Tarea 2: Planificación de Trabajos

CC4102 - Diseño y Análisis de Algoritmos

Profesor:	Gonzalo Navarro
Auxiliar:	Teresa Bracamonte
Alumnos:	Cristián Carreño
	Sergio Maass
Fecha:	17 de Noviembre de 2013

1. Introducción

Los algoritmos online cumplen un rol muy importante al permitir resolver problemas a medida que su entrada se va recibiendo, o sea, sin conocer el total de los datos previamente. Pero además, son muy útiles para aproximar soluciones a problemas difíciles en tiempo razonablemente corto.

Diversos problemas de optimización pertenecientes a la clase NP admiten soluciones aproximadas mediante algoritmos online p-competitivos. Un algoritmo de estas características permite resolver instancias arbitrarias del problema en tiempo polinomial, garantizando que la solución estará dentro de un radio p del óptimo. Esto es muy importante, ya que si $P \neq NP$ -lo que parece ser el caso- no es posible encontrar una solución óptima en tiempo polinomial.

En este informe se presentarán dos versiones del problema de planificación de trabajos ($NP - completo$[1]) y se diseñará un algoritmo online 2-competitivo para cada una. Se demostrará formalmente la competitividad de cada algoritmo y luego se comprobará experimentalmente.

2. Trabajos de una etapa

El problema consiste en un conjunto de n trabajos que deben asignarse a m máquinas idénticas. Cada trabajo j requiere una cantidad de tiempo T_j para ser finalizado. Además, cada trabajo j sólo puede ser asignado a una máquina A_j, y debe ejecutarse por completo y sin interrupciones en esta.

Sea A una asignación de trabajos a máquinas, y sea el *makespan* definido por: $makespan(A) = max_i \sum_{A_j=i} T_j$. El objetivo del problema es determinar la asignación A que minimiza el *makespan*.

2.1. Algoritmo 2-competitivo propuesto

El algoritmo consiste simplemente en asignar cada trabajo j a la máquina que posea la menor carga. En particular, para cada j:

1. Obtener la máquina M_i de menor carga L_i.
2. Asignar j a M_i
3. $L_i := L_i + T_j$

La complejidad del algoritmo depende del paso 1. Usando un *heap* podemos obtener la máquina de menor carga en tiempo $O(1)$ y luego reinsertarla en la estructura con su nueva carga en tiempo $O(\log m)$. Por lo tanto, para n trabajos tenemos un tiempo $O(n \log m)$.

[1] Garey, M.R. (1976). "The Complexity of Flowshop and Jobshop Scheduling"

2.2. Demostración de competitividad

Sea L_i la carga de la máquina con mayor tiempo asignado, y sea j su último trabajo. La carga de la máquina antes de esa asignación era $L_i - T_j$. Debido a que el algoritmo asigna un trabajo a la máquina de menor carga tenemos que $L_i - T_j \leq L_k, \forall k \in [1, m]$.

Luego, sumando sobre todas las máquinas obtenemos

$$\begin{aligned} L_i - T_j &\leq \frac{1}{m} \sum_{k=1}^{m} L_k \\ &= \frac{1}{m} \sum_{k=1}^{n} T_k \\ &\leq L^* \end{aligned} \tag{1}$$

Donde L^* corresponde al *makespan* del caso óptimo.

Finalmente, considerando (1) y el hecho de que $T_j \leq L^*$, ya que el trabajo más largo igual debe ser realizado por alguna máquina en el caso óptimo, tenemos que

$$\begin{aligned} L_i &= L_i - T_j + T_j \\ &\leq L^* + T_j \\ &\leq 2L^* \end{aligned} \tag{2}$$

Con lo que se demuestra que el *makespan* obtenido por el algoritmo online está acotado por el doble del *makespan* en el caso óptimo. O sea, el algoritmo es 2-competitivo.

2.3. Comprobación empírica de competitividad

Para comprobar empíricamente la competitividad del algoritmo propuesto, éste se implementó en Java junto a un algoritmo que encuentra el makespan óptimo para este problema. Luego se hicieron pruebas con ambos algoritmos, usando los mismos datos, y se compararon los makespan obtenidos.

2.3.1. Implementación del algoritmo online

El algoritmo online se implementó tal cual fue descrito en la sección 2.1. Se utilizó la clase *PriorityQueue* de *java.utils*, la cual está implementada con un *heap*, para almacenar las máquinas ordenadas según carga. De este modo, se pueden planificar n tareas en m máquinas en tiempo $O(n \log m)$.

2.3.2. Implementación del algoritmo óptimo

Para obtener la asignación de una lista L de n trabajos a m máquinas que produce un menor makespan, se procedió a generar todas las posibles asignaciones que pueden entregar distinto makespan. Para esto se utilizó un arreglo A de tamaño n donde cada

índice $i \in [0, n-1]$ representa un trabajo, y cada valor A_i representa la máquina que se le ha asignado al trabajo i. Además se utilizó un objeto G encargado de generar asignaciones para A a través de su método $G.nextAssignment()$, y de señalar mediante un método $G.hasNext()$ si siguen habiendo asignaciones para generar. El algoritmo funciona del siguiente modo:

- Se comienza con A con todas sus entradas en cero.
- Se inicializa el entero $minMakespan := Integer.MAX\_INT$
- Mientras $G.hasNext()$ entregue verdadero:
  - Se asigna $A' := G.nextAssignment()$.
  - Se prueba la asignación A' con la lista L y m máquinas y se calcula su $makespan$.
  - Si $makespan(A') < minMakespan$:
    - Se copia A' en A.
    - $minMakespan := makespan(A')$.
- Ahora A contiene la asignación que minimiza el makespan.

El objeto G mantiene un arreglo B interno que va modificando para producir cada asignación. B se inicializa con todas sus entradas en 0, representando que todos los trabajos son asignados a la máquina 0. $G.nextAssignment()$ opera del siguiente modo:

- Se itera desde $i = n - 1$ hasta $i = 0$:
  - Si $i = 0$, desde ahora $G.hasNext()$ siempre retornará falso. Se sale del ciclo.
  - Si $B_i + 1 < m$, se asigna $B_i := B_i + 1$ y se sale del ciclo.
  - En otro caso, se asigna $B_i = 0$ y se continúa en el ciclo.
- Se retorna B, el cual representa una nueva asignación.

De esta forma se verifican todas las asignaciones posibles que comienzan asignándole el trabajo 0 a la máquina 0. No es necesario verificar otros casos ya que asignarle el primer trabajo a otra máquina no reducirá el makespan en ningún caso.

Este algoritmo genera m^{n-1} asignaciones y para cada una de ellas el cálculo del makespan para una lista dada toma tiempo $O(n)$. Por lo tanto, el algoritmo tiene una complejidad de $O(nm^{n-1})$. Cabe destacar que este largo tiempo podría mejorarse asegurándose de no generar asignaciones equivalentes ([0 2 2 1] y [0 1 1 2] son equivalentes porque siempre entregarán el mismo makespan), y también podría paralelizarse para aprovechar las arquitecturas multicore. Sin embargo, lo que interesa en este trabajo es comparar los makespan obtenidos por este algoritmo con los que obtiene la versión online, y no cuánto se demore, por lo que se dejará así.

2.3.3. Experimentos

Para comprobar la competitividad del algoritmo online se procedió a comparar los makespan que entrega con los que entrega el algoritmo óptimo para la misma lista de trabajos y número de máquinas. Para esto se generaron listas con trabajos de duración aleatoria entre 1 y 100, y se obtuvieron los makespan para las asignaciones encontradas por ambos algoritmos.

2.3.4. Resultados

Se probó con listas de tamaño 8, 10, 12, 14 y 16, y 4 máquinas. Para cada tamaño de lista se produjeron 400 listas y se compararon los makespan obtenidos por ambos algoritmos.

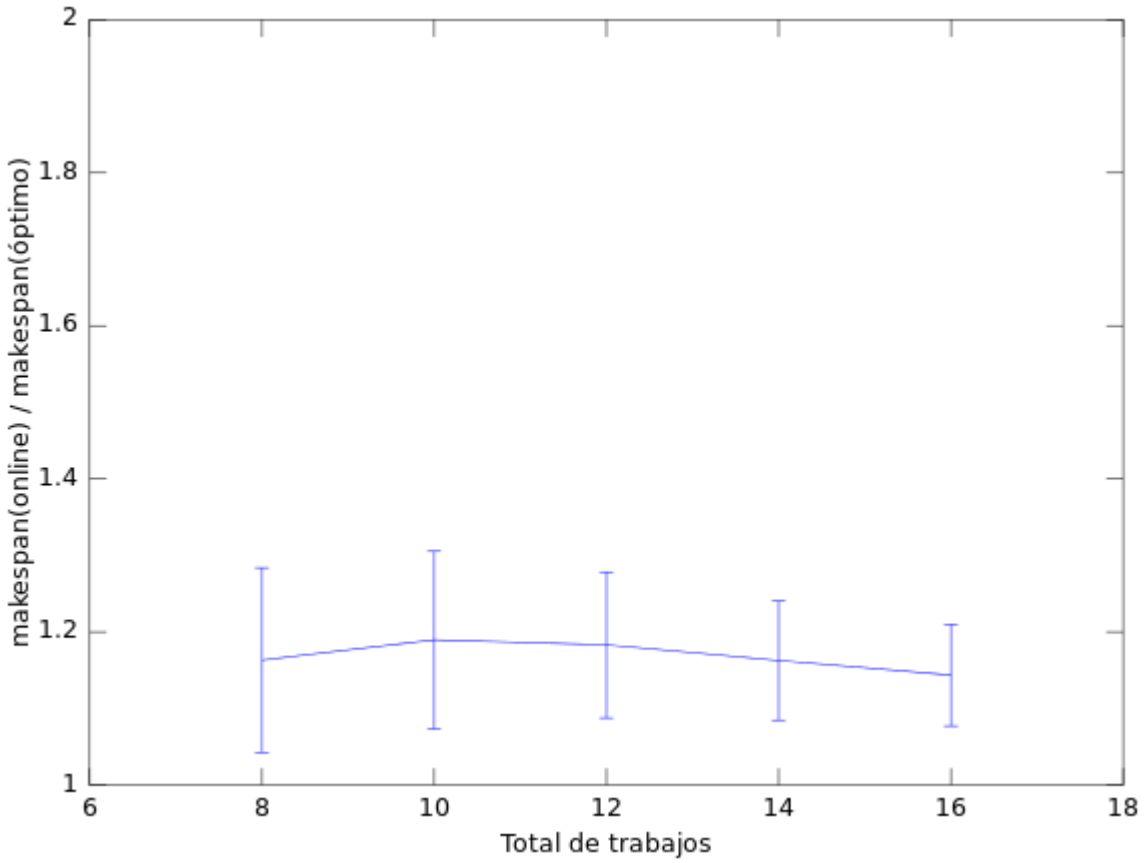

Figura 1: Comparación de *makespan* entre algoritmo online y algoritmo óptimo en función del número de trabajos, con 4 máquinas. Cada punto representa la media entre 400 datos, y las barras son la desviación estándar.

En la figura 1 se observa que los makespan obtenidos por el algoritmo online fueron mayores a los obtenidos por el algoritmo óptimo en un factor bastante menor a 2, que es la cota teórica calculada. Esta razón tiende a disminuir en promedio, y con desviaciones estándar cada vez más acotadas, a medida que se incrementa el número de trabajos.

Luego se realizó otra prueba similar pero variando el número de máquinas en vez del número de trabajos. Los resultados de este experimento se graficaron en la figura 2.

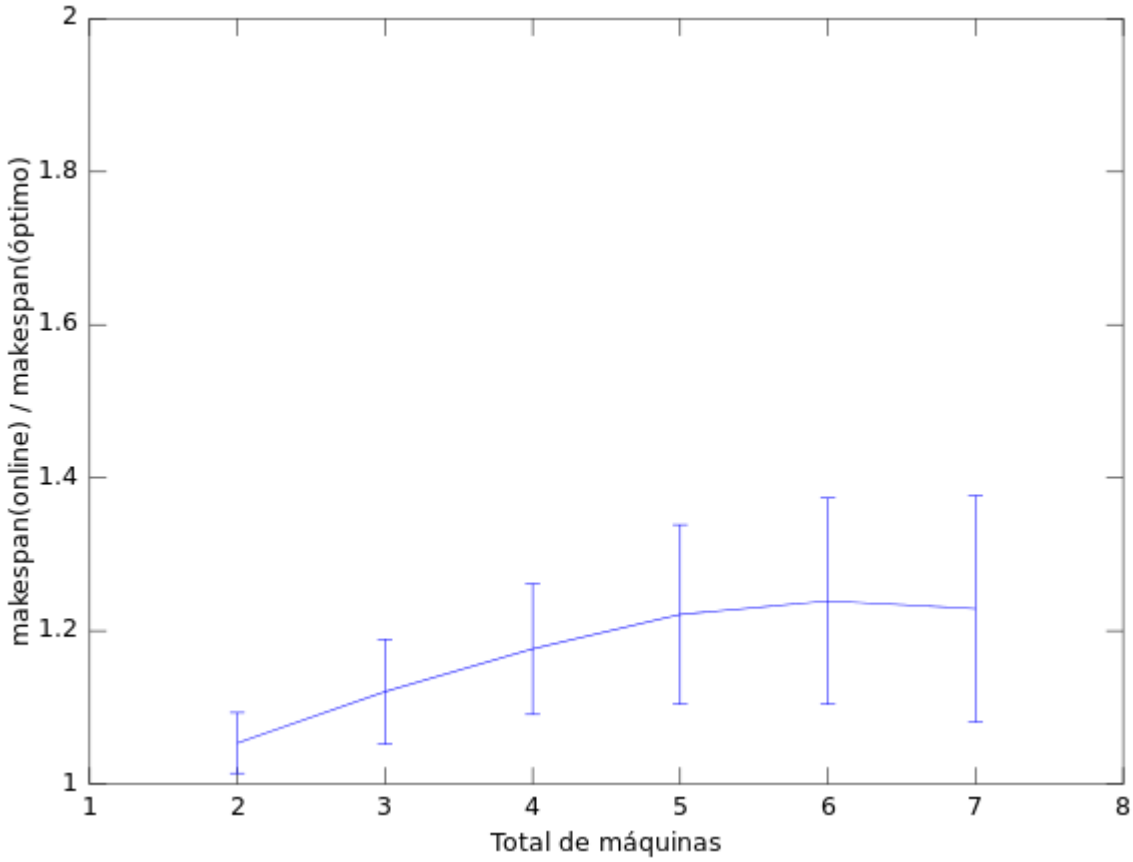

Figura 2: Comparación de *makespan* entre algoritmo online y algoritmo óptimo en función del número de máquinas, con 12 trabajos. Cada punto representa la media entre 400 datos, y las barras son la desviación estándar.

En la figura 2 se observa que al aumentar el número de máquinas la razón $\frac{makespan(online)}{makespan(optimo)}$ crece hasta alcanzar un máximo en $\frac{m}{2} = 6$ máquinas y luego comienza a disminuir. La disminución a partir de ese punto no es muy evidente debido al margen de error y a que sólo se realizó el experimento hasta 7 máquinas (debido al tiempo que tomaba calcular para $m \geq 8$), sin embargo tiene sentido ya que a medida que el número de máquinas se acerca al número de trabajos estas van quedando más desocupadas hasta que a cada máquina le toca exáctamente un trabajo, cuando $m = n$, con lo que $\frac{makespan(online)}{makespan(optimo)} = 1$. Por otra parte, se observa que incluso en el punto máximo, y considerando el margen de error, el rendimiento del algoritmo online con respecto al óptimo está considerablemente por debajo de la cota teórica de 2 calculada.

Extrapolando estos resultados podemos afirmar que el rendimiento del algoritmo online con respecto al óptimo suele ser bastante mejor que el peor caso teórico cuando procesa trabajos de duración aleatoria. Este rendimiento tiende a mejorar lentamente a medida que se incrementa el número de trabajos, y tiende a reducirse a medida que aumentan las máquinas hasta alcanzar un máximo en $m/2$, a partir de donde mejora progresivamente hasta igualar al óptimo en $m = n$.

3. Trabajos con múltiples etapas

El problema consiste en un conjunto de n trabajos de m etapas que deben ser procesados por m máquinas, cada etapa en una máquina distinta, una etapa a la vez. En particular:

- Cada trabajo j requiere un tiempo T_{jk} para ser procesado en la máquina k.
- Un trabajo no puede ejecutarse en dos máquinas en paralelo.
- Por cada trabajo se debe decidir en qué orden será enviado a cada máquina.
- El tiempo para trasferir un trabajo de una máquina a otra es nulo.
- Si para la máquina M_k existe un lapso de tiempo $[t1, t2]$ en el que no está ocupada y se recibe un trabajo j tal que $T_{jk} < t2 - t1$, se puede asignar el fragmento de trabajo j a la máquina k en el lapso vacío.

3.1. Algoritmo 2-competitivo propuesto

Dada una lista L de n trabajos, el algoritmo toma en orden cada trabajo j y procede del siguiente modo:

- Recorre todas las máquinas. Para cada máquina M_k, donde $k \in [0, m-1]$, se verifica si esta contiene un espacio cuyo tiempo sea menor o igual a T_{jk}, y si es así, entonces se asigna j a ese espacio.
- Las máquinas a las que no se les haya asignado un trabajo en el paso anterior se ordenan según su tiempo total.
- Mmm.. no estoy muy seguro de cómo sigue.

3.2. Demostración de competitividad

3.3. Comprobación empírica de competitividad

Del mismo modo que para el caso con trabajos de una etapa, para comprobar empíricamente la competitividad de este algoritmo se procedió a implementarlo en Java junto a un algoritmo que encuentra el makespan óptimo para este problema. Luego se hicieron pruebas con ambos algoritmos, usando los mismos datos, y se compararon los makespan obtenidos.

3.3.1. Implementación del algoritmo online

3.3.2. Implementación del algoritmo óptimo

Para obtener la asignación de una lista L de n trabajos de m etapas a m máquinas que produce el menor makespan, se generaron todas las posibles asignaciones. Para esto se utilizó un arreglo bi-dimensional A de tamaño $n \times m$, donde para cada trabajo $i \in [0, n-1]$, A_i es un arreglo de tamaño m tal que para todo $j \in [0, m-1]$, j representa

el orden en el cual el trabajo i debe ser procesado en la máquina $A_{i,j}$. Por ejemplo, si $A_{1,0} = 2$ y $A_{1,1} = 0$, entonces el trabajo 1 será procesado primero por la máquina 2 y después por la máquina 0.

Para generar las asignaciones se utilizó un objeto G encargado de generar asignaciones a través de su método $G.nextAssignment()$, y de señalar si hay más para generar a través de su método $G.hasNext()$, como en el caso de trabajos de una etapa.

El algoritmo funciona del siguiente modo:

- Se inicializa el entero $minMakespan := Integer.MAX\_INT$
- Mientras $G.hasNext()$ entregue verdadero:
  - Se asigna $A' := G.nextAssignment()$.
  - Se prueba la asignación A' con la lista L y m máquinas y se calcula su $makespan$.
  - Si $makespan(A') < minMakespan$:
    - Se copia A' en A.
    - $minMakespan := makespan(A')$.
- Ahora A contiene la asignación que minimiza el makespan.

El objeto G mantiene un arreglo bi-dimensional B interno que va modificando para producir cada asignación. Para esto, G posee una lista P de tamaño $m!$ con todas las permutaciones de los números entre 0 y $m - 1$. B se inicializa con todas sus entradas conteniendo la misma permutación (P_0), representando que todos los trabajos son asignados en el mismo orden a las máquinas. Además, se hace uso de un arreglo auxiliar I de tamaño n tal que para todo $i \in [0, n - 1]$, I_i representa el índice de la permutación en P que le corresponde al trabajo i ($B_i := P(I_i)$).

$G.nextAssignment()$ opera del siguiente modo:

- Se itera desde $i = n - 1$ hasta $i = 0$:
  - Si $i = 0$ y $I_0 = m!$, desde ahora $G.hasNext()$ siempre retornará falso. Se sale del ciclo.
  - Si $I_i + 1 < m!$, se asigna $I_i := I_i + 1$, $B_i := P(I_i)$, y se sale del ciclo.
  - En otro caso, se asigna $I_i = 0$, $B_i := P(I_i)$, y se continúa en el ciclo.
- Se retorna B, el cual representa una nueva asignación.

3.3.3. Experimentos

3.3.4. Resultados

4. Conclusiones