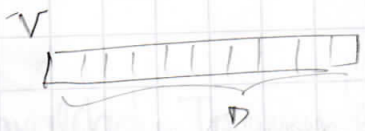
2015年11月30日 (月)

Aux # 11

P1



1) Usando $O(D \log D)$ bits adicionales

Ingredientes

* Un arreglo $U[1, D]$
* Un stack $S[1, D]$ (en forma de arreglo) + un puntero al tope, t.

} estamos en un modelo de muy bajo nivel donde puede haber slots con basura

* El valor de inicialización

```
Init(V, D, v): # inicializar de forma "lazy"
    t ← 0
    I ← v

Write(V, i, v)
    si V[i] no está
```

68, ≥53

2) Usar $\theta(D)$

Idea: usemos un arreglo B de D bits, tal que

$B[i] = 1 \Leftrightarrow V[i]$ ha sido modificado.

Claramente nos encontramos con el problema de inicializar B.

Para ello, consideremos B como un arreglo $B'[1, D']$

con $D' = \frac{D}{w}$ ← tamaño de palabra del procesador

Supongamos $w \geq \log D$

Usamos la técnica anterior para inicializar B' en 0s

$\Rightarrow$ esto usa $2D' \log D' = \frac{2D}{w} \log D' \leq \frac{2D}{\log D} \log D \leq 2D$ bits

Junto a B, esto usa $3D$ bits ($\in \Theta(D)$)

P2 "Búsqueda exhaustiva inteligente"

- Branch & Bound

1) Backtracking

- Mirando parte de una solución puedo descartar gran parte del espacio de soluciones. ← acotar su costo

Ej: si tengo $(x_1 \vee x_1 \vee x_1)$, ya que sé que $x_1 = F$ no lleva a nada.

Para Backtracking necesitamos un "test" que mire un subproblema y diga:

- $\nexists$ solución
- Encontré una solución
- No sé

Luego el backtracking se ve algo así:

- Dado un problema P_0
- $S \leftarrow \{P_0\}$ es el conjunto de subproblemas activos.
- Mientras S no esté vacío:
  - escoger un subproblema $P \in S$ y quitarlo de S.
  - expandir P en $P_1, \dots, P_k$ subproblemas menores.
  - para cada P_i:
    - Si test(P_i):
      - encuentro sol global $\Rightarrow$ lo retorno
      - no encuentra solución $\Rightarrow$ descarto P_i
      - else se agrega P_i a S.
- Se anuncia "$\nexists$ solución"

2) Branch & Bound $\Rightarrow$ para optimización (s.p.g. minimización)
   - mirando parte de una solución puedo <u>acotar</u> su costo

Aquí necesitamos una función lower-bound que acote <u>por abajo</u> el costo de cualquier solución completa que parta de una solución incompleta dada.

vendedor viajero

Ej: TSP. Si tenemos un tour parcial que pasa por un cjto S de nodos, completarlo necesita un camino en $(V-S)$ y arcos de $(V-S)$ a a y b (el inicio y fin del tour)

Luego Branch & Bound se ve algo así:

- Dado un problema P_0
- best $\leftarrow \infty$
- $S \leftarrow \{P_0\}$ es el conjunto de subproblemas activos
- Mientras S no esté vacío:
  - escoger un subproblema $P \in S$ y <u>quitarlo</u> de S.
  - expandir P en $P_1, \dots, P_k$ soluciones parciales
  - para cada P_i:
    - $\longrightarrow$ siguiente

- Para cada P_i
  - * Si P_i es una sol. completa ⇒ actualizar best.
  - * Si no,
    si lower-bound(P_i) < best.
    agregar P_i a S
- Se retorna best.

⇒ costo de completar S ≥ arco más liviano de a a (V-S) + arco más liviano de b a (V-S) + pesos del mínimo subgrafo que conecte todos los nodos de (V-S)

MST

2015年12月1日 (火)

ESQUEMA DE APROXIMACIÓN $(1+\varepsilon)$

PROBLEMA DE LA MOCHILA NP-COMPLETO, se deja aproximar (KNAPSACK, subset-sum)

Versión de decisión: $x_1, x_2, \ldots, x_n > 0$ (bits para repr t puede ser 32 bits)
tope t. $|input| = \theta(n \cdot \log t)$
Se puede encontrar un conjunto que sume exactamente t.

Versión de opt: t es máximo

Forma Greedy: recibo elemento y veo qué pasa si lo meto o no a la bolsa.

Nota: $\theta(n \cdot t)$ prog. dinámica, pres t es gigantesco.
↳ 2^{32} ?!
NO es polinomial en el input $\theta(n \log t)$... es exponencial!

Veamos algoritmo exacto:

Exacto:

```
L ← <0>   (lista ordenada crecientemente siempre)
for i ← 1 to n   (lo incluyo o no?)
    L ← merge(L, L + xi)   (unión de todos los subconjuntos posibles)
        (a cada elto de L, sumas xi)
    truncar(L, t)   (eliminar subconjuntos que se pasan de t)
return max L   (máx t)
```

en el merge se eliminan repetidas

$x = 1, 3, 2$

$L \leftarrow (0)$ $\quad L+1 = <1>$

$L \leftarrow <0, 1>$ $\quad L+3 = <3, 4>$

$L \leftarrow <0, 1, 3, 4>$ $\quad L+2 = <2, 3, 5, 6>$

$L \leftarrow <0, 1, 2, 3, 4, 5, 6>$

$= <0, 1, 3, 4>$ agrego

$<2, 3, 5, 6>$ no agrego

Lo anterior era Fuerza Bruta...

$2^n + 2^{n-1} + 2^{n-2}$

¿Cuánto tiempo requiere? $\sim \Theta(2^n)$ pueden estar todos los subconjuntos posibles en L
INACEPTABLE

Vamos a aproximar. La idea es hacer que L no se agrande mucho

- Versión aproximada: Dado un elemento $z \in L$, tendremos una L' donde garantizamos que un $y \in L$ que representa a z está en L'. Y represente a z sii $\frac{z}{1+\delta} \leq y \leq z$

z es representado por alguien un poco MENOR (no puedo usar mayor porque me podría pasar de t)

```
Aprox (ε)
    L ← <∅>
    for i ← 1 to n
        L ← merge(L, L + x_i)
        truncar(L, t)
        filtrar(L, δ)   ← borro los elts representados por otros
    return max L

Filtrar (L, δ)
    L' ← <L[1]>   tomo el mín; por definición nadie lo puede representar.
    last ← L[1]
    for i ← 2 to |L|
        if L[i] > last · (1+δ)
            L' ← L' concat L[i]
            last ← L[i]
    return L'
```

Este error relacionado a $1+\delta$ se va acumulando.

Este error relacionado a $(1+\delta)$ se va acumulando.

Después de filtrar tenemos:

$L[1] \geq 1$

$L[i] > (1+\delta) \cdot L[i-1]$ } ellos consecutivos van creciendo consecutivamente

si $\ell = |L|$,

$L(\ell) > (1+\delta)^{\ell-1}$

pero solo trabajamos mientras $L(\ell) \leq t$ (al truncar)

$/\log(\cdot)$ $(1+\delta)^{\ell-1} \leq t$ ← $(\log(1+\delta)\cdot(\ell-1) \leq \log t)$

$\ell \leq \frac{\log t}{\log(1+\delta)} + 1$ ¡tenemos cota sup. para el largo!

Entonces en cada iteración del for pago $\mathcal{O}\left(1 + \frac{\log t}{\log(1+\delta)}\right)$

~~$\mathcal{O}(2^n)$~~ $\rightarrow \mathcal{O}\left(\frac{n \log t}{\log(1+\delta)}\right)$ y $n \log t$ es el tamaño del input.

Pareciera ser lineal en el tamaño del input, pero debemos estudiar bien qué sorpresas salen del $(1+\delta)$

Aunque sean números enteros, hay que dejar que $\delta > 0 \in \mathbb{R}$ si no explota.

¿Cuál es la relación entre ε y δ? ← peor caso

$\frac{z}{1+\delta} \leq y \leq z$

↓ y pasa a 2ª etapa. se reemplaza luego con x

$\frac{z}{(1+\delta)^2} \leq \frac{y}{1+\delta} \leq x \leq y$

↓ pasa a 3ª etapa y se reemplaza por w

$\frac{z}{(1+\delta)^3} \leq \frac{x}{1+\delta} \leq w \leq x$

∴ después de n iteraciones, entrego z^* que representa a z óptimo

$\frac{z}{(1+\delta)^n} \leq z^* \leq z$

x_1 solución o x_4 el máx de todos, $\leq t$.

Lo peor es que el primer elto sea $x_1 = t$, luego viene $x_2 = t/(1+\delta)$ y el alg. elimina x_1, luego llega $x_3 = \frac{t}{(1+\delta)^2}$, se elimina x_2 y se reemplaza por x_3, y así.

$\Rightarrow$ Luego digo $\underbrace{\frac{z}{(1+\delta)^n}}_{1+\varepsilon \text{ aproximado}} \leq z^* \leq z$

$$1+\varepsilon = (1+\delta)^n$$
$$\Rightarrow (1+\varepsilon)^{1/n} = 1+\delta$$
$$\Rightarrow \boxed{\delta = (1+\varepsilon)^{1/n} - 1}$$

Carmen demuestra que $\delta = \varepsilon/2n$ funciona

Si originalmente había mucha distancia entre los elementos, quizás con filtrar no elimino ninguno de la lista, pero este problema era fácil por sé! porque al sumar los costos se llega rápido a t. El problema es difícil cuando los costos entre eltos son muy parecidos.

Reemplazando δ, el costo del apróx es

$$O\left(\frac{n^2 \log t}{\log(1+\varepsilon)}\right) \quad \text{usaremos} \quad \frac{\varepsilon}{1+\varepsilon} \leq \ln(1+\varepsilon) \leq \varepsilon$$

$$\Rightarrow O\left(\frac{\overbrace{(1+\varepsilon)}^{\varepsilon \in [1,2]} n^2 \log t}{\varepsilon}\right) = \boxed{O\left(\frac{n^2 \log t}{\varepsilon}\right)}$$

está en función de $\frac{1}{\varepsilon}$, que era lo que queríamos

El problema es polinomial en n, en $\log t$ y en $\frac{1}{\varepsilon}$.

FIN

NOT