

Universidad del Valle De Guatemala

Facultad de Ingeniería

Análisis y Diseño de Algoritmos



Proyecto 2 - Weighted Job Scheduling

Javier Mombiela 20067

Pablo González 20362

Jose Hernández 20053

Guatemala, 30 de mayo 2023

Definición y enunciado del problema

Enunciado

El problema que elegimos como grupo fue el “Weighted Job Scheduling” o el problema de programación de trabajos ponderados es un problema de optimización combinatoria que busca encontrar la secuencia óptima de ejecución de trabajos, considerando su duración y su valor o peso asociado.

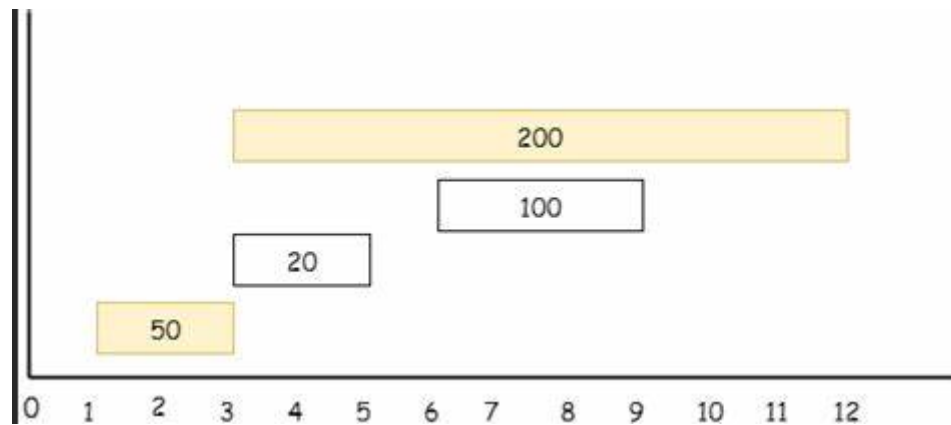
En el weighted job scheduling se tiene un conjunto de trabajos, en donde cada trabajo tiene un tiempo de inicio, un tiempo de finalización y un valor asociado. Entonces, el objetivo es seleccionar una secuencia de trabajos que maximice la suma de los valores asociados, siempre y cuando los trabajos no se traslapen. Que dos trabajos se traslapen significa que el tiempo de finalización del segundo trabajo esta entre el tiempo de inicio y el tiempo de finalización del primer trabajo. Si esto ocurre, no se pueden sumar los valores de estos dos trabajos.

Como resolverlo

Para poder resolver el problema, debemos de seguir una serie de pasos:

1. **Ordenar los trabajos por tiempo de finalización:** Primero, se deben de ordenar los trabajos por su tiempo de finalización. De este modo, se puede utilizar la técnica de programación dinámica para resolver el problema.
2. **Encontrar el último trabajo que no se traslape:** Para cada trabajo, se necesita encontrar el último trabajo que no se traslape con el trabajo actual. Esto se puede hacer utilizando la búsqueda binaria.
3. **Uso de programación dinámica:** Finalmente, podemos utilizar la programación dinámica para resolver este problema. Para hacer esto podemos definir un arreglo V , tal que $V[i]$ es el valor máximo obtenido considerando los trabajos disponibles que no se traslapen y cuyo valor sumado sea el máximo.

Ejemplo



Para poder resolver el weighted scheduling problem podemos empezar identificando las siguientes variables:

- Número de trabajos $n = 4$
- Propiedades de los trabajos = {Tiempo de inicio, tiempo de finalización, valor}
 - Trabajo 1: {1, 3, 50}
 - Trabajo 2: {3, 5, 20}
 - Trabajo 3: {6, 9, 100}
 - Trabajo 4: {3, 12, 200}

Ya que se definen las variables del problema, se puede proceder a resolverlo. Para poder encontrar la secuencia con el valor máximo, debemos de buscar las combinaciones posibles existentes. El primer paso es ordenar los trabajos por su tiempo de finalización y en este caso ya están sorteados del modo correcto. Como se mencionó anteriormente, las combinaciones no pueden tener dos trabajos traslapados. Por lo tanto, tomando el segundo paso de como resolver el problema, podemos observar que solo existen dos combinaciones válidas, las cuales son:

- Ejecutar el trabajo 1, luego el trabajo 2 y por último el trabajo 3, ya que el tiempo de finalización de cada trabajo no este entre los tiempos iniciales y finales del trabajo anterior. Esta combinación nos da un valor total de 170.
- La otra combinación posible sería la de ejecutar el trabajo 1 y luego el trabajo 4 porque el trabajo 1 termina en 3 y el trabajo 4 empieza en 3. El valor total de esta combinación sería de 250.
- Todas las otras combinaciones serían inválidas, ya que en estas si ocurriría el caso de que dos o más trabajos se traslaparían.

Una vez tenemos definidas nuestras posibles combinaciones, el paso final es sencillo, ya que solo debemos de comparar los valores totales de cada combinación y ver cuál es el máximo. En este caso tenemos que 250 es un valor mayor al de 170, por lo que podemos concluir que el valor máximo es 250 y este valor resulta de ejecutar el trabajo 1 y luego el 4.

Análisis del problema

Demostración de subestructura óptima

Para poder demostrar que el weighted job scheduling problem presenta una subestructura óptima, se deben identificar las decisiones, los subproblemas y finalmente, se debe demostrar la necesaria optimalidad. Por lo tanto, para poder demostrar la subestructura óptima de este problema, tenemos lo siguiente:

1. **Identificar la decisión:** Para este problema, la decisión que debemos tomar es si incluimos o no un trabajo particular en la solución óptima. Esto significa que para cada trabajo tenemos solo dos opciones: incluirlo en la solución (en caso de que este trabajo no se trasape con el trabajo anterior) o excluirlo y analizar el siguiente trabajo.
2. **Identificar subproblemas:** Dada la decisión anterior, los subproblemas resultantes son problemas más pequeños de programación de trabajos. Supongamos que la decisión que tomamos fue la de incluir un trabajo a la solución óptima, esto produce subproblemas para cada trabajo que no se traslapa con el trabajo incluido anteriormente. Estos subproblemas son más pequeños que el original y tiene las mismas características. Ahora supongamos que la decisión fue no incluir el trabajo en la solución, esto produce subproblemas para los trabajos restantes.
3. **Demostrar la necesaria optimalidad:** Supongamos que tenemos una solución al problema, pero que esta no es la óptima. Esto significa que existe al menos un subproblema que no se ha resuelto de manera óptima. Ahora, si reemplazamos la solución subóptima de este subproblema con su solución óptima, aumentaremos el valor de la solución global, lo que contradice el hecho de que nuestra solución original era la óptima.

Relación de recurrencia

Para poder construir la relación de recurrencia que calcula la solución óptima en el weighted scheduling problem, consideremos lo siguiente:

Sea $W[i]$ el valor máximo que se puede obtener al considerar los primeros i trabajos en la secuencia, ya ordenada, donde i varía de 1 a n . La relación de recurrencia que describe la solución óptima se define entonces de la siguiente manera:

$$V[i] = \max(valor[i] + V[j], V[i - j])$$

La función \max nos permite seleccionar el valor máximo entre incluir el trabajo i en la solución óptima o excluirlo en donde:

- $valor[i] + V[j]$ representa el caso en el que se incluye el trabajo i en la solución óptima.
 - $valor[i]$ es el valor asociado al trabajo i .
 - j es el índice del último trabajo que no se traslapa con el trabajo i . Si no existe, $j = 0$.
 - $V[j]$ es el valor máximo que se puede obtener de los trabajos que no se traslapa con el trabajo i .
- $V[i-j]$ representa el caso en que el trabajo i no se incluye en la solución óptima. En este caso, simplemente pasamos al siguiente trabajo y el máximo valor que se puede obtener es $V[i-1]$.

La relación de recurrencia permite calcular de manera eficiente el valor máximo que se puede obtener para cada subconjunto de trabajos, y la solución al problema será $V[n]$ donde n es el número total de trabajos.

Para los trabajos que no tienen trabajos anteriores que no se traslapan (en otras palabras, j no existe) se puede definir la siguiente relación:


$$V[i] = \max(val[i], V[i - j])$$

Y para el caso base, en donde ningún trabajo se ha seleccionado, podemos definir la siguiente relación:

$$V[0] = 0$$


Algoritmos de solución

Pseudocódigos



```
1  Función scheduleDP(arr, n)
2
3      Ordenar arr de acuerdo por tiempo de finalizacion
4
5      Inicializar table como un array de None de longitud n
6      Establecer table[0] como arr[0].weight
7
8      Para i en el rango de 1 hasta n
9          Establecer inclProf como arr[i].weight
10         Establecer l como el resultado de latestNonConflict(arr, i)
11
12         Si l no es igual a -1
13             inclProf se incrementa por table[l]
14
15         Establecer table[i] como el máximo entre inclProf y table[i - 1]
16
17     Establecer result como table[n - 1]
18
19     Retornar result
```

Pseudocódigo solución programación dinámica



```
1  Función scheduleGreedy(job)
2      Ordenar job por j.finish
3
4      Establecer n como la longitud de job
5      Inicializar result como una lista vacía
6      Añadir job[0] a result
7
8      Establecer i como 0
9      Para j en rango(1, n)
10         Si job[j].start >= job[i].finish
11             Añadir job[j] a result
12             Establecer i como j
13     Retornar result
```

Pseudocódigo solución algoritmo greedy

Análisis de los algoritmos

Programación Dinámica: La complejidad de tiempo del algoritmo de programación dinámica es $O(n^2)$, donde n es el número de trabajos. Esto se debe a que para cada trabajo, puede haber hasta $n-1$ trabajos que necesitamos verificar para determinar la compatibilidad, lo que lleva a una complejidad de tiempo cuadrática. Sin embargo, esto puede mejorarse a $O(n \log n)$ si usamos una búsqueda binaria para encontrar el último trabajo compatible en lugar de verificar todos los trabajos previos.

Algoritmo Greedy: La complejidad de tiempo del algoritmo voraz es $O(n \log n)$, donde n es el número de trabajos. Esto se debe principalmente a la necesidad de ordenar los trabajos, que generalmente toma $O(n \log n)$ utilizando algoritmos de ordenación eficientes como QuickSort o MergeSort. Una vez ordenados los trabajos, el algoritmo recorre la lista de trabajos una vez, seleccionando trabajos compatibles, lo que toma $O(n)$. Por lo tanto, la complejidad de tiempo total es dominada por el paso de ordenación.

Discusión sobre presencia greedy choice property

El algoritmo greedy que se proporcionó anteriormente para el weighted job scheduling presenta la "greedy choice property" (propiedad de elección codiciosa). La propiedad de elección codiciosa se refiere a tomar una decisión localmente óptima en cada etapa del algoritmo en la esperanza de alcanzar una solución global óptima.

En el caso del algoritmo greedy para el problema de programación de trabajos ponderados, la elección codiciosa se basa en seleccionar el trabajo con el tiempo de finalización más temprano que sea compatible con los trabajos previamente seleccionados. Esto se puede demostrar de la siguiente manera:

Primero, recordemos la propiedad de elección voraz (greedy choice property): Para este problema, dice que siempre podemos tomar el trabajo que termine antes (llamémoslo 'a') sin comprometer la optimalidad de la solución.

Vamos a demostrar por contradicción:

Supongamos que tenemos un conjunto de trabajos $J = \{1, 2, \dots, n\}$ ordenados por tiempo de finalización y que hay una solución óptima S para este conjunto de trabajos, tal que S no contiene el trabajo 'a' (el trabajo con el tiempo de finalización más temprano). Supongamos que 'b' es el primer trabajo en S .

Si 'b' no se superpone con 'a', podríamos añadir 'a' a S , obteniendo una solución S' con un peso total mayor, contradiciendo el hecho de que S es óptima.

Si 'b' se superpone con 'a', entonces no podemos incluir 'a' en S. Pero el peso de 'a' es al menos tan grande como el de 'b' (debido a que seleccionamos 'a' en base a su finalización temprana, no a su peso), por lo que podemos reemplazar 'b' con 'a' en S y obtener una solución S' con un peso total igual o mayor, contradiciendo también el hecho de que S es óptima.

Por lo tanto, nuestra suposición original de que la solución óptima S no contiene el trabajo 'a' debe ser incorrecta. Esto prueba que la propiedad de elección voraz se mantiene en este problema.

La elección voraz aquí es seleccionar el trabajo con el tiempo de finalización más temprano que no se superpone con los trabajos ya seleccionados, ya que esta elección siempre puede ser parte de una solución óptima. Esto nos permite construir una solución óptima incrementando un trabajo a la vez, de ahí la utilidad de los algoritmos voraces para este problema.

Análisis Empírico

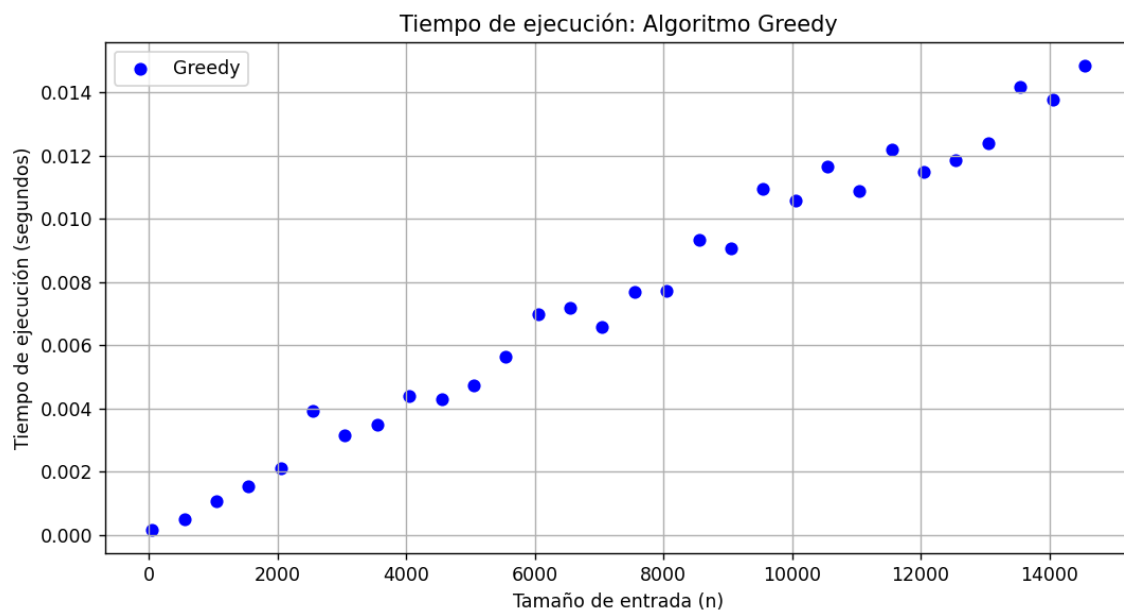
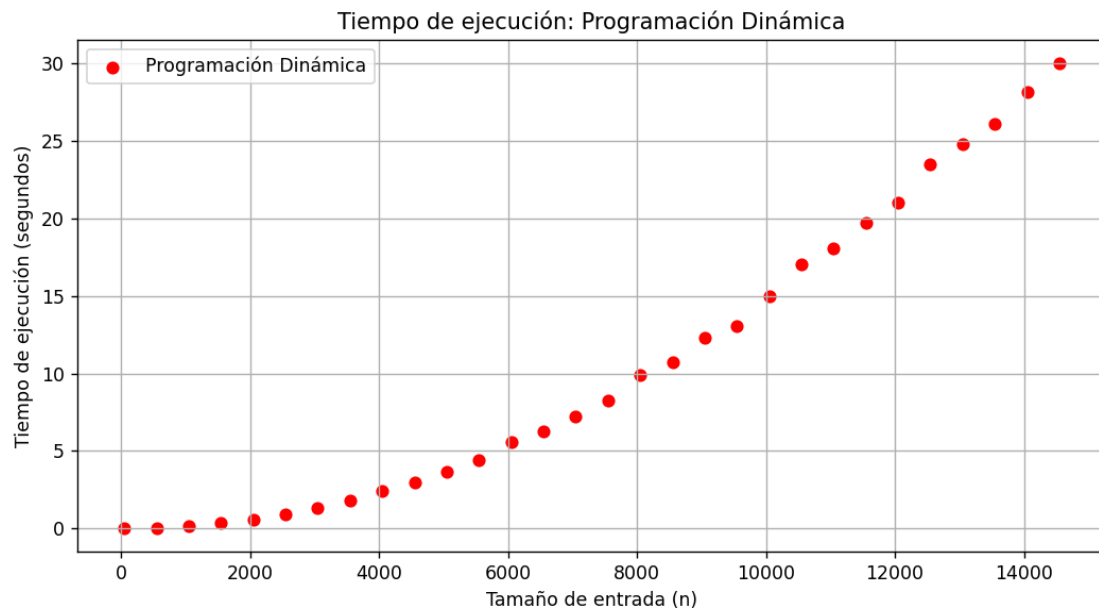
Listado de entradas de prueba

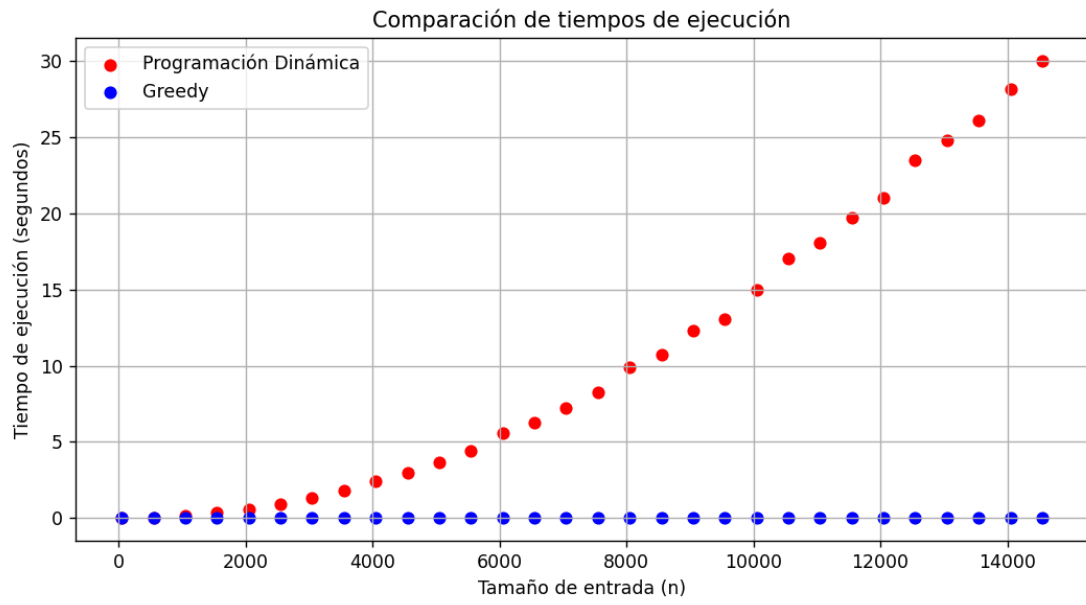
```
1 sizes = [50 + i*500 for i in range(30)]
2 dp_times = []
3 greedy_times = []
4
5 # realizando las pruebas
6 for n in sizes:
7     jobs = [Job(random.randint(1, 100), random.randint(101, 200), random.randint(1, 500)) for _ in range(n)]
```

Para el listado de entradas de prueba se decidió trabajar con un total de 30 diferentes tamaños, en donde la primera lista era de 50 elementos, y por cada siguiente lista que se generó, se le sumaban 500 datos más. Es decir, 50, 550, 1050... y así hasta llegar a $n = 14,500$.

Luego se realizaron pruebas utilizando cada tamaño de entrada de la lista sizes. Para cada tamaño, se generó una lista de trabajos con características aleatorias. Cada trabajo se define mediante tres parámetros: el tiempo de inicio, el tiempo de finalización y el peso. Estos valores se generaron aleatoriamente dentro de rangos predefinidos. El número de trabajos en cada lista se definió según el tamaño de entrada correspondiente.

Diagramas de dispersión





Regresión Polinomial

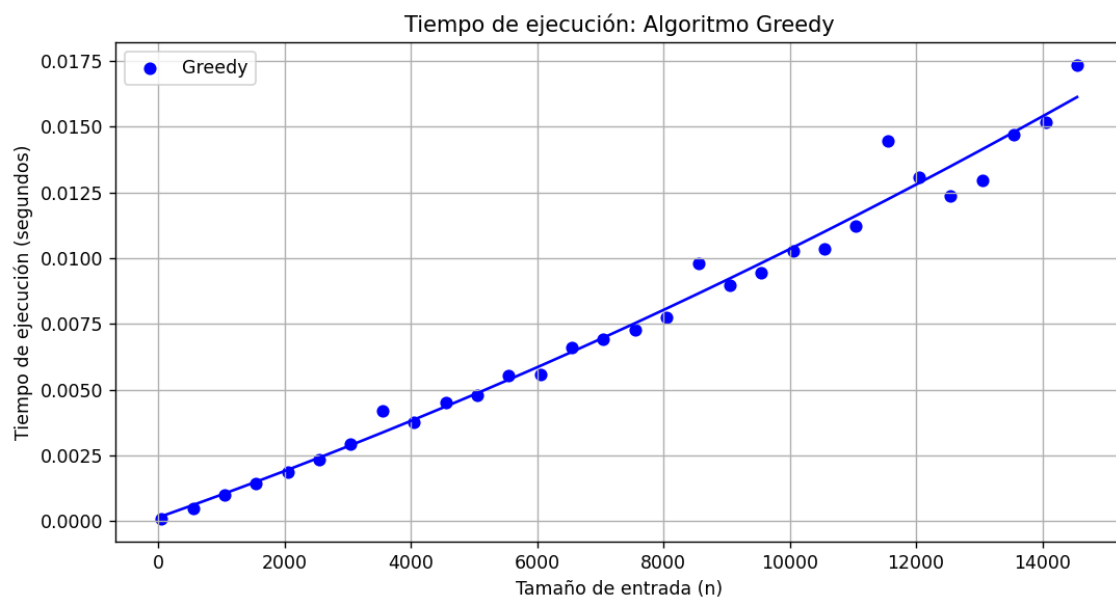
Función de regresión polinomial para Programación Dinámica:

$$1.36\text{e-}07 x^2 + 0.0001287 x - 0.2348$$



Función de regresión polinomial para Greedy:

$$-2.948\text{e-}11 x^2 + 1.569\text{e-}06 x - 0.001105$$



Discusión Final

Los algoritmos greedy, también conocidos como algoritmos voraces o algoritmos de avaricia, son una estrategia de resolución de problemas que intenta encontrar la solución óptima en cada paso con la esperanza de que estas soluciones locales óptimas conduzcan a una solución global óptima. Aunque este enfoque puede funcionar muy bien en algunos casos, no siempre garantiza la solución óptima global.

En el caso del problema de programación de trabajos ponderados (Weighted Job Scheduling Problem), un algoritmo greedy puede no proporcionar siempre la mejor solución. Este problema consiste en encontrar el conjunto de trabajos con la mayor suma de pesos (o beneficios) que no se superponen en el tiempo. Un algoritmo greedy podría abordar este problema seleccionando simplemente el trabajo disponible con el mayor peso en cada paso, pero esto podría dar lugar a soluciones subóptimas.

Por ejemplo, si se tiene un trabajo de peso 100 que dura 10 horas y dos trabajos de peso 60 que duran 5 horas cada uno, el algoritmo greedy elegiría el trabajo de peso 100, aunque la opción óptima sería seleccionar los dos trabajos de peso 60 para un total de 120.

Por otro lado, los algoritmos de programación dinámica, como el que se ha discutido anteriormente, pueden proporcionar la solución óptima a este problema. Estos algoritmos dividen el problema en subproblemas más pequeños y utilizan las soluciones de estos subproblemas para construir la solución al problema global. En el caso del problema de programación de trabajos ponderados, la programación dinámica puede considerar todas las combinaciones posibles de trabajos y, por lo tanto, es capaz de encontrar la solución óptima.

Sin embargo, la programación dinámica tiene un coste computacional más alto, lo que la hace menos eficiente que el algoritmo greedy para problemas de gran escala. En conclusión, si la optimización absoluta es necesaria, se debería utilizar la programación dinámica, pero si la eficiencia es más importante y se puede aceptar una solución aproximada, entonces el algoritmo greedy sería el camino a seguir.

- Según las funciones de regresión, el algoritmo greedy tiene un rendimiento considerablemente mejor en términos de tiempo de ejecución en comparación con el algoritmo de programación dinámica para el conjunto de datos probado. El coeficiente de x^2 en la regresión polinomial del algoritmo greedy es significativamente menor que el de la programación dinámica, lo que indica un crecimiento menos pronunciado del tiempo de ejecución con respecto al tamaño de entrada.
- Aunque el algoritmo greedy es más eficiente en términos de tiempo de ejecución, no siempre proporciona la solución óptima al problema de la programación de trabajos ponderados. Por otro lado, el algoritmo de programación dinámica puede ofrecer soluciones óptimas, pero a expensas de un mayor tiempo de ejecución.

- La elección entre utilizar programación dinámica o un enfoque greedy depende de los detalles específicos del problema y los requisitos de rendimiento. Si es crítico obtener la solución óptima y se puede tolerar un mayor tiempo de cálculo, entonces la programación dinámica es preferible. Por otro lado, si el tiempo de ejecución es una preocupación y se puede tolerar una solución que no sea óptima, el enfoque greedy puede ser más adecuado.

Bibliografía

Autor Desconocido. (s.f.). Weighted Job Scheduling problem [4 solutions]. Recuperado de:
<https://iq.opengenus.org/weighted-job-scheduling/>

GeeksforGeeks. (s.f.). Weighted Job Scheduling. Recuperado de:
<https://www.geeksforgeeks.org/weighted-job-scheduling/>

Rip Tutorial. (s.f.). Weighted Job Scheduling Algorithm. Recuperado de:
<https://riptutorial.com/dynamic-programming/example/25784/weighted-job-scheduling-algorithm>

TutorialsPoint. (s.f.). Weighted Job Scheduling. Recuperado de:
<https://www.tutorialspoint.com/Weighted-Job-Scheduling>