Teoría de Algoritmos – TP3

“Planificación de Órdenes de Trabajo”

Materia: 75.29 Teoría de Algoritmos

Alumno: Juan Sebastián Goldberg

Padrón: 82078

Mail: sebas.goldber@gmail.com

Fecha: 3-DIC-2014

Contenido

Contenido 2

Punto A 3

Planteo del problema 3

Solución propuesta 3

Cálculo del Óptimo 4

Justificación de órdenes del lenguaje y librerías 5

Punto B 5

Análisis 5

Orden del Algoritmo Verificador 6

Reducción 6

Código Fuente 6

tp3.py 7

lista\_ordenada.py 10

Punto A

Planteo del problema

Sea **A = {a1..an}** El conjunto de trabajos a realizar en una máquina.

Cada ai tiene definido:

* **ti**: Tiempo de máquina necesario para completar el trabajo (entero entre 1 y **W**).
* **vi**: Vencimiento (entero).
* **bi**: Beneficio (solo si se completa antes del vencimiento).

Objetivo: Obtener el máximo beneficio.

Solución propuesta

Ordenamos las tareas por vencimiento, luego la primer tarea es la 1, la siguiente la 2, y la última la **n**.

Si queremos obtener el beneficio óptimo (OPT) hasta el instante **t**, utilizando las tareas hasta la **i**, podemos plantear lo siguiente:

**OPT(t,i) = max{**

**OPT(t\_inicial(t,i), tarea\_anterior(i)) + b(t,i),**

**OPT(t, tarea\_anterior(i))**

**}**

Básicamente lo que estamos planteando es lo siguiente: El óptimo hasta el tiempo **t**, utizando las tareas hasta la **i** estará dado por las siguiente desición:

* Si incluimos la tarea **i**, entonces el beneficio óptimo estará constituido por el beneficio óptimo hasta el tiempo inicial de la tarea **i** (con finalización en t) utilizando las tareas hasta la tarea anterior a la **i**. A esto le sumaremos el beneficio que genera la tarea i si finaliza en el tiempo t.
* Si no incluimos la tarea i, entonces el óptimo estará dado por el beneficio óptimo hasta el tiempo t utilizando las tareas hasta la tarea anterior a la i.

Existen dos excepciones a lo planteados:

* Si el tiempo inicial de una tarea i que finaliza en el tiempo t es negativo, entonces hasta ese tiempo **t**, no se incluirá la tarea **i** en la solución óptima.
* Si no existe tarea anterior a la tarea **i** (**i**=1), entonces se incluirá el máximo entre el óptimo si incluimos la tarea y el beneficio de la tarea en sí en el tiempo de finalización **t**.
* Si ocurren ambas excepciones mencionadas, entonces el óptimo en el tiempo t mencionado será nulo.

Cálculo del Óptimo

Generamos una matriz **M** donde las filas sean el tiempo **t** (de 0 a **W**) y las columnas sean las tareas ordenadas por vencimiento (de 1 a **n**).

Inicializamos la fila t=0 toda en 0.

Luego el óptimo en cada componente de la matriz **M** estará dado por el siguiente algoritmo:

|  |
| --- |
| for t in xrange(1,self.ultimo\_vencimiento()+1): |
| for i in self.indices\_tareas(): |
| try: |
| try: |
| M[t][i] = max(  M[self.t\_inicial(t,i)][self.tarea\_anterior(i)]+  self.b(i,t), |
| M[t][self.tarea\_anterior(i)]) |
| except TiempoInicialNegativo: |
| M[t][i] = M[t][self.tarea\_anterior(i)] |
| except NoExisteTareaAnterior: |
| try: |
| M[t][i] = max(M[self.t\_inicial(t,i)][i], self.b(i,t)) |
| except TiempoInicialNegativo: |
| M[t][i] = 0 |

**Obtención de la Solución a Partir de la Matriz**

Obtener la solución a partir de la matriz es relativamente sencillo:

* Si hasta el tiempo t hasta la tarea i el beneficio es mayor que el beneficio hasta el tiempo t hasta la tarea anterior, entonces la tarea i está incluida en la planificación. Luego analizaremos lo que ocurre hasta el tiempo inicial de la tarea i con finalización en el tiempo t.
* Sino, debemos decidir si analizamos el tiempo t -1 o la tarea anterior, por lo tanto:
  + Si hasta el tiempo t-1 hasta la tarea i el beneficio es mayor que hasta el tiempo t hasta la tarea anterior, entonces analizaremos lo que ocurre hasta el tiempo t-1 de la tarea i.
  + Sino analizaremos lo que ocurre hasta el tiempo t de la tarea anterior a la i.
* Continuaremos este análisis hasta que se cumpla alguna de las siguientes condiciones:
  + No exista tarea anterior a la i, con lo cual i = 1, e incluiremos esta tarea en la solución solo si el beneficio hasta el tiempo t en análisis es mayor a cero.
  + t sea menor que cero.

El algoritmo es el siguiente:

|  |
| --- |
| t = self.ultimo\_vencimiento() |
| i = self.ultima\_tarea() |
| planificacion = [] |
| planificada = [False for i in self.indices\_tareas()] |
|  |
| while t>=0: |
| try: |
| if M[t][i] > M[t][self.tarea\_anterior(i)]: |
| planificacion.insert(0,i) |
| planificada[i] = True |
| t=self.t\_inicial(t,i) |
| i=self.tarea\_anterior(i) |
| elif M[t-1][i] > M[t][self.tarea\_anterior(i)]: |
| t = t - 1 |
| else: |
| i = self.tarea\_anterior(i) |
| except NoExisteTareaAnterior: |
| if M[t][i]>0: |
| planificacion.insert(0,i) |
| planificada[i] = True |
| break |

Justificación de órdenes del lenguaje y librerías

El lenguaje utilizado para la implementación del trabajo práctico es python.

Los órdenes referentes a operaciones realizadas con las estructuras provistas por el lenguaje y librerías se obtuvieron a partir de la siguiente documentación (en todos los casos se trabajo con los casos pesimistas):

* Órdenes para estructuras de datos nativas:
  + <https://wiki.python.org/moin/TimeComplexity>
* Documentación de módulo estándar de colas de prioridad:
  + <https://docs.python.org/2/library/heapq.html>
* Documentación de módulo estándar para realizar búsquedas binarias e inserción ordenada:
  + <https://docs.python.org/2/library/bisect.html>

Punto B

Análisis

El problema en que los tiempos y los vencimientos pasan a ser reales, en realidad se puede analizar de la siguiente forma. Supongamos que tenemos tiempo y vencimientos los cuales son números demasiado grandes. El algoritmo que resuelve el enunciado, está claro que es O(nW), Vemos que podemos escribir W >= 2\*\*piso(log(W)). Si k = piso(log(W)), entonces W >= 2\*\*k.

Con lo cual O(n\*W) = O(n\*2\*\*k). Cuando k es grande, por ejemplo 100, el problema pasa a ser computacionalmente intratable.

Dados tiempos y vencimientos reales arbitrarios finitos, podríamos convertirlos en enteros, pero si el último vencimiento es demasiado grande, el problema es computacionalmente intratable.

Extendiendo lo planteados a si los tiempos y los vencimientos son reales arbitrarios, W podría tomar números demasiado grandes o infinitos, con lo cual el problema es computacionalmente intratable.

Orden del Algoritmo Verificador

Dada una planificación P = [p1, p2,…, pn] y un número K, podemos plantear el siguiente algoritmo para verificar si se cumple que P produce un beneficio de al menos K:

**t = 0**

**B = 0**

**por cada p en P:**

**t = t + d(p) # Duración de la tarea p**

**B = B + b(p, t) # Beneficio de la tarea p si finaliza en t**

**si B >= K: return True**

**return False**

Vemos que el orden del algoritmo planteado es O(n), con lo cual el verificador corre en tiempo polinomial. Luego el problema es NP.

Reducción

Si elegimos el problema NP-completo Subset Sum (SS), lo que debemos demostrar es que el mismo puede resolverse utilizando como caja negra al problema TP3, es decir, estaremos reduciendo SS a TP3: SS <p TP3

En el problema Subset Sum tenemos un conjunto U = {w1, .., wn} de pesos, debemos encontrar un subconjunto S incluido en U que no supere un numero W dado. Queremos saber si planteado el problema y dado un K arbitrario, se verifica |S| >= K

Para reducir Subset Sum a TP3 plantearemos lo siguiente:

* ti = wi para cada tarea i (la duración)
* vi = W para cada tarea i (el vencimiento)
* bi = 1 para cada tarea i (el beneficio)

Luego dada la solución S’ de TP3, si |S’| >= K entonces se verifica que |S| >= K en Subset Sum.

Demostramos así que la resolución de TP3 es al menos tan difícil como la resolución de Subset Sum. Luego TP3 es NP-completo.

Código Fuente

Para acceder a la totalidad del código fuente puede clonar el siguiente repositorio:

<https://github.com/facultad/7529-tp3>

A continuación se incorpora el código fuente del trabajo práctico, y para ganar claridad y disminuir la cantidad de código se excluyeron los casos de prueba de cada módulo.

La mayoría de los métodos fueron documentados de forma de registrar el orden de los algoritmos implementados.

tp3.py

|  |
| --- |
| #!/usr/bin/python |
| # coding=utf-8 |
|  |
| import sys |
| from lista\_ordenada import ListaOrdenada |
|  |
|  |
| class NoExisteTareaAnterior(Exception): |
| pass |
|  |
|  |
| class TiempoInicialNegativo(Exception): |
| pass |
|  |
|  |
| class Tarea: |
|  |
| def \_\_init\_\_(self, id, tiempo, beneficio, vencimiento): |
| self.id = id |
| self.tiempo = tiempo |
| self.beneficio = beneficio |
| self.vencimiento = vencimiento |
|  |
| def \_\_cmp\_\_(self, other): |
| if self.vencimiento < other.vencimiento: |
| return -1 |
| if self.vencimiento > other.vencimiento: |
| return 1 |
| return 0 |
|  |
| def \_\_str\_\_(self): |
| return '%s(d=%s, b=%s, v=%s)' % (self.id, self.tiempo, self.beneficio, |
| self.vencimiento) |
|  |
| class TP3: |
|  |
| def \_\_init\_\_(self, lines): |
| """ |
| O(n\*log(n)) |
| n: len(lines) |
| """ |
| id = 0 |
| self.tareas = ListaOrdenada(permitir\_repetidos=True) # O(1) |
| for line in lines: # len(lines) |
| line = line.strip() # O(1) |
| if line == "": |
| continue |
| tiempo, beneficio, vencimiento = line.split(',') # O(1) |
| id += 1 |
| tiempo = int(tiempo) |
| beneficio = float(beneficio) |
| vencimiento = int(vencimiento) |
| tarea = Tarea(id, tiempo, beneficio, vencimiento) # O(1) |
| self.tareas.insert(tarea) # O(log(len(self.tareas))) |
|  |
| def indices\_tareas(self): |
| return range(len(self.tareas)) |
|  |
| def d(self, i): |
| """ |
| Duración de la tarea i |
| """ |
| return self.tareas[i].tiempo |
|  |
| def b(self, i, t): |
| """ |
| Beneficio de la tarea i si finaliza en el instante t. |
| """ |
| if t > self.tareas[i].vencimiento: |
| return 0 |
| return self.tareas[i].beneficio |
|  |
| def t\_inicial(self, t\_final, i): |
| """ |
| Instante inicial de la tarea i si finaliza en el instante t\_final |
| """ |
| t\_inicial = t\_final - self.d(i) |
| if t\_inicial < 0: |
| raise TiempoInicialNegativo() |
| return t\_inicial |
|  |
| def tarea\_anterior(self, i): |
| """ |
| Obtiene la tarea anterior a la tarea i. |
| """ |
| if i == 0: |
| raise NoExisteTareaAnterior() |
| return i-1 |
|  |
| def ultima\_tarea(self): |
| return len(self.tareas)-1 |
|  |
| def ultimo\_vencimiento(self): |
| return self.tareas[self.ultima\_tarea()].vencimiento |
|  |
| def resolver(self): |
| """ |
| O(W\*n) |
| W: Último vencimiento |
| n: Cantidad de tareas |
| Obtiene la planificación óptima y su beneficio. |
| Devuelve la tupla (planificacion, resto, beneficio) |
| """ |
|  |
| M = [] |
| # O(W\*n) |
| for t in xrange(0,self.ultimo\_vencimiento()+1): # W+1 |
| M.append([]) |
| for i in self.indices\_tareas(): # n |
| M[t].append(0) |
|  |
| for t in xrange(1,self.ultimo\_vencimiento()+1): # W+1 |
| for i in self.indices\_tareas(): # n |
| try: |
| try: |
| M[t][i] = max(M[self.t\_inicial(t,i)][self.tarea\_anterior(i)]+self.b(i,t), |
| M[t][self.tarea\_anterior(i)]) # O(1) |
| except TiempoInicialNegativo: |
| M[t][i] = M[t][self.tarea\_anterior(i)] # O(1) |
| except NoExisteTareaAnterior: |
| try: |
| M[t][i] = max(M[self.t\_inicial(t,i)][i], self.b(i,t)) #(1) |
| except TiempoInicialNegativo: |
| pass |
|  |
| t = self.ultimo\_vencimiento() |
| i = self.ultima\_tarea() |
| planificacion = [] |
| planificada = [False for i in self.indices\_tareas()] |
|  |
| # O(W+n) |
| while t>=0: |
| try: |
| # Si en el tiempo t para la tarea i el beneficio es mayor que |
| # el beneficio en el tiempo t de la tarea anterior, entonces la tarea i |
| # está incluida en la planificación. |
| if M[t][i] > M[t][self.tarea\_anterior(i)]: |
| planificacion.insert(0,i) |
| planificada[i] = True |
| t=self.t\_inicial(t,i) |
| i=self.tarea\_anterior(i) |
| # Sino, si en el tiempo t-1 la tarea i tiene mayor beneficio que |
| # en el tiempo t de la tarea anterior, entonces decrementamos t |
| # en una una unidad |
| elif M[t-1][i] > M[t][self.tarea\_anterior(i)]: |
| t = t - 1 |
| # Sino, significa que la tarea i no está incluida en la planificación |
| # y vamos a la tarea anterior. |
| else: |
| i = self.tarea\_anterior(i) |
| except NoExisteTareaAnterior: |
| # En caso que no exista tarea anterior, significa que estamos en la primer |
| # tarea, por tanto, en caso que el beneficio sea mayor a 0, entonces indica |
| # que la primer tarea está incluida en la planificación. |
| if M[t][i]>0: |
| planificacion.insert(0,i) |
| planificada[i] = True |
| break |
|  |
| resto = set() |
| # O(n) |
| for i in self.indices\_tareas(): |
| if not planificada[i]: |
| resto.add(self.tareas[i].id) |
|  |
| # O(n) |
| return ([self.tareas[i].id for i in planificacion], resto, |
| M[self.ultimo\_vencimiento()][self.ultima\_tarea()]) |
|  |
|  |
|  |
|  |
| def procesar(): |
| for filepath in sys.argv[1:]: |
| with open(filepath) as f: |
| tp3 = TP3(f.readlines()) |
| tp3.resolver() |
|  |
|  |
| if \_\_name\_\_ == '\_\_main\_\_': |
| if len(sys.argv) == 1: |
| unittest.main() |
| else: |
| procesar() |

lista\_ordenada.py

|  |
| --- |
| #!/usr/bin/python |
| # coding=utf-8 |
|  |
| import bisect |
|  |
| class ElementoNoEncontrado(Exception): |
|  |
| def \_\_init\_\_(self, elemento): |
| Exception.\_\_init\_\_(self, |
| "El elemento %s no se ha encontrado en la lista" % elemento) |
|  |
| class ListaOrdenada(): |
|  |
| def \_\_init\_\_(self, permitir\_repetidos=False): |
| self.lista = [] |
| self.permitir\_repetidos = permitir\_repetidos |
|  |
| def \_\_len\_\_(self): |
| return len(self.lista) |
|  |
| def \_\_getitem\_\_(self, i): |
| return self.lista[i] |
|  |
| def \_\_str\_\_(self): |
| return self.lista.\_\_str\_\_() |
|  |
|  |
| def iteritems(self): |
| """ |
| O(1) |
| """ |
| return iter(self.lista) |
|  |
| def insert(self, node): |
| """ |
| O(n\*log(n)) |
| """ |
| i = bisect.bisect\_left(self.lista, node) |
| if not self.permitir\_repetidos: |
| if i <> len(self.lista) and self.lista[i] == node: |
| raise Exception('El nodo %s ya existe en la lista.' % node) |
| return self.lista.insert(i, node) |
|  |
| def has(self, node): |
| """ |
| O(log(n)) |
| """ |
| i = bisect.bisect\_left(self.lista, node) |
| if i <> len(self.lista) and self.lista[i] == node: |
| return True |
| return False |
|  |
| def get\_item(self, item): |
| """ |
| O(log(n)) |
| """ |
| i = bisect.bisect\_left(self.lista, item) |
| if i <> len(self.lista) and self.lista[i] == item: |
| return self.lista[i] |
| raise ElementoNoEncontrado(item) |