Rev. Acad. Canar. Cienc., XIII (Núms. 1-2-3), 23-33 (2001) (publicado en Julio de 2002)

ESTIMACIÓN DEL TIEMPO VIRTUAL DE CPU DEL ALGORITMO DE DOS FASES DOBLEMENTE ESCALADO EN LAS CAPACIDADES

Antonio Sedeño Noda, Carlos González Martín y Sergio Alonso Rodríguez

Departamento de Estadística, Investigación Operativa y Computación (DEIOC)

Universidad de La Laguna, 38271-La Laguna, Tenerife, España.

Resumen

La complejidad de un algoritmo es la medida de eficiencia con la que éste halla la solución a un problema. Para que sea una medida válida que pueda ser usada para realizar con éxito comparaciones entre varios algoritmos y como estimador del tiempo de resolución, se ha de independizar de la máquina usada para el experimento computacional y de aspectos subjetivos dependientes del programador. La *complejidad en el caso peor* cumple con estas dos propiedades, y sin embargo, se basa en el establecimiento de cotas superiores para el tiempo de CPU sólo alcanzables por los problemas de condiciones más desfavorables. Por ello, se introduce entonces el concepto de *tiempo virtual de CPU*, que dota a los experimentos computacionales de las dos propiedades mencionadas a través de la localización de las denominadas *operaciones representativas* del algoritmo estudiado. En este trabajo se aplica este tipo de estudio, introducido por Ahuja, Magnanti y Orlin al algoritmo de dos fases doblemente escalado en las capacidades desarrollado por Sedeño Noda y González Martín.

Abstract

The complexity of an algorithm is the measurement of the efficiency with which the solution of a problem is found. A valid measurement can be successfully used to make comparisons between several algorithms and as a estimator of the time of resolution. Then, it is had to be independent of the machine selected to solve the problem, and the subjective aspects of the programmer. *Worst case complexity* fulfills these two properties, but nevertheless, it is based on the establishment of certain bounds only valid by rare problems. The concept of virtual CPU time is introduced by Ahuja, Magnanti and Orlin, to provide the two properties mentioned to the computational experiments through the specification of the denominated *representative operations* of the algorithm. In this work this complexity study is applied to the two-phases double capacities-scaling algorithm developed by Sedeño Noda and González Martín.

Palabras Claves: Complejidad caso peor, experiencia computacional, problema de flujo máximo.

1. INTRODUCCIÓN

En general, los sucesivos algoritmos introducidos para resolver un problema determinado persiguen la reducción de la complejidad teórica basada en el caso peor. Esta medida se calcula considerando el tiempo computacional necesario para resolver aquellos problemas particulares en los que el método se comporta peor. Sin embargo, estos casos patológicos pueden ser poco habituales, por lo que un algoritmo puede ser bueno en la práctica, aún cuando tenga una complejidad teórica no tan buena. Es por esto que el comportamiento empírico de un algoritmo es de gran interés.

La regla general en los estudios computacionales es la de considerar como única variable de medida el tiempo de CPU empleado en la resolución de una batería de problemas. Sin embargo, la consideración de este valor no da detalles sobre el comportamiento general en la práctica de un algoritmo. Más aún, el tiempo de CPU depende de factores tales como la propia máquina, el lenguaje utilizado, etc, que determinan que en este tiempo influyan otras cuestiones que no son específicas del algoritmo. Para evitar este problema, se introduce una medida basada en el concepto de *operaciones representativas* de

un algoritmo, cuyo recuento permite obtener el denominado *tiempo virtual de CPU*, (*CPUV*). Estos conceptos, que son debidos a Ahuja et al. [1], facilitan la comprensión de un algoritmo y proporcionan una medida fiable de su comportamiento.

En este trabajo calculamos el tiempo virtual de CPU para el algoritmo de dos fases doblemente escalado en las capacidades, propuesto por Sedeño-Noda y González- Martín [8] para el problema de flujo máximo. Este algoritmo utiliza para la mejora de la complejidad teórica una doble escala en las capacidades. Para la primera, se utiliza el factor de escala β que representa la base del sistema de numeración posicional utilizado en la representación de las capacidades. Tanto la complejidad teórica de este algoritmo como su comportamiento empírico dependen del mencionado factor. Por ello, la correspondiente estimación del tiempo virtual de CPU se realiza para los distintos valores de β que se han utilizado en el experimento.

Después de esta introducción, en la segunda sección recordamos las medidas teóricas usuales para estimar la bondad de un algoritmo en la práctica. En la tercera sección, introducimos los conceptos de operaciones representativas y de tiempo virtual de CPU. En la cuarta sección, formalizamos el problema de flujo máximo y exponemos, brevemente, el algoritmo de dos fases doblemente escalado en las capacidades. En la quinta sección, llevamos a cabo el estudio computacional sobre el algoritmo anterior donde se determina, a partir de las operaciones representativas del algoritmo, el tiempo virtual de CPU para varios valores de factor de escala β.

2. COMPLEJIDAD DEL CASO PEOR Y NOTACIÓN *O* GRANDE

Las diferentes instrucciones típicas de un algoritmo son instrucciones de asignación, aritméticas y lógicas. El número total de las mismas resulta de la suma de todas las instrucciones definidas anteriormente y determina el tiempo requerido en la ejecución del algoritmo.

La literatura especializada acepta ampliamente tres enfoques básicos para la medida de la bondad de un algoritmo: *análisis empírico, análisis caso-promedio y análisis del caso peor*, siendo este último, el más utilizado. El análisis del caso peor suministra una cota superior del número de pasos que un algoritmo realizaría para resolver el caso más desfavorable.

El tiempo de ejecución de un algoritmo depende de la naturaleza y tamaño de la entrada. Una función de complejidad de un algoritmo es una función del tamaño del problema que especifica el tiempo máximo que se necesitaría para resolver un caso particular del tamaño dado. En otras palabras, la función de complejidad da una medida de la proporción en que se incrementa el tiempo de resolución con respecto a un incremento en el tamaño del problema. Nos referiremos entonces a la función de complejidad del caso peor de un algoritmo a una cota superior del tiempo necesario para resolver un problema de un tamaño dado.

Para definir completamente la complejidad de un algoritmo necesitamos especificar además los valores para una o más constantes. En muchos casos, la determinación de esas constantes es una tarea no trivial, precisamente, porque su determinación podría depender del computador y de otros factores circunstanciales. Además, la dependencia de la función de complejidad de las constantes tiene todavía otros problemas: ¿Cómo comparamos un algoritmo que para un problema de tamaño n realiza $5n$ sumas y $3n$ comparaciones con un algoritmo que realiza n multiplicaciones y $2n$ restas?. Diferentes computadoras realizan operaciones aritméticas y lógicas a distintas velocidades. Debido a esto, ninguno de esos algoritmos será universalmente el mejor.

Estas dificultades son salvadas, ignorando las constantes en el análisis de la complejidad. Usaremos la notación *O grande* para remplazar expresiones como: "el algoritmo requiere un tiempo cnm para alguna constante c", por la expresión equivalente: "el algoritmo requiere un tiempo de orden nm, notacionalmente, O(nm)". Formalizaremos la definición como sigue:

Si f es una función definida en el conjunto de los números naturales, $\mathbb{N}$, diremos que un algoritmo tiene una función de complejidad O(f(n)) si existe una constante real c y un número natural n_0 ,tal que $\forall n \in \mathbb{N}, n \geq n_0$ el tiempo invertido por el algoritmo es, como mucho, $cf(n)$,.

La notación *O grande* tiene unas cuantas implicaciones adicionales. La complejidad de un algoritmo así medida es una cota superior del tiempo de ejecución del algoritmo para valores de n lo suficientemente grandes. Por lo tanto, esta notación indica sólo los términos más dominantes en el tiempo de ejecución y representa el hecho de que, para un n lo suficientemente grande, los términos con un crecimiento menor tienden a ser insignificantes si los comparamos con el término de mayor crecimiento.

Por ello, el análisis del caso peor es independiente del entorno computacional, es relativamente fácil de realizar y es definitivo en el sentido de que suministra conclusiones que permiten asegurar que un algoritmo es mejor que otro para el problema peor posible que pudiéramos encontrar.

Pero el análisis del caso peor no está exento de inconvenientes. Uno de sus mayores desventajas es que permite utilizar casos patológicos para determinar la eficiencia del algoritmo, aún cuando estos pudieran ser sumamente raros en la práctica. Por otro lado, este tipo de análisis garantiza una cota superior del número de pasos que puede realizar el algoritmo pero esconde información acerca de la resolución de casos no tan extremos, es decir, un algoritmo podría requerir para la resolución de muchos de sus problemas un número de pasos muy inferior al determinado por el análisis del caso peor.

3. RECUENTO DE OPERACIONES REPRESENTATIVAS

Como hemos comentado, el análisis del caso peor puede ser muy pesimista ya que utiliza casos patológicos para determinar la eficiencia de un algoritmo, aún cuando estos casos particulares se den con muy poca frecuencia. A menudo, el estudio empírico del comportamiento de un algoritmo es mucho más clarificador que su análisis del caso peor. Por esto, la comunidad investigadora valora la eficiencia práctica de un procedimiento teniendo en cuenta su comportamiento empírico.

La literatura existente sobre los tests computacionales tiene una tendencia excesiva a considerar el tiempo de CPU como primera medida de la eficiencia. En general, el tiempo de CPU depende de un subconjunto de detalles del entorno computacional tales como el lenguaje de programación elegido, el compilador usado, la potencia del computador; el estilo del programador; los problemas particulares seleccionados como casos de estudio; las combinaciones de los tamaños de la entrada,..., etc.. Debido a todo esto, los experimentos computacionales son a menudo difíciles de replicar en idénticas condiciones, lo que contradice el espíritu de la investigación científica.

Por otro lado, supongamos que un algoritmo realiza frecuentemente algunas operaciones fundamentales de forma repetida. Un típico análisis del tiempo de CPU no ayuda a identificar esas operaciones bloqueantes con lo que difícilmente podemos encontrar información para dirigir los esfuerzos futuros para entenderlo y mejorarlo.

Es por todo ello que hay que considerar medidas alternativas al uso de los valores de tiempo de CPU para el análisis empírico Para mejorarlo, podemos medir el comportamiento de un algoritmo contando el número de veces que ejecuta cada una de esas operaciones bloqueantes mientras resuelve casos particulares (*instances*) del problema. Es decir, debemos dirigir la experiencia computacional hacia el cálculo del número de operaciones bloqueantes realizadas, en lugar de obtener una cota superior teórica de este número.

Supongamos que la ejecución de cualquier línea de código requiere un tiempo $O(1)$ y que el código investigado tiene K líneas, $\{l_1, l_2, ..., l_K\}$. Entonces, dado un *instance* I del problema, sea $\alpha_I(k)$, con k=1,..., K el número de veces que se ejecuta la línea l_k para el instance I. Denotemos por $CPU(I)$ al tiempo de CPU en la resolución del instance I. Entonces, tenemos que $CPU(I) \in O\left(\sum_{k=1}^{K} \alpha_I(k)\right)$.

Sea S un subconjunto de $\{1,2,...K\}$ y sea L_S el subconjunto de líneas de código $\{l_i : i \in S\}$. Decimos que L_S es un conjunto de líneas de códigos representativas del programa, si para alguna constante c se tiene que para toda línea l_i del código $\alpha_I(i) \le c\left(\sum_{k \in S} \alpha_I(k)\right)$. En otras palabras, el tiempo

empleado en ejecutar la línea l_i está dominado por el tiempo empleado en la ejecución de las líneas de código en L_S. Dada esta definición, tenemos que $CPU(I) \in O\left(\sum_{k \in S} \alpha_I(k)\right)$.

Por lo tanto, el objetivo en el análisis empírico de un algoritmo es identificar las operaciones representativas y, para ello, realizar un recuento del número de veces que son ejecutadas. Algunas veces estas operaciones no se refieren específicamente a una línea de código sino a una operación del algoritmo que utiliza un número constante de líneas de código.

Utilizaremos también el concepto de *tiempo virtual de CPU* para un instance I que denotamos por $CPUV(I)$. El tiempo virtual de CPU de un algoritmo es una estimación lineal de su tiempo de CPU usando el recuento de las operaciones representativas, es decir, $CPUV(I) = \sum_{k \in S} c_k \alpha_I(k)$ para un conjunto de constantes c_k con k=1,...,$|S|$, tales que $CPUV(I)$ es la mejor estimación posible del tiempo de $CPU(I)$ para el instance I. Una posible vía de determinación de las constantes puede consistir en usar el análisis de regresión múltiple. Para ello, consideraríamos los vectores $(\text{CPU}(I), \alpha_I(1), ..., \alpha_I(|S|))$ generados en la resolución de varios *instances* para determinar las constantes que minimizan el error cuadrático $\sum_I (CPU(I) - CPUV(I))^2$.

Usar el tiempo virtual de CPU tiene varias ventajas. La primera es que nos permite identificar la proporción del tiempo que el algoritmo emplea en las diferentes operaciones representativas. Para ello basta con representar el cociente $\alpha_k(I)/CPUV(I)$ con k=1,...,$|S|$, lo que nos permitirá, de forma asintótica, identificar la operación *cuello de botella* que ralentiza la ejecución del algoritmo. Otra ventaja, es que las operaciones representativas son las mismas en cualquier computador y, por lo tanto, independientes de éste. Así, para determinar el tiempo virtual de CPU dado un computador tendríamos únicamente que estimar las constantes c_k con k=1,...,$|S|$.

4. PROBLEMA DE FLUJO MÁXIMO Y ALGORITMO DE DOS FASES DOBLEMENTE ESCALADO EN LAS CAPACIDADES

Con el fin de llevar a cabo una ilustración de la determinación del tiempo de CPU virtual de un algoritmo a partir del recuento de las operaciones representativas del mismo, introduciremos el algoritmo de dos fases doblemente escalado en las capacidades, propuesto por Sedeño-Noda y González-Martín [8] para resolver el problema de flujo máximo. Este algoritmo es una generalización del algoritmo de Sedeño-Noda y González-Martín [7].

El problema de flujo máximo fue resuelto por primera vez por Ford y Fulkerson[4]. Posteriormente, aparecen numerosos trabajos que tienden a mejorar la complejidad basada en el caso peor de los algoritmos precedentes. También se realizan diversos estudios computacionales que ponen de manifiesto las mejoras en la práctica de los mismos. Una bibliografía de consulta adecuada aparece en Ahuja et al. [1].

El problema de flujo máximo se formula de la manera siguiente: Dada una red dirigida, $G=(V,A)$, sea $V = \{1,..,n\}$ el conjunto de n nodos y A el conjunto de m arcos. Distinguimos dos nodos especiales en G: el nodo fuente s y el nodo sumidero t. Dado cualquier nodo i, definimos el conjunto $\text{Pred}(i) = \{j \in V / (j,i) \in A\}$ y el conjunto $\text{Suc}(i) = \{j \in V / (i,j) \in A\}$. Cada arco $(i,j) \in A$ tiene asociado una capacidad u_{ij} que denota la mayor cantidad de flujo que puede soportar el arco (i,j). Denotaremos por U al máximo valor del conjunto $\{u_{ij} / (i,j) \in A\}$.

Un *flujo* es una función $x : A \rightarrow R^+ \cup \{0\}$ que satisface:

$$\sum_{j \in Suc(i)} x_{ij} - \sum_{j \in Pred(i)} x_{ji} = \begin{cases} f & \text{si } i = s \\ 0 & i \in V - \{s,t\} \\ -f & \text{si } i = t \end{cases} \qquad (1)$$

$$0 \le x_{ij} \le u_{ij},\ (i,j) \in A \qquad (2)$$

para cierto $f \ge 0$. El problema de flujo máximo consiste en encontrar un flujo x tal que f es máximo.

A continuación expondremos brevemente el algoritmo de dos fases doblemente escalado en las capacidades (2FDEC) propuesto por Sedeño-Noda y González-Martín para resolver el problema de flujo máximo. El esquema del algoritmo es el siguiente:

```
Algoritmo 2FDEC;
{
X:=0;
Sea R(x) la red residual;
Δβ := β^⌈log_β U⌉ ;
Mientras Δβ > 1 hacer
     {
     2FEC( 2^⌈log β⌉ , U'(Δβ) ) ;
     Δβ := Δβ / β
     }
}
```

Se observa que la ejecución del algoritmo depende del parámetro β que define la base de la primera escala en las capacidades. El algoritmo 2FDEC llama $\lceil \log_\beta U \rceil$ veces al algoritmo 2FEC. Este último algoritmo es llamado con el parámetro $2^{\lceil \log \beta \rceil}$, debido a que la primera fase de escalado comienza con $\Delta = 2^{\lceil \log \beta \rceil}$ y con una capacidad residual máxima $U'(\Delta_\beta)$. El esquema del algoritmo 2FEC es el siguiente:

```
Algorithm 2FEC( Δ^ini, U);
{
Δ := Δ^ini;
Mientras Δ > 1 hacer
     {
     K(Δ) := min(n, 2(Un²/Δ)^1/3)
     {primera fase}
     Caminos_Incrementales_Mínimos(Δ, K(Δ));
     {segunda fase}
     Mientras R(Δ) contenga un camino P de s a t hacer
          {
          δ := min{ r_ij : (i,j) ∈ P } ;
          Enviar δ unidades de flujo a través de P
          }
     Δ := Δ/2
     }
}
```

En cada Δ-fase de escalado, el algoritmo 2FEC realiza a su vez dos fases. En la primera fase se aplica el algoritmo de caminos incrementales mínimos con la siguiente regla de parada: la primera fase termina cuando la etiqueta distancia del nodo fuente es mayor o igual a $K(\Delta) = \min(n, 2\left(Un^2/\Delta\right)^{1/3})$, es decir, cuando no existan caminos de longitud igual o inferior a $K(\Delta)$. En la segunda fase, se utiliza el

algoritmo de Ford y Fulkerson. Es decir, en esta fase, los caminos incrementales son identificados mediante un recorrido en profundidad en $R(\Delta)$ (red residual que únicamente contempla aquellos arcos con capacidad residual superior o igual a Δ) comenzando en el nodo fuente s.

Los siguientes teoremas establecen la complejidad basada en el caso peor para el algoritmo 2FDEC, dependiendo del valor de β.

Teorema 1. *El algoritmo 2FDEC, para* $\beta > n/8$, *requiere un tiempo* $O(nm \log U)$.

Teorema 2. *El algoritmo 2FDEC, para* $\beta \leq n/8$ *y para* $K(\Delta, \Delta_\beta) = \min(n, 2\left(Un^2/\Delta_\beta \Delta\right)^{1/3})$, *requiere un tiempo* $O(nm \log_\beta (U/n) \log \beta)) = O(nm \log(U/n))$.

Una explicación detallada de estos algoritmos, así como la demostración de sus complejidades se pueden encontrar en Sedeño-Noda y González-Martín [7],[8].

5. ESTUDIO COMPUTACIONAL

En este apartado presentamos un estudio computacional para el algoritmo 2FDEC, haciendo variar β en el conjunto {2,3,4,5,6,7,8,9}, con la intención de explicar el comportamiento empírico del algoritmo para los distintos valores de β. Además de medir el tiempo de CPU, consideramos el recuento de las operaciones representativas realizadas por el algoritmo.

El algoritmo ha sido codificado en Pascal estándar y ejecutado en una estación de trabajo 712/80 HP9000.

La complejidad teórica del algoritmo depende de los siguientes parámetros: número de nodos, número de arcos y capacidad máxima. Así, los casos particulares para el problema de flujo máximo, son redes generadas aleatoriamente con diferentes valores para esos parámetros. Hemos utilizado el generador NETGEN desarrollado por Klingman et al. [6] y el generador MFGEN desarrollado por nosotros.

El generador NETGEN requiere los siguientes parámetros: número de nodos, n, número de arcos, m, y la mínima y máxima capacidad de cualquier arco, c_1 y c_2, respectivamente. Entonces, las redes son generadas aleatoriamente con capacidades uniformemente distribuidas en $[c_1, c_2]$. En este experimento computacional, el intervalo seleccionado es $[1, U]$.

El generador MFGEN requiere los siguientes parámetros: n, m y la capacidad máxima U. Este generador trabaja en dos fases. En la primera fase, se construye un camino conectando el nodo 1 con los demás nodos. Este proceso emplea n-1 arcos. Por lo tanto, en la segunda fase, los restantes m-n+1 arcos son generados aleatoriamente. Las capacidades de los arcos están uniformemente distribuidas en el intervalo $[1, U]$.

Ambos generadores necesitan una semilla. Por lo tanto, la especificación de cualquier *instance* se completa con cuatro parámetros: *semilla*, n, m y U.

No hemos utilizado el generador RMFGEN dado por Goldfard y Grigoriadis [5] que no permite la generación de redes con un número de arcos superior a $6n$, ya que estamos interesados en un estudio computacional de mayor profundidad y necesitamos trabajar también con redes densas.

En la Tabla 1 se muestran los valores utilizados para el número de nodos (n), el número de arcos (m) y el cociente (d=m/n). En esta tabla se puede ver que, para cada valor del número de nodos, hemos elegido los valores del número de arcos. Estos valores vienen dados por: $10n$, $20n$, $30n$ y $40n$. Los valores de la máxima capacidad (U) han sido: 100, 10000, 1000000. Las combinaciones de todos los posibles valores de cada parámetro dan lugar a 48 casos particulares (4x4x3). Para cada una de estas posibles especificaciones de red, hemos considerado 10 réplicas, lo que nos da un total de 480 redes aleatorias. Las 10 semillas para cada una de las réplicas son:

n	m	d=m/n
250	2500, 5000, 7500, 10000	10, 20, 30, 40
500	5000, 10000,15000 20000	10, 20, 30, 40
750	7500, 15000, 22500,30000	10, 20, 30, 40
1000	10000, 20000, 30000, 40000	10, 20, 30, 40

Tabla 1. Número de nodos, de arcos y cociente d=m/n.

12345678, 36581249, 23456183, 46545174, 35826749, 43657679, 378484689, 23434767, 56563897 y 78656756. Cada una de estas especificaciones han sido la entrada de los dos generadores. Por lo tanto se han resuelto un total de 960 problemas para cada valor de β.

El tiempo de CPU en segundos, tomado por el algoritmo para resolver el problema, ha sido la variable seleccionada. Este tiempo no incluye el tiempo empleado en los procedimientos de entrada/salida. Con el fin de determinar las operaciones representativas del algoritmo de dos fases doblemente escalado en las capacidades, hemos medido el tiempo total de CPU empleado por el algoritmo de Ford y Fulkerson, (CPU_FF), es decir, el tiempo empleado en la fase 2 del algoritmo 2FEC. Por lo tanto, el tiempo total empleado por el algoritmo de caminos incrementales mínimos (fase 1) viene dado por CPU-CPU_FF. Para los problemas generados mediante NETGEN, el tiempo de CPU_FF es inferior al 5% del tiempo de CPU total, mientras que para los problemas generados mediante MFGEN no llega al 1.5%. Estas últimas afirmaciones son conclusiones de los resultados presentados en la Tabla 2. Por lo tanto, a la hora de determinar las operaciones representativas del algoritmo, únicamente tendremos en cuenta las que determinan el esfuerzo computacional de la primera fase del algoritmo. Para el caso de $\beta \leq n/8$ las mencionadas operaciones son:

		NETGEN				MFGEN			
n	β	CPU_FF	CPU	AR	AA	CPU_FF	CPU	AR	AA
250	2	0,02	0,66	16122,90	20245,73	0,00	0,56	12845,11	18275,23
250	3	0,02	0,53	14139,33	17920,45	0,00	0,46	11859,66	16849,77
250	4	0,04	0,49	16088,94	20207,23	0,02	0,43	12845,11	18275,23
250	5	0,02	0,45	15353,10	19254,78	0,00	0,40	12421,18	17556,89
250	6	0,02	0,41	14937,56	18903,47	0,00	0,36	12375,18	17577,74
250	7	0,01	0,38	15308,02	19375,47	0,00	0,34	12743,17	18049,36
250	8	0,03	0,40	16098,66	20197,41	0,01	0,36	12845,11	18275,23
250	9	0,02	0,39	14945,05	18919,51	0,00	0,36	12505,46	17738,42
500	2	0,04	1,36	16844,53	21775,73	0,00	1,16	24919,62	30140,29
500	3	0,04	1,10	14005,95	18478,68	0,00	0,96	21594,63	26431,64
500	4	0,08	1,04	16843,21	21771,52	0,06	0,88	24919,62	30140,29
500	5	0,04	0,94	14414,11	18970,29	0,00	0,82	22165,77	27007,73
500	6	0,04	0,85	15006,73	19583,05	0,00	0,76	23039,98	28040,78
500	7	0,03	0,80	16077,77	20918,90	0,00	0,71	23740,61	28778,66
500	8	0,06	0,84	16586,43	21494,91	0,04	0,72	24919,62	30140,29
500	9	0,04	0,84	14651,45	19341,31	0,01	0,72	22676,82	27685,47
750	2	0,06	1,97	28849,65	34189,67	0,00	1,72	25668,55	33110,03
750	3	0,06	1,62	25197,64	30090,18	0,00	1,44	19727,63	26320,28
750	4	0,13	1,50	28784,93	34112,06	0,06	1,33	25668,55	33110,03
750	5	0,05	1,36	25406,31	30346,23	0,00	1,25	20110,45	26827,29
750	6	0,05	1,23	26380,31	31531,60	0,00	1,16	22479,78	29415,64
750	7	0,04	1,18	26869,77	31980,19	0,00	1,08	22191,32	29141,01
750	8	0,11	1,25	28885,90	34168,23	0,04	1,14	25668,55	33110,03
750	9	0,06	1,22	26629,74	31751,30	0,00	1,12	20987,33	27940,11
1000	2	0,09	2,61	24810,88	30229,49	0,00	2,31	29219,48	35213,58
1000	3	0,08	2,11	21564,91	26494,87	0,00	1,95	24222,54	29768,12
1000	4	0,18	2,00	24810,88	30229,49	0,14	1,82	29219,48	35213,58
1000	5	0,08	1,81	21424,53	26395,98	0,00	1,73	24744,59	30207,15
1000	6	0,06	1,62	22970,44	28094,33	0,00	1,58	25707,27	31442,23
1000	7	0,07	1,59	23161,34	28533,11	0,00	1,48	26441,56	32105,11
1000	8	0,14	1,64	24801,63	30160,42	0,08	1,52	29219,48	35213,58
1000	9	0,08	1,58	22906,08	28150,39	0,01	1,49	25626,74	31386,16

Tabla 2. Media del tiempo de CPU, AA y AR para los generadores NETGEN y MFGEN.

AR: El número de arcos examinados en la actualización de las etiquetas distancias, es decir, el esfuerzo computacional total en dichas actualizaciones. En cada Δ-subfase de escalado, el algoritmo examina $O(K(\Delta,\Delta_\beta)m)$ arcos. Luego, el número total de arcos examinados esta acotado por $O(nm\log(U/n))$.

AA: El número de arcos examinados para la determinación de los caminos admisibles, es decir, el esfuerzo computacional total en los envíos de flujo. El esfuerzo total en esta operación vuelve a estar acotado por $O(nm\log(U/n))$.

Así, en la Tabla 2 se muestran los resultados experimentales para los problemas obtenidos mediante los dos generadores mencionados, dependiendo de la especificación de los valores de número de nodos (n) y el valor de β seleccionado. En dicha tabla aparece el tiempo medio de CPU, el tiempo

medio de CPU_FF, el número medio de arcos examinados en la actualización de las etiquetas distancias (*AR*) y el número medio de arcos examinados en la determinación de los caminos admisibles. Esta tabla ha sido obtenida al promediar la ejecución de los algoritmos en las réplicas, en el número de arcos (m) y en la capacidad (U). Por esto, únicamente aparecen en la tabla el número de nodos y el valor de β.

En la Tabla 2 se puede observar que el tiempo de CPU, para los problemas generados por NETGEN, es mayor que para los problemas generados por MFGEN. El comportamiento del tiempo de CPU frente a β es decreciente

La Figura 1, muestra el cociente *AA*/(*AA*+*AR*) frente al crecimiento del factor de escala para ambos generadores. En esta figura se puede observar que el número de arcos examinados para la identificación de caminos admisibles es siempre sensiblemente superior al número de arcos examinados en la actualización de las etiquetas distancias. Así, podemos decir que, con independencia del generador, aproximadamente el 56,5% de las operaciones en las que se examinan arcos, se realizan en la detección de caminos admisibles y el resto en las actualizaciones de las etiquetas distancias. Además, esta proporción permanece constante frente a los valores de β. Esto último es congruente con la teoría ya que el número de arcos examinados en las dos operaciones representativas depende de igual manera con respecto a β. Sin embargo, para el caso del algoritmo de escalado en las capacidades dado por Ahuja y Orlin [2], el número de arcos examinados para la admisibilidad aumenta y el número de arcos examinados para la actualización de las etiquetas decrece cuando aumenta β (ver Ahuja et al. [3]).

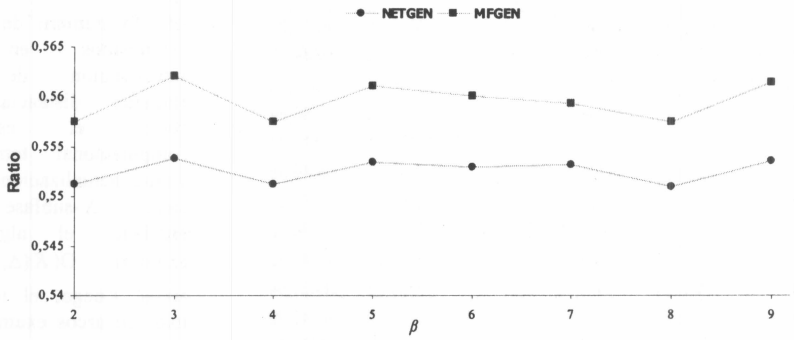

Figura 1. Cociente AA/(AA+AR) frente a β.

A continuación, procedemos a estimar el *tiempo virtual de CPU* del algoritmo para los distintos valores de β, con respecto a cada generador, como una función lineal de las operaciones representativas. Esto permitirá obtener una estimación del ratio de crecimiento del tiempo de CPU como una función polinomial de los nodos (n) y arcos (m). Para el algoritmo que presentamos, el tiempo de CPU virtual viene dado por $CPUV = \alpha_{AA} AA + \alpha_{AR} AR$. Para estimar las constantes α_{AA} y α_{AR} se realiza una regresión lineal para el tiempo de CPU (no incluye el tiempo de CPU_FF) frente a las variables independientes *AA* y *AR* para los distintos valores de β y para ambos generadores. La Tabla 3, muestra los valores obtenidos para las constantes del modelo, así como el valor del coeficiente de determinación R^2 para cada uno de ellos.

	NETGEN			MFGEN		
β	α_{AA}	α_{AR}	R^2	α_{AA}	α_{AR}	R^2
2	5,519e-5	0	0,877	0	5,870e-5	0,918
3	4,968e-5	0	0,874	4,474e-5	0	0,893
4	3,923e-5	0	0,866	3,388e-5	0	0,905
5	4,153e-5	0	0,866	3,850e-5	0	0,888
6	3,610e-5	0	0,873	3,397e-5	0	0,898
7	3,436e-5	0	0,876	0	3,939e-5	0,900
8	3,241e-5	0	0,869	2,879e-5	0	0,901
9	3,512e-5	0	0,876	3,277e-5	0	0,900

Tabla 3. Constantes para el tiempo de CPU virtual para los distintos valores de β. Un valor cero es conclusión del correspondiente test de hipótesis

En la Tabla 3, se observa que la expresión lineal del tiempo virtual de CPU (CPUV) para los problemas generadores por NETGEN únicamente depende del número de arcos examinados en la detección de admisibilidad, es decir, del término *AA*. Sin embargo, no ocurre lo mismo para los problemas generados mediante MFGEN. Para ellos, cuando β es igual a 2 y a 7, depende exclusivamente de *AR*; en otro caso, depende de *AA* (si alguna constante es igual a cero, esto es conclusión del correspondiente contraste de hipótesis adicional sobre la misma).

A continuación calculamos el ratio de crecimiento de las operaciones representativas que influyen en el tiempo virtual de CPU como una función polinomial del tamaño de la red. Esto permitirá estimar el tiempo virtual de CPU como una función del tamaño del problema. Supondremos que el modelo de regresión para *AA* y *AR* es, en ambos casos, $cn^{\alpha}d^{\gamma}$, donde $d = m/n$. Las constantes c, α y γ son estimadas por regresión lineal después de tomar logaritmos de *AA* y *AR*.

	NETGEN		MFGEN	
β	Estimación de AA ó AR	R^2	Estimación de AA ó AR	R^2
2	$AA = 28,91n^{0,33}d^{1,45}$	0,917	$AR = 6,16n^{0,58}d^{1,39}$	0,873
3	$AA = 15,38n^{0,34}d^{1,56}$	0,942	$AA = 20,41n^{0,35}d^{1,49}$	0,972
4	$AA = 28,71n^{0,33}d^{1,45}$	0,917	$AA = 16,22n^{0,48}d^{1,36}$	0,904
5	$AA = 194,09d^{1,46}$	0,896	$AA = 31,12n^{0,31}d^{1,45}$	0,968
6	$AA = 20,51n^{0,34}d^{1,51}$	0,905	$AA = 28,64n^{0,36}d^{1,38}$	0,939
7	$AA = 22,28n^{0,34}d^{1,45}$	0,931	$AR = 12,74n^{0,42}d^{1,44}$	0,938
8	$AA = 28,38n^{0,32}d^{1,46}$	0,915	$AA = 16,22n^{0,48}d^{1,36}$	0,904
9	$AA = 19,36n^{0,34}d^{1,51}$	0,925	$AA = 25,47n^{0,34}d^{1,45}$	0,970

Tabla 4. Estimación de AA ó AR para los distintos valores de β.

La Tabla 4, muestra la estimación de *AA* o *AR*, según aparezca ó no en el modelo lineal para el tiempo virtual de CPU (ver Tabla 3) para los distintos valores de β y los dos generadores.

A partir de las funciones obtenidas en la Tabla 4, se puede determinar el tiempo virtual de CPU sin más que multiplicar por la correspondiente constante dada en la Tabla 3. Así, en las Figuras 2a, 2b, 3a y 3b mostramos el cociente CPUV/CPU con el fin de observar la bondad de la estimación. Para ambos generadores, se puede observar que, cuando aumenta el producto *nd*, el cociente calculado se encuentra entre 0,8 y 1,1.

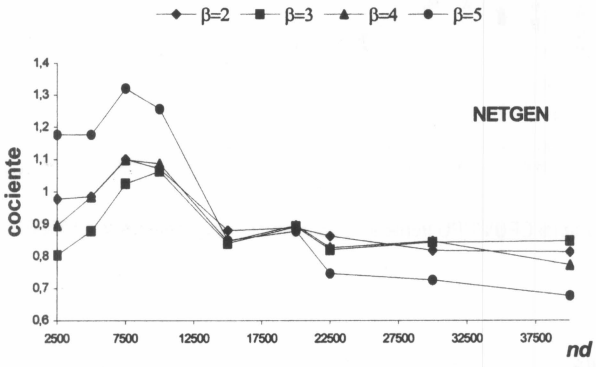

Figura 2a. Cociente CPUV/CPU frente a nd para problemas generados con NETGEN (β=2 hasta 5).

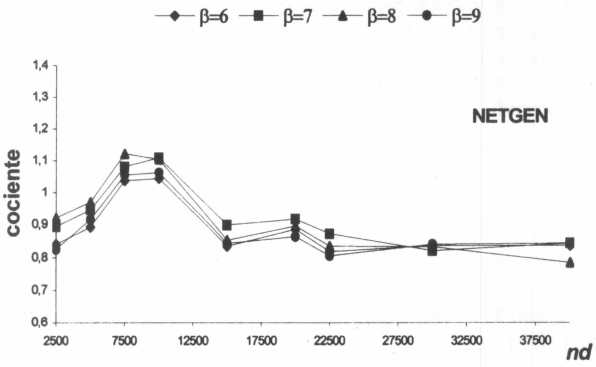

Figura 2b. Cociente CPUV/CPU frente a nd para problemas generados con NETGEN (β=6 hasta 9).

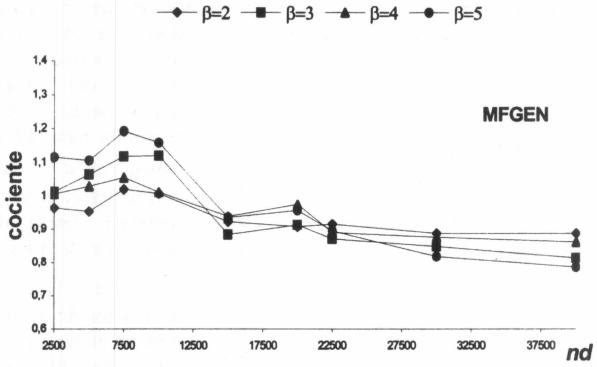

Figura 3a. Cociente CPUV/CPU frente a nd para problemas generados con MFGEN (β=2 hasta 5).

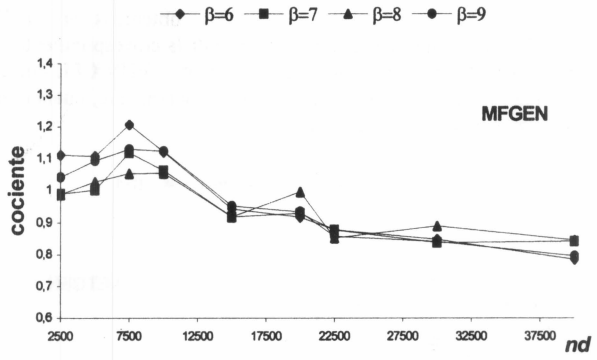

Figura 3b. Cociente CPUV/CPU frente a nd para problemas generados con MFGEN (β=6 hasta 9).

6. CONCLUSIONES

En este trabajo se ha estimado la función de tiempo virtual de CPU para el algoritmo de dos fases doblemente escalado en las capacidades para el problema de flujo máximo. Para él se han establecido dos conjuntos de instrucciones representativas en las acciones de detección de admisibilidad y en la actualización de etiquetas, y sobre ellas se ha calculado el ajuste de la función de complejidad. Finalmente la bondad de ajuste se ha comprobado comparándola con el tiempo de CPU cuyo cociente se encuentra próximo a la unidad.

BIBLIOGRAFÍA

[1] Ahuja R., Magnanti T. L. and Orlin J. B. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, inc. (1993)

[2] Ahuja R. and Orlin J. B. "Distance-Directed Augmenting Path algorithms for Maximum Flow and Parametric Maximum Flow problems". *Naval Research Logistics Quarterly* 38, 413-430 (1991).

[3] Ahuja R., Kodialam M., Mishra A. K. and Orlin J. B. "Computational investigations of maximim flow algorithms". *European Journal of Operational Research* 97, 509-542 (1997).

[4] Ford L. R. and Fulkerson D. R. "Maximal Flow through a Network". *Canadian Journal of Mathematics* 8, 399-404 (1956).

[5] Goldfard D. and Grigoriadis M. D. "A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow". *Annals of Operations Research* 13, 83-123 (1988).

[6] Klingman D., Napier A. and Stutz J. "Netgen: a Program for generating large scale capacitated Assignment, Transportation and Minimum Cost Flow Network problems". *Management Science* 20, 814-821 (1974).

[7] Sedeño-Noda A. and González-Martín C. "An $O(nm\log(U/n))$-Time Maximum-Flow Algorithm". *Naval Research Logistics Quarterly* 47 (6), 511-520 (2000).

[8] Sedeño-Noda A., González-Martín C. "A Two-Phase Double Capacity-Scaling Algorithm To Solve The Maximum Flow Problem". Enviado para su publicación a *TOP* (2001).