conocidos.

Porque con la solución (personas que recibieron regalo) debería poder llegar a todos los vértices del grafo.
El verificador es polinomial ($O(V+E)$) ∴ el problema está en NP

b) Debo ahora demostrar que el problema es NP-Completo. Hago esto reduciendo un problema que sé sabido que está en NP-Completo al problema del cumpleaños. Si puedo demostrar esto, entonces demuestro que el problema de Coty es NP-Completo ya que es lo mismo que decir que el problema es al menos tan difícil como un problema NP-Completo.

Entonces reduzco Vertex Cover al Problema del Cumpleaños

→ busco demostrar $VC \leq_p Cumple$

Esto significa, conceptualmente, que puedo suponer que el problema del cumple está resuelto con una caja negra, y ~~[illegible]~~ intento usar esta caja negra para resolver Vertex Cover.

Busco VC → Cumple → Solución VC

Elegí Vertex Cover porque es un problema NP-Completo, que es bastante similar al problema del Cumple.
Vertex Cover recibe un grafo y un número K, y devuelve un conjunto de vértices de tamaño K que cubren todos los aristas del grafo.
En su versión de decisión, dice si es posible encontrar un conj de vértices de tamaño K que cubran todas las aristas.

Como se puede ver, VC es muy parecido al problema del Cumple porque en este también se trata de volver una lista de personas (vértices) con tal que, si se les asigna un regalo, puedan hablar con todas las demás personas invitadas (vértices que cubren todas las aristas)

Para resolver entonces Vertex Cover usando el problema del cumpleaños, tenemos que:

a) Transformar la entrada de VC. Cumple recibe una lista con pares conocidos, por esto recorremos el grafo de VC y juntamos las aristas de la forma: (V, w). El K de VC es igual al K del Cumple.

b) En su versión de decisión, el problema del Cumple devuelve si existe una forma de asignar K personas para dar regalos, esto es lo mismo que decir si existe o no un Vertex Covers de tamaño K por lo que no hace falta transformar la salida.

∴ con esto demostramos que se puede reducir un problema NP-Completo al problema del cumpleaños, lo que significa que este es un problema NP-Completo.

¿Por qué el verificador de CC recibe un grafo entonces?

IJAKI
LLORENS

4) Algoritmo potenca (b, n) que nos devuelva b^n en tiempo $O(\log n)$

```
def potenca(b, n):
    IF n == 0: b^0 = 1!!
        return b
    IF n%2 == 0:
        return potenca(b, n/2) · potenca(b, n/2)
    else:
        return potenca(b, (n+1)/2) · potenca(b, n/2)
```

¿Por qué llamar dos veces?

```
x = potenca(b, n/2)
return x · x
```

Usamos la propiedad de $a^n \cdot a^k = a^{n+k}$ de las potencias para implementar el algoritmo que usa la técnica de división y conquista para devolver la potencia de b^n en tiempo logarítmico.

La complejidad del algoritmo es $O(\log n)$ y se puede demostrar usando el teorema maestro

Lo que hace el algoritmo es ir dividiendo por cada n, el problema en dos, para poder calcular la potencia y es por esto que la complejidad es logarítmica.

↓

Dividir en dos pero resolver dos veces.

$$b^{\frac{n+1}{2}} \cdot b^{\frac{n}{2}} = b^{\frac{2n+1}{2}} = b^{n+\frac{1}{2}} \neq b^n$$

IJNKI
LLORENS

5) Dado un numero n mostrar la cantidad con menos terminos de escribirlo como suma de cuadrados usando PD

```
def_min terminos (n):
    OPT = [None] * (n+1)
    OPT[0] = 0
    OPT[1] = 1
    for i in range (2, n+1):
        j = 1
        while (j*j <= i):
            OPT[i] = min (OPT[i], OPT[i - j*j] + 1)
            j += 1
    return OPT
```

Para explicar el algoritmo me gustaría primero hacer un seguimiento, con el ejemplo de n = 9:

Sabemos nosotros que el OPT de 9 debería ser 1 solo término, 3^2, así es como construye la solución mi algoritmo propuesto:

$n = 9 \longrightarrow$ j=1, min(None, OPT[9 - 1·1] + 1) = min(None, OPT[8] + 1)

Y en este punto del algoritmo, o porque nos lo dice Messi (Fc btw) que OPT[8] = $2^2 + 2^2 = 2$

∴ con j = 1, OPT[9] = 3 (y es lo mismo que $2^2 + 2^2 + 1$)

- j=2, min(3, OPT[9-4] + 1) = min(3, $\underbrace{OPT[5]}_{= 3 = 2^2 + 1 \cdot 1}$ + 1) = 3
- j=3, min(3, OPT[9 - 9] + 1) = 1
  y como el OPT de 0 es 0, da 1

∴ termina dando que el OPT[9] = 1

Y esto quiere decir que el algoritmo encontró un numero que puede ser escrito solamente con una potencia

Podemos hacer el seguimiento también para n = 10

→ - j=1, min(None, OPT[10 - 1] + 1) = 2
- j=2, min(2, OPT[10-4] + 1) = min(2, $\underbrace{OPT[6]}_{2^2 + 1^2 + 1}$ + 1) = min(2, 5) =
- ... y se termina quedando con el 2

La ecuación de recurrencia entonces para el algoritmo es:

$$OPT[n] = \forall j \not\in j \cdot j \leq n : \min\left(OPT[n], OPT[n-j^2]+1\right)$$ ✓

Para todo j tal que $j^2 \leq n$, se busca el mínimo entre el OPT actual y el mínimo calculado anteriormente. De esta manera usamos el método de programación dinámica para ir construyendo la solución a medida que avanzamos.

La complejidad del algoritmo es $O(N)$ ya que para construir la solución iteramos sobre este número N, uno podría llegar a pensar también que el while ($j^2 \leq i$) afecta en la complejidad, pero estoy eligiendo descartarlo ya que la cantidad de iteraciones que realiza el while son relativamente pocas comparadas con recorrer todos los números de 1 a N. → Es $O(N \cdot \sqrt{N}) = O(N^{1.5})$

Se podría decir también que el while afecta $\sqrt{N}$ así quedaría $O(N + \sqrt{N})$

Por último implementamos el algoritmo de reconstrucción:

while adentro del for se multiplica la complejidad.

```
def min_terminos(n):
    OPT = -min_terminos(n)
    solucion = []
```

No llego a hacer la reconstrucción, pero sería ir viendo el arreglo de óptimos y calculando con la misma fórmula de la ecuación de recurrencia para ir buscando cuáles son los términos y componer la solución

B=