Algoritmos Codiciosos

Estructuras de Datos y Algoritmos - IIC2133

Los **algoritmos codiciosos** son aplicables a problemas de optimización con las siguientes características:

hay n inputs

queremos elegir un *subconjunto* de estos inputs que cumpla con dos condiciones:

satisfacer algunas restricciones —**solución factible**

maximizar o minimizar una *función objetivo*

Una solución factible que maximiza o minimiza la función objetivo es una **solución óptima**

Un algoritmo codicioso que efectivamente produce una solución óptima trabaja en etapas, considerando un input a la vez:

los inputs van siendo considerados en un cierto orden*

en cada etapa se decide si el input particular está o no en la solución óptima

si la inclusión del input en la solución óptima parcial viola las restricciones, entonces el input se descarta

Los algoritmos codiciosos *no siempre* producen soluciones óptimas

*Los inputs van siendo considerados en un cierto orden:

el orden está basado en alguna *medida de optimización*, p.ej., la propia función objetivo

seleccionamos un input de forma localmente óptima

… y "esperamos" que esta selección nos lleve a una solución óptima

El procedimiento abstracto **`greedy`**, en la próxima página, opera de la siguiente manera:

> **`select`** selecciona un input del arreglo ***a***, sacándolo de ***a***
>
> el input seleccionado es asignado a ***x***
>
> **`feasible`** determina si ***x*** puede ser incluido en **`solution`**
>
> **`union`** incluye ***x*** en **`solution`** y actualiza la función objetivo

```
greedy([] a, n):
    solution = ∅
    for k = 1,...,n:
        x = select(a)
        if feasible(solution, x):
            solution = union(solution, x)
    return solution
```

greedy describe la forma esencial de un algoritmo codicioso del llamado *modelo de subconjunto*:

dado un problema particular

hay que implementar apropiadamente los métodos **select**, **feasible** y **union**

Hay algoritmos codiciosos que no seleccionan un subconjunto de los inputs,

… sino que los incluyen a todos.

En estos casos, tomamos decisiones considerando los inputs en algún orden

… cada decisión se toma usando un criterio de optimización, que puede ser evaluado usando las decisiones ya tomadas —el *modelo de ordenación*

Ejemplos de algoritmos codiciosos ya estudiados

El algoritmo de Kruskal es un ejemplo de un algoritmo codicioso del *modelo de subconjunto*

```
kruskal():
    S = new Set()
    for each v in V:
        makeSet(v)
    ordenar las aristas de E de menor a
mayor costo w
    for each (u,v) in E, tomadas en
            orden no decreciente por costo w:
        if findSet(u) ≠ findSet(v):
            S = S ∪ {(u,v)}
            union(u,v)
    return S
```

```
greedy([] a, n):
  S = ∅
  for k = 1,...,n:
    x = select(a)
    if feasible(S, x):
      S = S ∪ {x}
  return S
```

El algoritmo de Dijkstra es un ejemplo de un algoritmo codicioso del *modelo de ordenación*

```
Dijkstra(Vertex s):
    Init(s)
    S = Ø
    q = new Queue(V)
    while !q.empty():
        u = q.xMin()
        S = S U {u}
        for each v in α[u]:
            reduce(u,v)
```

```
greedy([] a, n):
    S = ∅
    for k = 1,...,n:
        x = selectNext(S, a)
        S = S ∪ {x}
    return S
```

La mochila con objetos fraccionables

Tenemos n objetos y una mochila:

el objeto k pesa w_k

la capacidad de la mochila es m

si incluimos en la mochila una fracción x_k del objeto k ($0 \leq x_k \leq 1$), entonces obtenemos una ganancia $p_k x_k$

El problema es cómo llenar la mochila cumpliendo tres condiciones:

i) maximizar la ganancia total obtenida, $\sum_{1 \le k \le n} p_k x_k$

ii) sin exceder su capacidad m, es decir, $\sum_{1 \le k \le n} w_k x_k \le m$

iii) $0 \le x_k \le 1, \ 1 \le k \le n$

Una *solución factible* es cualquier conjunto $\{x_1, \ldots, x_n\}$ que cumple (ii) y (iii)

Una *solución óptima* es una solución factible que cumple (i)

P.ej.: $n = 3$ $m = 20$

$p_1, p_2, p_3 = 25, 24, 15$ $w_1, w_2, w_3 = 18, 15, 10$

Algunas soluciones factibles:

	x_1, x_2, x_3	$\sum_{1 \le k \le n} w_k x_k$	$\sum_{1 \le k \le n} p_k x_k$
a)	1/2, 1/3, 1/4	16.5	24.25
b)	1, 2/15, 0	20	28.2
c)	0, 2/3, 1	20	31
d)	0, 1, 1/2	20	31.5

Si la suma de todos los pesos es > m , entonces las soluciones óptimas llenan la mochila.

El problema consiste en seleccionar un subconjunto de objetos y determinar para cada uno la fracción x_i que debe ir dentro de la mochila

Posibles estrategias codiciosas:

Incluir a continuación el objeto con la mayor ganancia —solución b), no óptima

Incluir a continuación el objeto con el menor peso —solución c), no óptima

Incluir a continuación el objeto con la máxima *ganancia por unidad de capacidad usada* —solución d), óptima

greedyKnap supone que los objetos están ordenados de mayor a menor cociente p_k/w_k :

```
greedyKnap([] w, [] x, m, n):
    for k = 1,...,n:
        x[k] = 0.0
    U = m
    for k = 1,...,n:
        if w[k] > U:
            break
        x[k] = 1.0
        U = U - w[k]
    if k ≤ n:
        x[k] = U/w[k]
```

Actividades que usan un mismo recurso

Sea $S = \{1, 2, \ldots, n\}$ un conjunto de n actividades que deben usar un mismo recurso para poder ejecutarse; el recurso puede ser usado por sólo una actividad a la vez:

> cada actividad i tiene una hora de inicio s_i y una hora de término f_i , $s_i \leq f_i$, y, si se ejecuta, transcurre durante el intervalo de tiempo $[s_i , f_i)$
>
> las actividades i y j son **compatibles** si $[s_i, f_i)$ y $[s_j, f_j)$ no se traslapan, es decir, si $s_i \geq f_j$ o $s_j \geq f_i$

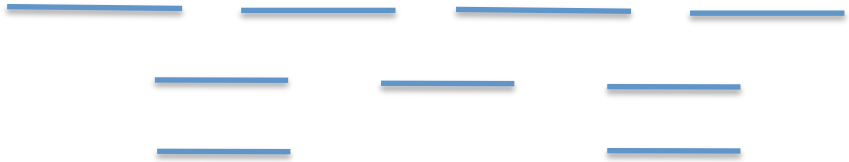
El problema consiste en seleccionar un *subconjunto de tamaño máximo de actividades mutuamente compatibles*

Estrategias codiciosas que no necesariamente producen soluciones óptimas:

- la próxima actividad seleccionada es aquélla que tiene la hora de inicio más temprana

- la próxima actividad seleccionada es aquélla que dura menos

- la próxima actividad seleccionada es aquélla con el menor número de traslapes



Si las actividades están ordenadas de menor a mayor hora de término — $f_1 \leq f_2 \leq \ldots \leq f_n$ — usamos el siguiente algoritmo codicioso:

la próxima actividad seleccionada es aquélla *con la hora de término más temprana* que puede programarse compatiblemente (ya que así se maximiza el tiempo disponible restante)

```
seleccion([] s, [] f):
  A = {1}
  j = 1
  for i = 2,...,n:
     if s[i] ≥ f[j]:
       A = A ∪ {i}
       j = i
  return A
```

El conjunto A colecciona las actividades seleccionadas

j especifica la actividad agregada a A más recientemente

Como las actividades son consideradas en orden de menor a mayor hora de término, f_j es siempre la máxima hora de término de las actividades en A

Toma tiempo O(n) para programar n actividades, si estas vienen ordenadas según sus horas de término

Una vez hecha una elección codiciosa, tenemos un nuevo problema de optimización del mismo tipo que el problema original

Programar tareas con plazos y ganancias

Tenemos que hacer n tareas.

Cada tarea i, $1 \leq i \leq n$, tiene dos propiedades:

un plazo d_i (entero)

una ganancia p_i que se obtiene si y sólo si la tarea es hecha a tiempo (dentro del plazo)

Para hacer una tarea, hay que procesarla en una máquina durante una unidad de tiempo; sólo hay una máquina disponible.

Una *solución factible* es un subconjunto T de tareas tal que cada tarea en T puede hacerse a tiempo:

> el **valor** de T es la suma de las ganancias de las tareas incluidas en T, $\sum_{k \in T} p_k$

Una solución factible es *óptima* si su valor es máximo

P.ej., para $n = 4$ sean

$p_1, p_2, p_3, p_4 =$ 100, 10, 15, 27

$d_1, d_2, d_3, d_4 =$ 2, 1, 2, 1

solución factible	**orden de procesamiento**	**valor**
{ 1, 2 }	2, 1	110
{ 1, 3 }	1, 3 o 3, 1	115
{ 1, 4 }	4, 1	127
{ 2, 3 }	2, 3	25
{ 3, 4 }	4, 3	42
{ 1 }	1	100
{ 2 }	2	10
{ 3 }	3	15
{ 4 }	4	27

Tomemos la propia función objetivo como medida de optimización para elegir la próxima tarea:

> la próxima tarea a considerar es la que aumenta más la suma $\sum_{k \in T} p_k$, sujeta a que el T resultante sea factible
>
> consideramos las tareas en orden decreciente de los p_i's

Al igual que en los ejemplos anteriores, hay que demostrar que esta estrategia produce una solución óptima

La estrategia funciona en el ejemplo:

inicialmente, $T = \varnothing$ y $\sum_{i \in T} p_i = 0$ y consideramos las tareas en el orden 1, 4, 3 y 2

agregamos la tarea 1 a T —tiene la mayor ganancia y $T = \{1\}$ es factible

agregamos la tarea 4 a T —$T = \{1, 4\}$ es factible

consideramos la tarea 3, pero la descartamos —$T = \{1, 3, 4\}$ no es factible

finalmente, consideramos la tarea 2 y la descartamos —$T = \{1, 2, 4\}$ no es factible

Códigos de Huffman

El *algoritmo codicioso de Huffman*

… es una técnica para comprimir datos

… usa la frecuencia de ocurrencia de cada carácter para construir una forma óptima de representar cada carácter como un string binario

El problema es diseñar un *código binario* en que cada carácter sea representado por un string binario único (palabra)

Supongamos que tenemos un archivo con 100,000 caracteres, en que seis caracteres ocurren con las siguientes frecuencias (en miles): a: 45, b: 13, c: 12, d: 16, e: 9, f: 5

Un *código de largo fijo* necesita 3 bits/carácter para representar seis caracteres: a = 000, b = 001, …, f = 101; y en total 300,000 bits para codificar el archivo completo

Un *código de largo variable* —los caracteres más frecuentes tienen palabras más cortas, los menos frecuentes, más largas— puede ser mucho mejor; p.ej., el código

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

requiere sólo 224,000 bits para representar el archivo.

Códigos prefijos. Ninguna palabra es también un prefijo de alguna otra palabra (la compresión óptima siempre puede lograrse con un código prefijo):

– los códigos prefijos facilitan la codificación y decodificación

La decodificación necesita una representación para el código de modo que la primera palabra pueda identificarse fácilmente:

– un árbol binario cuyas hojas son los caracteres dados

– la representación para un carácter es la ruta raíz-carácter: 0 = "vaya al hijo izquierdo" y 1 = "vaya al hijo derecho"

Un *código prefijo óptimo* (OPC) se representa por un árbol binario lleno —cada nodo que no es una hoja tiene dos hijos:

- El código de largo fijo anterior no es óptimo —en su árbol hay palabras que comienzan 10…, pero no 11…, lo que significa que hay palabras más largas innecesarias
- Si C es el alfabeto, el árbol para un OPC tiene
  - $|C|$ hojas, una para cada letra del alfabeto
  - $|C| - 1$ nodos internos

Dado un árbol T correspondiente a un código, el número de bits necesarios para codificar un archivo —el *costo* de T— es

$$B(T) = \sum f(c)\ d_T(c)$$

$f(c)$ es la frecuencia del carácter c en el archivo

$d_T(c)$ es la profundidad de la hoja correspondiente a c en T

Construcción de un código. El algoritmo de David Huffman produce un OPC —el *código de Huffman*— construyendo el árbol de manera *bottom-up*:

– comienza con un conjunto de $|C|$ hojas

– realiza una secuencia de $|C| - 1$ "mezclas" para crear el árbol final

En el siguiente algoritmo

- C es un conjunto de n caracteres
- cada carácter $c \in C$ tiene una frecuencia $f[c]$
- usamos una cola de prioridades para identificar a los dos objetos de menor frecuencia que mezclamos para crear un nuevo objeto cuya frecuencia es la suma de esas frecuencias

```
Node huffman(Set C)
    n = |C|
    Q = C
    for i = 1 ... n:
        z = Node()
        x = left[z] = xMin(Q)
        y = right[z] = xMin(Q)
        f[z] = f[x] + f[y]
        insert(Q, z)
    return xMin(Q)
```

Implementamos `Q` como un *heap* binario:

- la inicialización de `Q` puede ejecutarse en tiempo O(n)
- el ciclo **`for`** se ejecuta $n - 1$ veces
- cada **`xMin`** e **`insert`** sobre el *heap* requiere tiempo O(log n)
- el ciclo **`for`** contribuye tiempo O(n log n)
- **`huffman`** toma tiempo O(n log n) para n caracteres

Es decir, el proceso de construir un árbol óptimo mediante mezclas puede empezar con la *elección codiciosa de mezclar los dos caracteres con las frecuencias más bajas*:

- podemos ver el costo de una mezcla como la suma de las frecuencias de los dos ítemes mezclados
- puede probarse que el costo total del árbol construido es la suma de los costos de sus mezclas
- de modo que de todas las posibles mezclas en cada paso, **`huffman`** elige la que introduce el menor costo