Diseño y Análisis de Algortimos para el problema Gestión Agrícola

Amanda Cordero Lezcano
Christopher Guerra Herrero

Contents

Chapter 1

Problema de Mochila Ilimitada

El problema **Unbounded Knapsack (UK)** es una extensión del clásico problema de la mochila (**Knapsack (K)**), donde los elementos pueden ser seleccionados múltiples veces. Este problema tiene aplicaciones en diversas áreas, como la logística, la planificación de recursos y la optimización de inventarios. En este texto, exploraremos la relación entre **UK** y **K**, y demostraremos que **UK** es un problema NP-duro mediante una reducción formal desde **UK** a **K**.

1.1 Demostración de que el problema UK es NP-duro

Para demostrar que el problema **Unbounded Knapsack (UK)** es NP-duro, realizaremos una reducción desde **UK** a **Knapsack (K)**, que es un problema NP-completo. La idea central es transformar una instancia de **UK** en una instancia equivalente de **K**, resolviendo subproblemas de manera iterativa hasta obtener una solución completa.

Definición del problema UK

Dada una instancia de **UK**, se tiene:

- Un conjunto de n elementos, donde cada elemento i tiene un peso w_i y un valor v_i.
- Una capacidad máxima de la mochila W.

El objetivo es maximizar el valor total de los elementos seleccionados sin exceder la capacidad W, permitiendo que cada elemento pueda ser seleccionado múltiples veces.

Reducción de UK a K

La reducción se realiza mediante el algoritmo 1.1

Algorithm 1 Reducción de UK a K

1. **Inicialización:**

   $S \leftarrow \emptyset$ {Conjunto de elementos seleccionados}

   $P \leftarrow 0$ {Peso total acumulado}

   $R \leftarrow W$ {Capacidad residual inicial}

2. **Iteración:**

while $R > 0$ **do**

   a. Resolver **UK** sobre el conjunto de elementos E con capacidad R.

   b. Sea S_{temp} el conjunto de elementos seleccionados (sin repeticiones) y P_{temp} el peso total de S_{temp}.

   if $S_{\text{temp}} = \emptyset$ **then**

      Break {No se pueden agregar más elementos}

   end if

   c. Actualizar:

      $S \leftarrow S \cup S_{\text{temp}}$

      $P \leftarrow P + P_{\text{temp}}$

      $R \leftarrow R - P_{\text{temp}}$

   d. Eliminar los elementos de S_{temp} de E: $E \leftarrow E \setminus S_{\text{temp}}$

end while

3. **Terminación:**

   Devolver S {Conjunto de elementos seleccionados}

Demostración de correctitud

Para demostrar que la reducción de **UK** a **K** es válida, debemos probar que la solución obtenida mediante el algoritmo descrito es equivalente a la solución óptima de **K**. A continuación, se detallan los aspectos clave de la demostración:

- **No repetición de elementos**: En cada iteración, el algoritmo selecciona un subconjunto de elementos **sin repeticiones**, es decir, cada elemento se incluye en la mochila a lo sumo una vez. Esto es consistente con la definición de **K**, donde los elementos no pueden repetirse.

- **Optimalidad local**: En cada paso, el algoritmo selecciona un subconjunto de elementos que maximiza el valor dentro de la capacidad residual disponible. Si no se eligiera la mejor combinación en una iteración, existiría otra selección con mayor valor que podría reemplazarla, contradiciendo la optimalidad de la solución. Por lo tanto, cada selección es localmente óptima.

- **Conjunto finito de elementos**: Dado que el conjunto de elementos es finito, el algoritmo termina en un número finito de pasos. En cada iteración, se elimina al menos un elemento del conjunto de candidatos, lo que garantiza que el proceso no continúe indefinidamente. Además, no es necesario utilizar todos los elementos disponibles, ya que solo se seleccionan aquellos que contribuyen a maximizar el valor sin exceder la capacidad de la mochila.

- **Complejidad de la transformación**: La reducción realiza una transformación en tiempo polinomial. En el peor caso, el algoritmo ejecuta **UK** una vez por cada elemento distinto, lo que resulta en una complejidad de $O(n \cdot T_{\text{UK}})$, donde T_{UK} es la complejidad de resolver **UK**. Si **UK** fuera resoluble en tiempo polinomial, entonces **K** también lo sería, ya que la transformación preserva la clase de complejidad.

- **Uso completo de la capacidad**: El algoritmo termina cuando no se pueden agregar más elementos sin exceder la capacidad de la mochila. Esto asegura que la capacidad W se utiliza de manera óptima, y la solución acumulada es equivalente a la solución óptima de **K**.

Mediante esta reducción, hemos demostrado que cualquier instancia de **K** puede ser transformada en una secuencia de instancias de **UK**. Dado que **K** es un problema NP-completo, concluimos que **UK** es NP-duro.

1.2 Algoritmo exacto

El problema de la **mochila ilimitada** consiste en seleccionar elementos con pesos y valores dados para maximizar el valor total sin exceder la capacidad W, permitiendo múltiples copias del mismo elemento.

Subestructura óptima

La propiedad clave es que la solución óptima para capacidad i puede construirse a partir de soluciones óptimas para capacidades menores. Si un elemento de peso w_j se incluye en la solución óptima para capacidad i, entonces el valor óptimo será $v_j + \text{DP}[i - w_j]$.

Idea intuitiva

Utilizamos un arreglo DP donde DP[i] almacena el valor máximo alcanzable con capacidad i. Comenzamos con capacidad 0 y progresivamente resolvemos capacidades mayores usando soluciones ya calculadas.

1.2.1 Programación dinámica

Estados y transiciones

- **Estado**: DP[i] representa el valor máximo para capacidad i.
- **Inicialización**: DP[0] = 0 (ningún elemento cabe en capacidad 0).
- **Transición**: Para cada capacidad i y elemento j:

$$\text{DP}[i] = \max\left(\text{DP}[i],\, v_j + \text{DP}[i - w_j]\right) \quad \text{si } w_j \leq i$$

Pseudocódigo Unbounded Knapsack

W, `weights`, `values`

Inicializar arreglo DP[$0 \ldots W$] con 0's

for $i = 1$ to W **do**
  for $j = 0$ to $n - 1$ **do**
    if `weights`[j] $\leq i$ **then**
      DP[i] $\leftarrow$ max(DP[i], DP[$i -$ `weights`[j]] + `values`[j])
    end if
  end for
end for
return DP[W]

Complejidad

- **Temporal**: $O(n \cdot W)$ (dos bucles anidados: W iteraciones externas, n internas).
- **Espacial**: $O(W)$ (solo se almacena un arreglo de tamaño $W + 1$).

1.2.2 Análisis del algoritmo

Correctitud

La demostración se basa en inducción matemática:

- **Caso base**: DP[0] = 0 es trivialmente correcto.
- **Hipótesis inductiva**: Supongamos que DP[k] es óptimo para todo $k < i$.
- **Paso inductivo**: Para capacidad i, consideramos todos los elementos j con $w_j \leq i$. Si el elemento j es parte de la solución óptima, entonces:

$$\text{DP}[i] = v_j + \text{DP}[i - w_j]$$

Por hipótesis inductiva, DP[$i - w_j$] ya es óptimo. Al maximizar sobre todos los j, garantizamos que DP[i] también es óptimo.

Optimalidad del orden

El procesamiento en orden ascendente de capacidades asegura que al calcular DP[i], todos los valores DP[k] para $k < i$ ya han sido computados correctamente, permitiendo reutilizar soluciones previas.

1.3 Algoritmo de 2-aproximación

Con información de [3]

Comencemos por describir un algoritmo que garantiza una solución dentro de un factor de 2 del valor óptimo. Supongamos que el artículo j tiene la mayor densidad de beneficio, definida como p_i/w_i, donde p_i es el beneficio del artículo i y w_i es su peso. Denotamos z_L como el valor objetivo obtenido al seleccionar tantas unidades del artículo j como sea posible sin exceder la capacidad de la mochila C. Formalmente:

$$z_L = p_j \lfloor C/w_j \rfloor,$$

donde $\lfloor C/w_j \rfloor$ representa el número máximo de unidades enteras del artículo j que caben en la mochila.

Es importante notar que si se relaja la restricción de integridad (es decir, si se permite fraccionar los artículos), el valor objetivo óptimo sería:

$$z = p_j(C/w_j),$$

el cual constituye una cota superior para el valor óptimo z^* bajo las restricciones originales. Por lo tanto, se cumple que:

$$z_L \leq z^* \leq z.$$

Además, dado que $z = z_L + p_j\epsilon$, donde $0 \leq \epsilon < 1$, se deduce que:

$$z \leq 2z_L.$$

En consecuencia, tenemos que:

$$z^* \leq 2z_L,$$

lo cual implica que el algoritmo que selecciona únicamente el artículo con mayor densidad de beneficio y llena la mochila con este tipo de elementos constituye una 2-aproximación al valor óptimo del problema.

Chapter 2

Problema de Scheduling

Con información de [1] y [2]

2.1 Load Balancing

Formulamos el **Problema de Load Balancing** de la siguiente manera. Se nos da un conjunto de m máquinas $M_1, \ldots, M_m$ y un conjunto de n trabajos; cada trabajo j tiene un tiempo de procesamiento t_j. Buscamos asignar cada trabajo a una de las máquinas de manera que las cargas colocadas en todas las máquinas estén lo más *balanceadas* posible.

Más concretamente, en cualquier asignación de trabajos a máquinas, podemos denotar por $A(i)$ el conjunto de trabajos asignados a la máquina M_i. Bajo esta asignación, la máquina M_i necesita trabajar un tiempo total de

$$T_i = \sum_{j \in A(i)} t_j,$$

y declaramos que esta es la carga en la máquina M_i. Buscamos minimizar una cantidad conocida como el *makespan*, que es simplemente la carga máxima en cualquier máquina:

$$T = \max_i T_i.$$

Nuestro problema de optimización tiene una versión de decisión: Dado un valor M, ¿existe una asignación de tareas tal que el makespan sea $\leq M$? A continuación nos apoyaremos del problema de desición y del conocido problema Partition para demostrar que la versión de optimización de Load Balancing es NP duro.

2.1.1 Demostración de que Load Balancing es NP duro

Teorema 1. *El problema de optimización de Load Balancing es NP duro*

Proof. Para demostrar que el problema de decisión es NP-completo, realizamos una reducción desde el problema **Partition**, primero demostremos que este es NP-completo a su vez reduciéndolo a Subset-Sum.

Demostración de que Partition es NP completo

Problema Partition: Dado un conjunto de números $\{p_1, p_2, \ldots, p_n\}$, ¿es posible dividirlo en dos subconjuntos disjuntos A y B tales que:

$$\sum_{p_i \in A} p_i = \sum_{p_i \in B} p_i?$$

Sea $P = \sum_{i=1}^{n} p_i$. El problema puede reformularse como: ¿existe una partición tal que:

$$\sum_{p_i \in A} p_i = \frac{P}{2}?$$

Para demostrar que **Partition** es NP-completo, seguimos los siguientes pasos:

Paso 1: Partition está en NP

Un problema está en NP si, dada una solución candidata, podemos verificar su validez en tiempo polinomial.

Una solución candidata para **Partition** consiste en un subconjunto $A \subseteq S$. Para verificar si A es una solución válida:
- Calculamos suma$(A) = \sum_{x \in A} x$.
- Verificamos si suma$(A) = \frac{\sum_{i=1}^{n} a_i}{2}$.
- Esto se puede hacer en tiempo polinomial con respecto al tamaño de S.

Por lo tanto, **Partition** está en NP.

Paso 2: Partition es NP-duro

Para demostrar que **Partition** es NP-duro, realizamos una reducción polinomial desde el problema **Subset-Sum**, que es conocido como NP-completo.

Problema Subset-Sum

El problema **Subset-Sum** se define como sigue:

Entrada: Un conjunto finito $S = \{a_1, a_2, \ldots, a_n\}$ de números enteros positivos y un objetivo T.

Pregunta: ¿Existe un subconjunto $A \subseteq S$ tal que:

$$\sum_{x \in A} x = T?$$

Reducción desde Subset-Sum a Partition

Dada una instancia del problema **Subset-Sum** con conjunto $S = \{a_1, a_2, \ldots, a_n\}$ y objetivo T, construimos una instancia del problema **Partition** como sigue:

1. Calcula la suma total de los elementos de S:

$$P = \sum_{i=1}^{n} a_i.$$

2. Agrega un nuevo número b al conjunto S, donde:

$$b = |P - 2T|.$$

3. Define el nuevo conjunto $S' = S \cup \{b\}$.

Ahora preguntamos: ¿Es posible particionar S' en dos subconjuntos A y B tales que:

$$\sum_{x \in A} x = \sum_{y \in B} y?$$

En tal caso la cardinalidad de los subconjuntos será

$$\frac{P + |P - 2T|}{2}$$

Por lo que uno de los dos tendrá T de cardinalidad al excluir a b

Reducción desde Parition a Load Balancing

Dada una instancia del problema Load Balancing de decisión, construimos el problema Partition como sigue:

- Sea $m = 2$ (dos máquinas).
- Asigna los tiempos de procesamiento $p_1, p_2, \ldots, p_n$ a las tareas.
- Define $T = \frac{P}{2}$.

Ahora preguntamos: ¿Existe una asignación de tareas a las dos máquinas tal que el makespan sea $\leq T$?

Equivalencia

Si existe una asignación de tareas entonces habrá una partición. Si no existe, no existirá la partición. Por lo tanto, resolver el problema de decisión de Load Balancing permite resolver el problema Partition. Como el problema Partition es NP-completo y hemos mostrado que cualquier instancia de Partition puede reducirse en tiempo polinomial a una instancia del problema de decisión de Load Balancing, concluimos que el problema de decisión de Load Balancing es NP-completo.

Finalmente, demostramos que el problema de optimización de Load Balancing es NP-duro. Para ello, notamos que:

1. Si podemos resolver el problema de optimización (encontrar el makespan mínimo T^*), entonces podemos resolver el problema de decisión para cualquier valor T comparando T con T^*:

$$\text{Respuesta al problema de decisión} = \begin{cases} \text{sí}, & \text{si } T \geq T^*, \\ \text{no}, & \text{si } T < T^*. \end{cases}$$

2. Como el problema de decisión es NP-completo, resolver el problema de optimización implica resolver un problema NP-completo.

Por lo tanto, el problema de optimización de Load Balancing es NP-duro. □

2.1.2 Algoritmos de aproximación

Hemos demostrado que el problema de Load Balancing (en su versión de optimización) es NP-duro mediante una reducción desde el problema Partition al problema de decisión, y luego conectando el problema de decisión con el problema de optimización. Ahora vamos a ver algunos algoritmos para aproximar la solución del mismo.

Greedy-Balance

Primero consideramos un algoritmo greedy muy simple para el problema. El algoritmo hace un recorrido por los trabajos en cualquier orden; cuando llega al trabajo j, lo asigna a la máquina cuya carga sea la más pequeña hasta ese momento.

Algorithm 2 Greedy-Balance

Empezar sin trabajos asignados

for cada máquina M_i **do**

  $T_i \leftarrow 0$ {Inicializar la carga de la máquina}

  $A(i) \leftarrow \emptyset$ {Inicializar el conjunto de trabajos asignados a M_i}

end for

for cada trabajo $j = 1, \ldots, n$ **do**

  Sea M_i la máquina con la menor carga

  Asignar el trabajo j a la máquina M_i

  $A(i) \leftarrow A(i) \cup \{j\}$

  $T_i \leftarrow T_i + t_j$

end for

Análisis del Algoritmo

Sea T el makespan de la asignación resultante; queremos mostrar que T no es mucho mayor que el makespan mínimo posible T^*. Por supuesto, al intentar hacer esto, nos encontramos inmediatamente con el problema básico mencionado anteriormente: necesitamos comparar nuestra solución con el valor óptimo T^*, aunque no sabemos cuál es este valor y no tenemos forma de calcularlo. Para el análisis, por lo tanto, necesitaremos una cota inferior para el óptimo, una cantidad con la garantía de que, no importa cuán bueno sea el óptimo, no puede ser menor que esta cota.

Hay muchas posibles cotas inferiores para el óptimo. Una idea para una cota inferior se basa en considerar el tiempo total de procesamiento $\sum_j t_j$. Una de las m máquinas debe hacer al menos una fracción $1/m$ del trabajo total, y por lo tanto tenemos lo siguiente:

$$T^* \geq \frac{1}{m} \sum_j t_j. \tag{2.1}$$

Hay un tipo particular de caso en el que esta cota inferior es demasiado débil para ser útil. Supongamos que tenemos un trabajo que es extremadamente largo en relación con la suma de todos los tiempos de procesamiento. En una versión suficientemente extrema de esto, la solución óptima colocará este trabajo en una máquina por sí solo, y será el último en terminar. En tal caso, nuestro algoritmo greedy en realidad produciría la solución óptima, pero la cota inferior en (2.1) no es lo suficientemente fuerte para establecer esto.

Esto sugiere la siguiente cota inferior adicional para T^*:

$$T^* \geq \max_j t_j. \tag{2.2}$$

Ahora estamos listos para evaluar la asignación obtenida por nuestro algoritmo greedy.

Teorema 2. *El algoritmo* ***Greedy-Balance*** *produce una asignación de trabajos a máquinas con un makespan $T \leq 2T^*$.*

Proof. Aquí está el plan general para la demostración. Al analizar un algoritmo de aproximación, se compara la solución obtenida con lo que se sabe sobre el óptimo; en este caso, nuestras cotas inferiores (2.1) y (2.2). Consideramos una máquina M_i que alcanza la carga máxima T en nuestra asignación, y nos preguntamos: ¿Cuál fue el último trabajo j asignado a M_i? Si t_j no es demasiado grande en relación con la mayoría de los otros trabajos, entonces no estamos muy por encima de la cota inferior (2.1). Y, si t_j es un trabajo muy grande, entonces podemos usar (2.2).

Cuando asignamos el trabajo j a M_i, la máquina M_i tenía la carga más pequeña de cualquier máquina; esta es la propiedad clave de nuestro algoritmo greedy. Su carga justo antes de esta asignación era $T_i - t_j$, y como esta era la carga más pequeña en ese momento, se sigue que todas las máquinas tenían

una carga de al menos $T_i - t_j$. Así, sumando las cargas de todas las máquinas, tenemos:

$$\sum_k T_k \geq m(T_i - t_j),$$

o equivalentemente,

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k.$$

Pero el valor $\sum_k T_k$ es simplemente la carga total de todos los trabajos $\sum_j t_j$ (ya que cada trabajo se asigna exactamente a una máquina), y por lo tanto la cantidad en el lado derecho de esta desigualdad es exactamente nuestra cota inferior en el valor óptimo, de (2.1). Así,

$$T_i - t_j \leq T^*.$$

Ahora consideramos la parte restante de la carga en M_i, que es solo el trabajo final j. Aquí simplemente usamos la otra cota inferior que tenemos, (2.2), que dice que $t_j \leq T^*$. Sumando estas dos desigualdades, vemos que:

$$T_i = (T_i - t_j) + t_j \leq 2T^*.$$

Como nuestro makespan T es igual a T_i, este es el resultado que queremos. □

Sorted-Balance

Ahora pensemos en cómo podríamos desarrollar un algoritmo de aproximación mejor, es decir, uno para el cual siempre estemos garantizados de estar dentro de un factor estrictamente menor que 2 del óptimo. Para hacer esto, es útil pensar en los peores casos para nuestro algoritmo de aproximación actual.

Un ejemplo malo para nuestro algortimo greedy pudiera tener la siguiente característica: distribuimos todo de manera muy uniforme entre las máquinas, y luego llega un último trabajo gigante. Intuitivamente, parece que ayudaría organizar primero los trabajos más grandes de manera adecuada, con la idea de que, más tarde, los trabajos pequeños solo pueden causar un daño limitado. Y, de hecho, esta idea conduce a una mejora cuantificable.

Así, ahora analizamos la variante del algoritmo greedy que primero ordena los trabajos en orden decreciente de tiempo de procesamiento y luego procede como antes. Demostraremos que la asignación resultante tiene un makespan que es, como máximo, 1.5 veces el óptimo.

La mejora proviene de la siguiente observación. Si tenemos menos de m trabajos, entonces la solución greedy claramente será óptima, ya que coloca cada trabajo en su propia máquina. Y si tenemos más de m trabajos, entonces podemos usar la siguiente cota inferior adicional para el óptimo.

$$T^* \geq 2t_{m+1}. \tag{2.3}$$

Algorithm 3 Sorted-Balance

Iniciar sin trabajos asignados.

Asignar $T_i = 0$ y $A(i) = \emptyset$ para todas las máquinas M_i.

Ordenar los trabajos en orden decreciente de tiempos de procesamiento t_j.

Suponer que $t_1 \geq t_2 \geq \cdots \geq t_n$.

for cada trabajo $j = 1, \ldots, n$ **do**

  Sea M_i la máquina que alcanza el mínimo $\min_k T_k$.

  Asignar el trabajo j a la máquina M_i.

  Actualizar $A(i) \leftarrow A(i) \cup \{j\}$.

  Actualizar $T_i \leftarrow T_i + t_j$.

end for

Proof. Considera solo los primeros $m+1$ trabajos en el orden clasificado. Cada uno de ellos toma al menos un tiempo t_{m+1}. Hay $m+1$ trabajos y solo m máquinas, por lo que debe haber una máquina a la que se le asignen dos de estos trabajos. Esta máquina tendrá un tiempo de procesamiento de al menos $2t_{m+1}$. □

Teorema 3. *El algoritmo* ***Sorted-Balance*** *produce una asignación de trabajos a máquinas con un makespan* $T \leq \frac{3}{2}T^*$.

Proof. La demostración será muy similar al análisis del algoritmo anterior. Como antes, consideraremos una máquina M_i que tiene la carga máxima. Si M_i solo contiene un trabajo, entonces el programa es óptimo.

Supongamos que la máquina M_i tiene al menos dos trabajos, y sea t_j el último trabajo asignado a la máquina. Nótese que $j \geq m+1$, ya que el algoritmo asignará los primeros m trabajos a m máquinas distintas. Por lo tanto, $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, donde la segunda desigualdad es (2.3).

Ahora procedemos como en la demostración de (3), con el siguiente cambio. Al final de esa demostración, teníamos las desigualdades $T_i - t_j \leq T^*$ y $t_j \leq T^*$, y las sumamos para obtener el factor de 2. Pero en nuestro caso aquí, la segunda de estas desigualdades es, de hecho, $t_j \leq \frac{1}{2}T^*$; por lo tanto, sumar las dos desigualdades nos da la cota:

$$T_i \leq \frac{3}{2}T^*.$$

□

2.2 Problema de Load Balancing con Precedencia

El problema de **Load Balancing con Precedencia** se define como sigue:

Entrada:

- Un conjunto de n tareas $T = \{t_1, t_2, \ldots, t_n\}$.
- Cada tarea t_i tiene un tiempo de procesamiento $p_i > 0$.
- Un grafo dirigido acíclico (DAG) $G = (T, E)$ que representa las restricciones de precedencia. Una arista $(t_i, t_j) \in E$ indica que la tarea t_i debe completarse antes de que t_j pueda comenzar.
- m procesadores idénticos disponibles para ejecutar las tareas.

Salida:

- Una asignación de tareas a procesadores y un horario que minimice el **makespan** $C_{\max}$, definido como el tiempo máximo de finalización de todas las tareas:
$$C_{\max} = \max_{j=1,\ldots,m} \sum_{t_i \in S_j} p_i,$$
donde S_j es el conjunto de tareas asignadas al procesador j.
- El horario debe respetar las restricciones de precedencia: si $(t_i, t_j) \in E$, entonces t_i debe completarse antes de que t_j comience.

2.2.1 Demostración de que Load Balancing con Precedencia es NP-duro

Para demostrar que el problema de **Load Balancing con Precedencia** es NP-duro, realizamos una reducción desde el problema de **Load Balancing sin precedencia**.

Construcción de la Instancia

Dada una instancia del problema de **Load Balancing sin precedencia**, construimos una instancia del problema de **Load Balancing con Precedencia** como sigue:

- Mantenemos el mismo conjunto de tareas T y sus tiempos de procesamiento $p_1, p_2, \ldots, p_n$.
- No añadimos ninguna restricción de precedencia, es decir, el grafo $G = (T, E)$ es vacío ($E = \emptyset$).

Equivalencia

- Si podemos resolver el problema de **Load Balancing con Precedencia** (con $E = \emptyset$), entonces también podemos resolver el problema de **Load Balancing sin precedencia**.
- Por lo tanto, ambos problemas son equivalentes cuando no hay restricciones de precedencia.

Complejidad de la Reducción

La construcción de la instancia del problema de **Load Balancing con Precedencia** a partir del problema de **Load Balancing sin precedencia** es trivial y se realiza en tiempo polinomial.

Bibliography

[1] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 1st ed. Pearson, 2005. Extraído de página 600.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1st ed. W. H. Freeman and Company, 1979.Extraído de página 238.

[3] Donguk Rhee. *Esquemas de Aproximación Polinomial Más Rápidos para Problemas de la Mochila*. Tesis de Maestría en Investigación de Operaciones, Instituto Tecnológico de Massachusetts, 2016.