Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

Algoritmia y Complejidad

Laboratorio – Sesión 2

Algoritmos voraces

Laboratorio Jueves 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2018/2019

Eduardo Graván Serrano – 03212337L

Marcos Barranquero Fernández – 51129104N

Ejercicio 2: Ficheros

Enunciado:

Se tiene que almacenar un conjunto de n ficheros en una cinta magnética (soporte de almacenamiento de recorrido secuencial), teniendo cada fichero una longitud conocida l1, l2, …, ln. Para simplificar el problema, puede suponerse que la velocidad de lectura es constante, así como la densidad de información en la cinta. Se conoce de antemano la tasa de utilización de cada fichero almacenado, es decir, se sabe la cantidad de peticiones pi correspondiente al fichero i que se van a realizar. Tras la petición de un fichero, al ser encontrado la cinta es automáticamente rebobinada hasta el comienzo de la misma. El objetivo es decidir el orden en que los n ficheros deben ser almacenados para que se minimice el tiempo medio de carga, creando un algoritmo voraz correcto.

Código:

```
from random import randint


class fichero(object):
    """clase fichero que almacena la informacion sobre los ficheros
que van a estar en la cinta"""
    longitud = 0
    llamadas = 0

    def __init__(self, x, y):
        self.longitud = x
        self.llamadas = y

    def __str__(self):
        cadena = "Long: ", self.longitud, "; Llamadas: ",
self.llamadas, "L/ll: ", (
            self.longitud/self.llamadas)
        return str(cadena)

    __repr__ = __str__


def inicializar_cinta(n):
    """ Este metodo se encarga de inicializar el vector con n
ficheros y valores aleatorios"""
    ficheros = []
    for _ in range(n):
        ficheros.append(fichero(randint(1, 200), randint(1, 50)))
    return ficheros


def ordenar_cinta(ficheros):
    """Este metodo se encarga de ordenar la cinta magnetica
    Para ello, dividir el valor de la longitud del fichero por el
numero de llamadas que se le va a hacer
```

```
    Despues de hacer esta division, se ordenan usando el algoritmo
de la burbuja
    De esta forma se consigue que los ficheros se almacenen en el
orden mas eficiente"""
    long = len(ficheros)
    # Algoritmo de la burbuja:
    for i in range(long):
        for j in range(0, long-i-1):
            if ((ficheros[j].longitud / ficheros[j].llamadas)) >
((ficheros[j+1].longitud / ficheros[j+1].llamadas)):
                ficheros[j], ficheros[j+1] = ficheros[j+1],
ficheros[j]
    print(ficheros)
    return ficheros

# Metodo de prueba


def test():
    ficheros = ordenar_cinta(inicializar_cinta(20))
    for obj in ficheros:
        print("Long: ", obj.longitud, "Llamadas: ",
              obj.llamadas, "LL: ", obj.longitud/obj.llamadas)


test()
```

Explicación:

Primero importamos un módulo que nos permite generar enteros aleatorios, para poder generar datos aleatorios para cada fichero.

Creamos una clase fichero que contiene como atributos la longitud y el nº de llamadas.

Creamos una lista de ficheros ordenados aleatoriamente.

Para ordenar de la forma más eficiente, dividimos la longitud entre el nº de llamadas y ordenamos los ficheros en función de eso, de menor a mayor.

De esta forma, los ficheros que sean llamados muchas veces y tengan menor longitud estarán los primeros, serán más accesibles y por tanto consumirán menor tiempo. Y al revés: los ficheros más grandes que sean llamados menos veces se encuentran al final de la lista.

Ejercicio 5: Dijkstra

Enunciado:

Se tiene un grafo dirigido G = < N, A >, siendo N = {1, …, n} el conjunto de nodos y A ⊆ NxN el conjunto de aristas. Cada arista (i, j) ∈ A tiene un coste asociado cij (cij > 0 ∀i, j ∈ N; si (i, j) ∉ A puede considerarse cij = +∞). Sea M la matriz de costes del grafo G, es decir, M [i, j] = cij. Teniendo como datos la cantidad de nodos n y la matriz de costes M, se pide encontrar tanto el

camino mínimo entre los nodos 1 y n como la longitud de dicho camino usando el algoritmo de Dijkstra, utilizando las siguientes ideas: • Crear una estructura de datos que almacene las distancias temporales conocidas (inicializadas al coste de la arista del vértice 1 a cada vértice j, o +∞ si no existe dicha arista) para los vértices no recorridos (inicialmente, todos salvo el 1). • Seleccionar como candidato el que tenga menor distancia temporal conocida, eliminarle del conjunto de vértices no recorridos, y actualizar el resto de distancias temporales si pueden ser mejoradas utilizando el vértice actual. • Se necesitará almacenar de alguna manera la forma de recorrer el grafo desde el vértice 1 al vértice n (no necesariamente igual al conjunto de decisiones tomadas).

Código:

```
import math


def dijkstra(nodos, matriz_costes):
    # Lista de nodos en el orden visitado hasta llegar a la
solución.
    camino = []
    candidatos = list(nodos)  # Lista de nodos aún no tomados en el
camino.

    primero = nodos[0]

    camino.append(primero)  # Añado primer nodo
    candidatos.remove(primero)  # Quito el primero puesto que ya lo
he tomado.
    # Elimino el primer nodo de la lista de nodos restantes.
    nodos.remove(primero)

    # Añado en el array distancias las distancias del primer nodo
al resto.
    distancias = []
    for i in range(1, len(matriz_costes[0])):
        distancias.append(matriz_costes[0][i])

    # Mientras aún haya candidatos:
    while candidatos:

        # Buscaré el nodo con el camino mínimo en el array de
distancias:
        coste_camino_minimo = math.inf
        nodo_camino_minimo = None
        for coste_camino in distancias:
            nodo = distancias.index(coste_camino) + 1

            # Si encuentro un camino más barato a un nodo sin
visitar que el
            # anterior, lo tomo:
```

```
            if coste_camino < coste_camino_minimo and nodo in
candidatos:
                nodo_camino_minimo = nodo
                coste_camino_minimo = coste_camino

        # Una vez tomado el nodo cuyo camino es más barato, lo
elimino de candidatos.
        candidatos.remove(nodo_camino_minimo)

        # ... y lo añado al camino seguido:
        camino.append(nodo_camino_minimo)

        # Y actualizo la matriz de costes:
        coste_distancias_nodo_minimo = []
        for i in range(1, len(matriz_costes[nodo_camino_minimo])):
            coste_distancias_nodo_minimo.append(
                matriz_costes[nodo_camino_minimo][i])

        # Después, actualizo distancias en caso de hayar un camino
más barato a un nodo:
        for i in range(len(distancias)):
            distancias[i] = min(
                distancias[i], coste_camino_minimo +
coste_distancias_nodo_minimo[i])

    # Finalmente, calculo el coste:
    coste_camino = 0
    for i in range(len(camino)-1):
        # Sumo el coste de ir del nodo camino[i] al nodo
camino[i+1]
        coste_camino += matriz_costes[camino[i]][camino[i+1]]
    return coste_camino, camino


if __name__ == '__main__':

    #     20
    # |0|--->|1|
    #  |     ^|
    # 3|  1/  |2
    #  v /    v
    # |2|--->|3|
    #     4

    nodos = [0, 1, 2, 3]
    matriz_costes = [
        [0, 20, 3, math.inf],
        [math.inf, 0, math.inf, 2],
        [math.inf, 1, 0, 4],
```

```
        [math.inf, math.inf, math.inf, 0]
    ]

    coste, camino_minimo = dijkstra(nodos, matriz_costes)
    print("Coste: ", coste, "Camino: ", camino_minimo)
```

Explicación:

El método recibe una lista de los nodos del grafo y una matriz de distancias de nodo a nodo.

Inicialmente se instancian dos listas:

- Camino: lista de nodos en el orden en el que se han ido tomando.
- Candidatos: lista de nodos que quedan por tomar.

Se añade el primer nodo a camino (el nodo de inicio) y se elimina de candidatos.

Se crea la lista de distancias del primer nodo al resto (si no puede llegar a alguno, la distancia es infinito).

Después, se entra en un bucle que busca el camino mínimo por el cual se llega a un nodo que no se encuentre en camino aún. Una vez encontrado, se elimina de candidatos y se añade a camino. Después, se actualiza la matriz de costes. Finalmente pero dentro del bucle, se actualiza la lista de distancias mínimas para ajustarla al último nodo tomado.

Al final del método, se calcula el coste total de llegar a el nodo final.

Ejercicio 6: Shrek

Enunciado:

Shrek, Asno y Dragona llegan a los pies del altísimo castillo de Lord Farquaad para liberar a Fiona de su encierro. Como sospechaban que el puente levadizo estaría vigilado por numerosos soldados se han traído muchas escaleras, de distintas alturas, con la esperanza de que alguna de ellas les permita superar la muralla; pero ninguna escalera les sirve porque la muralla es muy alta. Shrek se da cuenta de que, si pudiese combinar todas las escaleras en una sola, conseguiría llegar exactamente a la parte de arriba y poder entrar al castillo. Afortunadamente las escaleras son de hierro, así que con la ayuda de Dragona van a "soldarlas". Dragona puede soldar dos escaleras cualesquiera con su aliento de fuego, pero tarda en calentar los extremos tantos minutos como metros suman las escaleras a soldar. Por ejemplo, en soldar dos escaleras de 6 y 8 metros tardaría 6 + 8 = 14 minutos. Si a esta escalera se le soldase después una de 7 metros, el nuevo tiempo sería 14 + 7 = 21 minutos, por lo que habrían tardado en hacer la escalera completa un total de 14 + 21 = 35 minutos. Diseñar un algoritmo eficiente que encuentre el mejor coste y manera de soldar las escaleras para que Shrek tarde lo menos posible es escalar la muralla, indicando las estructuras de datos elegidas y su forma de uso. Se puede suponer que se dispone exactamente de las escaleras necesarias para subir a la muralla (ni sobran ni faltan), es decir, que el dato del problema es la colección de medidas de las "miniescaleras" (en la estructura de datos que se elija), y que solo se busca la forma óptima de fundir las escaleras.

Código:

```
def shrek(escaleras):
    """ Para que la suma acumulada sea la menor, deben de soldarse
    escaleras de menor a mayor coste.
    """

    suma = 0
    for escalera in escaleras:
        suma = suma*2 + escalera
    print(suma)

    escaleras = escaleras.sort()
    # Una vez ordenadas, se acumula el coste:
    suma = 0

    print(escaleras)
    for escalera in escaleras:
        suma = suma*2 + escalera
    return suma


escaleras = [1, 6, 4, 1, 5, 8]
tiempo = shrek(escaleras)
print(tiempo)
```

Explicación:

El código es bastante simple. Para minimizar el coste de soldar, se deben soldar de menor a mayor. Así, la suma acumulada se minimiza, puesto que se volverán a sumar más veces las escaleras pequeñas que las grandes.

Por tanto, ordenamos de menor a mayor el vector de escaleras. Después, la variable suma almacena el acumulado de la suma de escaleras y lo devuelve.