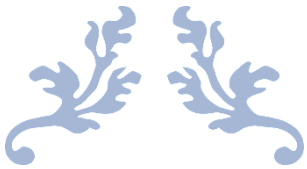

PRÁCTICA 1: ALGORITMOS VORACES: SELECCIÓN DE ACTIVIDADES

Algoritmos Avanzados. Grado en Ingeniería Informática.

19 DE OCTUBRE DE 2017
JORGE ARANDA GARCÍA
JOSE VICENTE BAÑULS GARCÍA

ÍNDICE

Tabla de contenido

INTRODUCCIÓN

El objetivo de esta práctica es permitirnos a los alumnos profundizar en el conocimiento de los algoritmos voraces y su funcionamiento. Para ello se ha propuesto el problema de la selección de actividades, en el que se busca una serie de actividades compatibles entre sí, es decir, cuyos tiempos de inicio y fin no se pisen, de tal forma que se escoja un número máximo. Para ello se han propuesto dos opciones. La primera: o bien el array de tiempos de comienzo se encuentra ordenado de mayor a menor, o bien el array de tiempos de fin se encuentra ordenado de menor a mayor. La segunda: ninguno de estos array se encuentra ordenado. Las soluciones propuestas para cada apartado son bien distintas, como se podrá ver en apartados posteriores.

ENTORNO DE DESARROLLO

El entorno de desarrollo utilizado ha sido Eclipse Java Neon y la versión utilizada de JDK ha sido JDK 1.8.0_60. El uso de este entorno de desarrollo se debe al alto grado de familiarización que tenemos con él, por lo que nos resulta sencillo de usar. Quizás la única desventaja con respecto a la herramienta propuesta, BlueJ, es la necesidad de crear una clase Main para poder ejecutar y comprobar cada método, aunque tampoco sea demasiado costoso. También, para ediciones de la misma, realización de la memoria y coordinación en el desarrollo utilizamos una herramienta de control de versiones, en nuestro caso nos decantamos por GitHub para el repositorio y SourceTree como herramienta de visualización de la misma, dada su facilidad de uso y la integración con IDEs como Eclipse o Visual Studio Code.

ACTIVIDAD 1: VERSIÓN ORIGINAL

En esta actividad, la especificación de la precondición se basa en dos puntos:

- El array de tiempos de inicio de las actividades se encuentre ordenado de manera decreciente

- El array de tiempos de fin se encuentra ordenado de forma creciente.

En el caso de la configuración escogida para nuestro algoritmo, basándonos en el ordenamiento creciente de los tiempos de fin, la lógica de la función de selección consiste en ir cogiendo cada vez la siguiente actividad, empezando desde el primero hasta el último. El primero siempre va a ser solución, puesto que no se va a solapar con ninguna actividad anterior (aún no hay), por lo tanto, esa posición se marca a TRUE en el array de booleanos que servirá como solución. Se pasa a la siguiente actividad y si es compatible también se marca a TRUE. Si no a FALSE. Hay que tener en cuenta que este proceso se repite en todas las actividades, pero comparando con la última actividad escogida. Por ejemplo, si hemos cogido las actividades 0 y 1, la actividad 2 se compararía con la 1. En caso de que no sea compatible se pasaría a la actividad 3, que se volvería a comparar con la actividad 1. Hay que tener en cuenta al estar ordenadas las actividades, asumimos que el índice para cada actividad es su correspondiente posición en el array.

La cabecera del método que se corresponde con esta actividad es:

```
public boolean[] seleccionActividades(int[] c, int [] f){...}
```

En cuanto a la complejidad de este algoritmo depende directamente del bucle que recorre el array comprobando qué actividades son compatibles, es decir, comprobando que el inicio de la actividad a evaluar sea superior o igual al tiempo de finalización de la anterior escogida.

Tal y como se ve en el código, a la función de selección le añadimos una segunda comprobación usando un OR. Dicha función se encarga de comprobar la casuística de una actividad que no produzca solapamiento por entrar de manera integra anterior temporalmente a la previa seleccionada. En este caso no se aplica dada la precondicion de ordenar temporalmente el array de entrada, sin embargo nos sirve como esqueleto de cara a los siguientes ejercicios, y dada la evaluación perezosa de Java EE, una vez se cumple la primera no destina recursos a la segunda condición, no afectando al rendimiento.

Por todo ello, la complejidad final es de O(n), pero sin embargo, alberga la desventaja de obligar al usuario a introducir el array de tiempos de finalización ordenado crecientemente.

ACTIVIDAD 1.5: UN PASO INTERMEDIO

Esta actividad no ha sido explícitamente pedida. La realización de la misma se justifica gracias a que creímos oportuno analizar un paso intermedio entre ambas actividades propuestas, pese a que supusiera un esfuerzo extra.

Consiste en lo siguiente: Dado que en la segunda actividad se propone el uso del algoritmo en el caso de que ninguno de los array esté ordenado, nos entro la duda acerca de que ocurriría con el rendimiento si usaremos un una forma de ordenarlos para seguir usando la misma función de selección del apartado

anterior, aunque con algunas modificaciones, de una forma tal, que el usuario no tenga que ordenar nada.

Para ello recibíamos ambos array, y mediante un array de índices auxiliar, conseguíamos obtener la posición de cada tiempo de fin ordenado. Con ello evitamos la complejidad de alterar el array original.

El método de ordenación utilizado para el array de índices fue ShellSort:

Este método compara cada elemento con el que está a un cierto número de posiciones, llamado salto, en lugar de compararlo con el que está justo a su lado.

Gracias a ello, este tipo de ordenación apenas se ve afectado por la distribución de los datos de entrada, tal y como se puede apreciar en el video adjunto.

Este salto es variable, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas con el mismo salto hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

El porqué de usar este método es porque es uno de los pocos con complejidad menor de $O(n^2)$, además del hecho de ser un algoritmo que hemos tratado menos en clase y sin embargo tiene una complejidad de $O(n^{4/3})$. De esta forma obtenemos una complejidad parecida a la pedida en la actividad 1, con unos datos de entrada iguales a los de la actividad 2.

Además, de esta forma podemos tener un análisis más exhaustivo del algoritmo y poder así comparar tres variantes de éste: algoritmo con vector ordenado vs. algoritmo con vectores desordenados y ordenación interna vs. algoritmo con vectores desordenados.

El método que se corresponde con esta actividad es:

public boolean*[] seleccionActividadesMejorado (****int*** *[]c,* ***int*** *[] f){...}*

La complejidad de este método depende de la complejidad del método de ordenación. Este método, tal y como explicamos anteriormente, no ha sido escogido en vano, pues hemos intentado escoger un método que tenga la menor complejidad.

A pesar de tener varios bucles con orden lineal, al no estar anidados la complejidad depende de la ordenación del array de índices.

ACTIVIDAD 2: VERSIÓN FINAL

Tras haber afrontado las dos actividades anteriores, en el que de una forma u otra se obtenía algún tipo de ordenación sobre los array, surge la necesidad de implementar un algoritmo que encuentre una solución al problema que no necesite de los array ordenados.

Para ello es necesario alterar la función de selección. La función que nosotros hemos escogido consiste en lo siguiente:

- Escoger la actividad que tenga un tiempo menor final.
- Seleccionar la siguiente con el tiempo de fin menor y comprobar si es compatible.
- Cuando se encuentra una actividad que sea compatible y que además tenga el menor tiempo posible, para lo cual la actividad, antes han podido ser descartadas algunas actividades con tiempo de fin menor, y se entra en la lógica de solución.

La cabecera del método que se corresponde esta actividad es:

```
public int seleccionActividadesSinOrden(int[] c, int [] f)
```

La complejidad de este método depende de dos bucles anidados uno que recorre los elementos para ir seleccionando candidatos y otro bucle que comprueba si el elemento seleccionado es de tiempo de fin mínimo. Por lo tanto, la complejidad es de orden cuadrático, $O(n^2)$.

RESULTADOS

A continuación, se presentan una serie de resultados en los que los algoritmos han sido ejecutados con distintos valores de entradas. Estos datos han sido recogidos mediante tablas .xls que automáticamente crea la herramienta OptimEx, propuesta en clase

RECORDATORIO:

Método seleccionActividades → Array de tiempos de fin ordenado.

Método seleccionActividadesMejorado → Arrays sin orden. Ordenación interna de índices.

Método seleccionActividadesSinOrden → Arrays sin orden. Función de selección tiempos de fin menores.

Los datos de entrada son los mismos para todos los casos, simplemente en el primer método está ordenado el array de tiempos de fin de manera creciente.

seleccionActividades	{0,7,5,2,16,11,23,15,12,24}, {3,8,11,18,20,21,24,24,25,29}	5
seleccionActividadesMejorado	{7,24,11,5,0,12,23,2,16,15}, {8,29,21,11,3,25,24,18,20,24}	5
seleccionActividadesSinOrden	{7,24,11,5,0,12,23,2,16,15}, {8,29,21,11,3,25,24,18,20,24}	5

Las actividades seleccionadas por nuestro algoritmo son {0,1,4,6,8}

Método	Datos de entrada	Resultado
seleccionActividades	{0,7,5,9,3,20}, {3,8,10,11,15,22}	4
seleccionActividadesMejorado	{0,7,5,9,3,20}, {3,8,10,11,15,22}	4
seleccionActividadesSinOrden	{0,7,5,9,3,20}, {3,8,10,11,15,22}	4

Las actividades seleccionadas por nuestro algoritmo son {0,1,3,5}

Método	Datos de entrada	Resultado
seleccionActividades	{0,2,7,3,8,10}, {3,7,8,9,10,11}	4
seleccionActividadesMejorado	{0,8,10,3,2,7}, {3,10,11,9,7,8}	4
seleccionActividadesSinOrden	{0,8,10,3,2,7}, {3,10,11,9,7,8}	4

Las actividades seleccionadas por nuestro algoritmo son {0,1,2,5}

Método	Datos de entrada	Resultado
seleccionActividades	{23,25,19,15,0}, {24,26,27,28,30}	2
seleccionActividadesMejorado	{0,15,19,25,23}, {30,28,27,26,24}	2
seleccionActividadesSinOrden	{0,15,19,25,23}, {30,28,27,26,24}	2

Las actividades seleccionadas por nuestro algoritmo son {3,4}

Método	Datos de entrada	Resultado
seleccionActividades	{1,4,5,6,3,0}, {3,17,17,17,17,18}	2
seleccionActividadesMejorado	{4,5,6,3,0,1}, {17,17,17,17,18,3}	2
seleccionActividadesSinOrden	{4,5,6,3,0,1}, {17,17,17,17,18,3}	2

Las actividades seleccionadas por nuestro algoritmo son {0,5}

Método	Datos de entrada	Resultado
seleccionActividades	{0,3,7,8,12,11}, {20,21,25,26,30,35}	1
seleccionActividadesMejorado	{0,3,7,8,12,11}, {20,21,25,26,30,35}	1
seleccionActividadesSinOrden	{0,3,7,8,12,11}, {20,21,25,26,30,35}	1

Las actividades seleccionadas por nuestro algoritmo son {0}

Método	Datos de entrada	Resultado
seleccionActividades	{0,2,4,5,7,8,7,12,10,11}, {2,4,6,6,8,9,10,13,15,17}	6
seleccionActividadesMejorado	{4,7,10,2,8,11,7,5,12,0}, {6,8,15,4,9,17,10,6,13,2}	6
seleccionActividadesSinOrden	{4,7,10,2,8,11,7,5,12,0}, {6,8,15,4,9,17,10,6,13,2}	6

Las actividades seleccionadas por nuestro algoritmo son {0,1,3,4,8,9}

Hay que tener en cuenta que todas estas soluciones están en función de los arrays desordenados.

CONCLUSIONES

Podríamos afirmar con total seguridad que el método más eficiente es aquel en el que se le pasa el array ordenado y el algoritmo lo espera como tal (Actividad 1), esto es importante dado que, si el algoritmo no está preparado para ello, por mucho que le pasemos el array ordenado, tiene que dedicar recursos a ordenarlo o buscar una alternativa eficaz para resolver el problema.
Sin embargo, al recibir los datos de entrada ordenados puede centrarse únicamente en recorrer el array y escoger aquellos que sean válidos para formar una solución.

Con respecto a la tercera implementación (Actividad 2), le facilita las cosas al usuario, puesto que éste se dedica simplemente a introducir los datos de entrada tal y como los recoja, sin preocuparse de qué hace el algoritmo por dentro. A cambio, la complejidad escala a $O(n^2)$, lo cual es un aumento considerable.

Por último, en la implementación intermedia (Actividad 1.5), podemos apreciar una solución en la cual el usuario puede introducir los datos sin ningún tipo de filtro, sin que la complejidad escale de manera drástica.

Por todo ello, se puede apreciar claramente, que, este algoritmo, cuanto más específico sea con respecto a los datos que va a utilizar, más eficaz va a ser procesando los mismos. Una vez lo hacemos general, si decidimos ordenarlo, la elección de un buen algoritmo de ordenación o de una buena función de selección, ejercen un gran papel a la hora de disminuir el ruido en los datos de entrada, permitiendo ignorar si los datos se encuentran prácticamente ordenados, totalmente desordenados, se repiten, o en cualquier otro tipo de configuración