Proyecto Final DAA

Nahomi Bouza Rodríguez

Problema "Greedy" - Adilbek and watering system

Adilbek tiene que regar su jardín. Lo va a hacer con la ayuda de un complejo sistema de riego: solo tiene que suministrarle agua, y los mecanismos harán todo el trabajo restante.

El sistema de riego consume un litro de agua por minuto (si no hay agua, no funciona). No puede contener más de C litros. Adilbek ya ha vertido C_0 litros de agua en el sistema. Él va a empezar a regar el jardín ahora mismo y lo va a regar por m minutos, y el sistema de riego debe contener al menos un litro de agua al comienzo del i-ésimo minuto (por cada i de 0 a $m-1$).

Ahora Adilbek se pregunta qué hará si el sistema de riego se queda sin agua. Él llamó a n de sus amigos y les preguntó si iban a traer un poco de agua. El i-ésimo amigo respondió que no puede traer más de a_i litros de agua; llegará al principio del minuto t_i y vertirá toda el agua que tiene en el sistema (si el sistema no puede contener tal cantidad de agua, el exceso de agua se vierte); y luego le pedirá a Adilbek que pague p_i dólares por cada litro de agua que ha traído. Puede suponer que si un amigo llega al comienzo del minuto t_i y el sistema se queda sin agua al principio del mismo minuto, el amigo vierte su agua lo suficientemente rápido para que el sistema no deje de funcionar.

Por supuesto, Adilbek no quiere pagar a sus amigos, pero tiene que regar el jardín. Así que tiene que decirles a sus amigos cuánta agua deben traer. Formalmente, Adilbek quiere elegir n enteros $k_1, k_2, \ldots, k_n$ de una manera que:

- si cada amigo i trae exactamente k_i litros de agua, entonces el sistema de riego funciona durante todo el tiempo necesario para regar el jardín.
- la suma $\sum_{i=0}^{n} k_i \cdot p_i$ es la mínima posible.

Ayuda a Adilbek a determinar la cantidad mínima que tiene que pagar a sus amigos o determina que Adilbek no puede regar el jardín por m minutos.

Entrada

La primera línea contiene un entero q $(1 \leq q \leq 5 \cdot 10^5)$ – el número de consultas.

La primera línea de cada consulta contiene cuatro números enteros n, m, C y C_0 $(0 \leq n \leq 5 \cdot 10^5, 2 \leq m \leq 10^9, 1 \leq C_0 \leq C \leq 10^9)$ — el número de amigos, el número de minutos de riego, la capacidad del sistema de riego y el número de litros vertidos por Adilbek.

Cada una de los siguientes n líneas contiene tres enteros t_i, a_i, p_i $(0 < t_i < m, 1 \leq a_i \leq C, 1 \leq b_i \leq 10^9)$ — el minuto de llegada del i-ésimo amigo, la cantidad máxima de agua que i-ésimo amigo puede traer y el costo de cada litro que trae el i-ésimo amigo.

Se garantiza que la suma de todos los n sobre todas las consultas no excede $5 \cdot 10^5$.

Salida

Para cada consulta, imprima un número entero: la cantidad mínima que Adilbek tiene que pagar a sus amigos, o -1 si Adilbek no puede regar el jardín por m minutos.

Reinterpretación del problema

El sistema está en operación durante m minutos, dispone de un contenedor cuya capacidad máxima es de C unidades y cada minuto, el sistema consume una unidad de ese contenedor.

En determinados instantes, representados por el conjunto $T = t_1, t_2, t_3, \ldots, t_n$, se pueden comprar unidades adicionales para reabastecer el contenedor. En esos momentos:

La cantidad máxima de unidades que puede adquirirse en el instante t_i está dada por $A = a_1, a_2, a_3, \ldots, a_n$.
Cada unidad tiene un costo asociado, expresado por $P = p_1, p_2, p_3, \ldots p_n$.

El objetivo es garantizar que el sistema continúe operando durante los m minutos sin que el contenedor se quede vacío en ningún momento, pero también sin exceder la capacidad máxima del mismo. Además, se busca minimizar el costo total de las unidades adquiridas.

En otras palabras, se requiere encontrar una cantidad de unidades compradas $B = b_1, b_2, b_3, \ldots, b_n$ tal que el costo total $\sum b_k \cdot p_k$ sea el más bajo posible, cumpliendo con las restricciones de capacidad y demanda.

Propuesta de solución

La solución propuesta consiste en simular el sistema, donde utilizaremos tantas unidades disponibles como sea posible en cada minuto, sin exceder el límite máximo en ningún momento. Nos aseguraremos de quedarnos siempre con las unidades más económicas. Las unidades solo se comprarán cuando vayan a ser utilizadas, y siempre optaremos por la más barata de las que tenemos disponibles.

Formalmente, manejaremos una lista de precios que representará el costo de cada unidad prestada en ese momento. Esta lista se actualizará minuto a minuto, asegurándonos de mantener en ella únicamente las unidades más económicas disponibles hasta ese instante. Cuando una unidad sea necesaria, extraeremos la más barata de la lista, la compraremos y la añadiremos a la solución.

Si en un momento determinado hay nuevas unidades disponibles a un precio específico, las adquiriremos tantas veces como sea posible, hasta llenar la lista. Si tras esto aún sobran unidades, compararemos sus precios con los de las unidades ya prestadas: si alguna de las prestadas es más cara, la reemplazaremos con la nueva unidad más barata. Esto asegura que siempre tengamos las unidades más baratas disponibles en cada momento.

Si en algún punto antes de finalizar la simulación la lista queda vacía, el problema será insoluble con los datos actuales y devolveremos un valor de -1. De lo contrario, el resultado será la suma de los precios de todas las unidades compradas, lo que representa el costo mínimo para mantener el sistema operativo durante el periodo de tiempo determinado.

Para optimizar el manejo de la lista, en lugar de una simple lista de tamaño fijo, utilizaremos una estructura más eficiente, como un diccionario $< precio, cantidad >$, que almacenará la cantidad de unidades prestadas a cada precio. Además, los precios se mantendrán ordenados, ya sea mediante el diccionario (si permite ordenación) o utilizando una estructura adicional, como un heap.

Demostración

Primero demostremos que si el algoritmo encuentra una solución es porque existe una.

Sea S_{greedy} la solución generada por el algoritmo greedy y $C(t)$ el conjunto de estados del contenedor (capacidad ocupada en el tiempo t) en cualquier momento de la ejecución. Vamos a demostrar que si existe una solución factible, el algoritmo greedy la encontrará.

Primero, si existe una solución factible, entonces hay una forma de distribuir la compra de unidades a lo largo de los tiempos $t_1, t_2, \ldots, t_n$ de tal manera que el contenedor nunca se queda vacío y nunca sobrepasa su capacidad máxima en ningún momento.

Ahora, consideremos la solución factible en la que el contenedor se mantiene lo más vacío posible en cada instante. Denotemos $C'(t)$ al conjunto de estados del contenedor en cualquier momento bajo esta solución óptima. Se cumple que $C_{\text{greedy}}(t) \geq C'(t)$, ya que, en cada oportunidad de comprar unidades, el algoritmo greedy tomará prestadas tantas unidades como pueda, sin exceder la capacidad.

El algoritmo greedy siempre intenta llenar el contenedor con las unidades disponibles, y en cada momento compra la unidad más barata que está disponible en el contenedor. Esto garantiza que S_{greedy} siempre tendrá al menos tantas unidades disponibles como la solución óptima S', y comprará como mínimo la misma cantidad de unidades que S'.

Puesto que hemos demostrado que $C_{\text{greedy}}(t) \geq C'(t)$ y que S es factible, también lo será S_{greedy}. El contenedor nunca se queda vacío.

Además, el contenedor no se desbordará, ya que, si el algoritmo greedy intenta tomar prestadas más unidades de las que caben en el contenedor, desechará las unidades excedentes hasta que se ajuste a su capacidad máxima. Por lo tanto, en cada momento, el contenedor contendrá como máximo k unidades prestadas, y como las compras se hacen a partir de estas, nunca se comprará un número de unidades que sobrepase la capacidad del contenedor.

En conclusión, si existe al menos una solución factible, el algoritmo greedy construirá una solución válida.

Demostremos ahora que si el algoritmo encuentra una solución, entonces existe una solución factible.

Si el algoritmo encuentra una solución, entonces en ningún momento se compran suficientes unidades como para desbordar el contenedor (el contenedor siempre contiene a lo sumo k unidades disponibles, y las unidades que se compran se toman de estas). Además, el contenedor nunca se queda vacío, ya que, en caso contrario, el algoritmo devolvería un valor que indica que no existe solución.

Por lo tanto, el algoritmo construye una solución que no desborda el contenedor y que en ningún momento lo deja vacío. Esto garantiza que la solución es factible.

En consecuencia, si el algoritmo encuentra una solución, podemos concluir que existe al menos una solución factible, y esa solución es la que ha construido el algoritmo.

Optimalidad

Tanto en la solución del algoritmo greedy como en cualquier solución óptima, se compran exactamente $max(m - C_0, 0)$ unidades.

Sea S_{greedy} la solución del algoritmo greedy. En S_{greedy} se compran exactamente $max(m - C_0, 0)$ unidades, ya que solo se adquieren unidades cuando se usan, y de todas las unidades disponibles en cada momento, se compra siempre la más barata. Como se utiliza una unidad en cada minuto, durante los primeros $max(m, C_0)$minutos se consumen únicamente las unidades iniciales, dado que son las más económicas (con un costo de 0). Posteriormente, se compran en cada uno de los minutos $max(m - C_0, 0)$ restantes, una en cada minuto. Por lo tanto, en total se adquieren $max(m - C_0, 0)$ unidades.

En S_{opt}, se deben comprar como máximo $max(m - C_0, 0)$ unidades, ya que es necesario utilizar una unidad en cada minuto. No se compran más de m unidades, ya que si se adquirieran al menos $max(m - C_0, 0) + 1$, en el minuto $max(m - C_0, 0)$ quedaría al menos una unidad sin usar en el contenedor de S_{opt}.

Supongamos que el último momento en que se compraron unidades en S_{opt} fue el minuto t. Esto implicaría que, en ese momento, el contenedor tendría más de una unidad, porque sobró una al final. En tal caso, podríamos construir una nueva solución a partir de S_{opt}, comprando una unidad menos en el minuto t. Esta nueva solución sería factible, a menos que el contenedor se vaciara antes del minuto $max(m - C_0, 0)$ debido a este cambio. Sin embargo, dado que en S_{opt} el contenedor tenía más de una unidad en ese momento, esto no ocurriría.

El costo de esta nueva solución sería menor que el de S_{opt}, lo cual es una contradicción, ya que S_{opt} es la solución de costo mínimo. Por lo tanto, la suposición de que en una solución óptima se compran más de m unidades es incorrecta.

En conclusión, como se compran al menos $max(m - C_0, 0)$ unidades y como mucho $max(m - C_0, 0)$ unidades, entonces en cualquier solución óptima se adquieren exactamente $max(m - C_0, 0)$ unidades.

Demostremos que la solución greedy es óptima.

Sea $B = \{b_1, b_2, b_3, \ldots, b_n\}$ la solución que da el algoritmo greedy, donde b_i es la cantidad de unidades que se compran del tipo i.

Sea S_G el conjunto de soluciones óptimas con el prefijo común más grande con B. Es decir, que si el prefijo común más grande que comparte B con algún óptimo tiene tamaño k, entonces S_G contiene todas las soluciones óptimas que comparten con B en ese prefijo de longitud k.

Si $B \in S_G$ entonces la solución del greedy es óptima, y queda demostrado.

Si $B \notin S_G$, sea i la primera posición en la que todas las soluciones óptimas en S_G difieren de B.

Sea $S = \{s_1, s_2, s_3, \ldots s_n\}$, de todas las soluciones óptimas en S_G, aquella que cumple que $|s_i - b_i|$ es mínima (la menor entre todos los elementos de S_G).

Sea $b_i < s_i$:

Si $b_i < s_i$, como $\sum_{k=0}^{n} b_k = \sum_{k=0}^{n} s_k = \max(m - C_0, 0)$, como fue desmostrado anteriormente y $\forall k, k \in (0, i)$ se cumple $gbk = s_k$ (i es la primera posición en la que S y B difieren), entonces necesariamente tiene que existir un j, $i < j$, tal que $b_j > s_j$.

Sea j el primero de estos, entonces $\forall k$, $k \in (i, j)$ se cumple que $b_k \leq s_k$.

Se cumple necesariamente que $p_i \leq p_j$:

Supongamos $p_i > p_j$. Entonces, podríamos retirar una unidad a s_i y dársela a s_j, en la distribución S. Esta nueva distribución S' sería factible, pues solo podría infactibilizarse si posponer la compra de una unidad desde el momento t_i hasta el momento t_j provocase que el contenedor se quedase vacío antes de alcanzar el minuto t_j, y esto es imposible dado que la solución greedy emplea al menos una unidad menos de i que el óptimo (pues $b_i < s_i$), emplea a lo sumo la misma cantidad de unidades en el intervalo (i, j) que este (pues $\forall k, k \in (i, j)$, $b_k \leq s_k$) y se cumple que el greedy es factible, o sea, emplear una unidad menos en i no provoca que se vacíe el contenedor antes.

El costo de esta distribución S' sería $\sum_{k=0}^{n} s_k \cdot p_k - p_i + p_j < \sum_{k=0}^{n} s_k \cdot p_k$, o sea sería una distribución de menor costo que la distribución que teníamos, dado que $p_i > p_j$.

Luego tendríamos una distribución factible y de menor costo que el óptimo S, lo cual es absurdo. Por tanto $p_i \leq p_j$ es cierto.

Se cumple también que $p_i \geq p_j$:

Supongamos $p_i < p_j$. Como las soluciones son iguales hasta este punto, entonces en el momento t_i ambos contenedores, el de la solución greedy y el de la solución óptima, están llenos hasta la misma capacidad.

Se cumple $\forall k, k \in (i, j), s_k > 0$, que $p_k \leq p_j$: Supongamos $p_k > p_j$ para alguno de ellos, entonces, podríamos quitarle una unidad a s_k y dársela a s_j, en la distribución S.

Es posible quitarle una unidad a s_k pues $s_k > 0$, y darle una unidad a s_j pues $s_j < b_j \leq a_j$. Esta nueva distribución S' sería factible, pues solo podría infactibilizarse si posponer la compra de una unidad desde el momento t_k hasta el momento t_j provocase que el contenedor se quedase vacío antes de alcanzar el minuto t_j, y esto es imposible dado que la solución greedy emplea al menos una unidad menos de i que el óptimo (pues $b_i < s_i$), emplea a lo sumo la misma cantidad de unidades en el intervalo (i, j) que este (pues $\forall k, k \in (i, j), b_k \leq s_k$) y se cumple que el greedy es factible, o sea, emplear una unidad menos en i no provoca que se vacíe el contenedor antes, por tanto, tampoco lo hará tomar una menos en k.

El costo de esta distribución S' sería $\sum_{l=0}^{n} s_l \cdot p_l - p_k + p_j < \sum_{l=0}^{n} s_l \cdot p_l$, o sea sería una distribución de menor costo que la distribución que teníamos, dado que $p_k > p_j$.

Luego tendríamos una distribución factible y de menor costo que el óptimo S, lo cual es absurdo. Por tanto $p_k \leq p_j$ es cierto, $\forall k, k \in (i, j), s_k > 0$. Por tanto $\forall k, k \in (i, j)$ tal que $p_k > p_j$ se cumple necesariamente que $s_k = 0$.

Como la solución S tuvo capacidad para comprar cada una de las unidades s_k en su momento, sin desbordamiento, entonces el greedy tiene capacidad para "tomar prestadas" tantas unidades como compra S en cada uno de estos momentos.

Como para todas las unidades que S compra en $[i, j)$, se cumple que son unidades más baratas que la unidad disponible en t_j, entonces el greedy toma prestadas, en este intervalo, tantas unidades más baratas que p_j como unidades compra S en él.

Pero como $b_i < s_i$ y $\forall k, k \in (i, j), b_k \leq s_k$, entonces S compra en este intervalo al menos una unidad más que G.

En consecuencia, G toma prestadas tantas unidades más baratas que p_j en este intervalo como unidades compra más uno, al menos.

O sea, G toma prestada al menos una unidad más de estas que las que compra de este intervalo.

El algoritmo óptimo llega al momento t_j con al menos una unidad en el contenedor, dado que el algoritmo greedy compra al menos una unidad menos que S en el intervalo $[i, j)$, y si S llega a t_j con el contenedor vacío entonces G no llega, lo cual es contradictorio pues la solución del greedy es factible.

Como el algoritmo óptimo llega a t_j con al menos una unidad, entonces el algoritmo greedy llega a t_j con al menos una unidad prestada.

Por tanto, el greedy llega al momento t_j con al menos una unidad prestada más barata que p_j, pues en algún momento en el intervalo $[i, j)$ hubo una, y como esta no se compró, entonces o bien se mantuvo durante el resto del intervalo o bien fue sustituida por una unidad incluso más barata.

Como el greedy llega al momento t_j con al menos una unidad prestada más barata que p_j, entonces el greedy tuvo que haber comprado esta unidad para poder comprar cualquiera de las obtenibles en j:

Mientras esta unidad esté prestada en el contenedor, no podrán comprarse unidades de precio p_j , pues en cada iteración del greedy se compra la unidad más barata, y esta unidad es más barata que p_j. Esta unidad solo será sacada del contenedor si se compra o si es extraída del contenedor por alguna unidad más barata. Si es extraída por alguna unidad más barata, esta (o estas) habrán sacado antes todas las unidades de precio p_j, puesto que son más caras que dicha unidad y el algoritmo greedy sustituye las unidades más caras primero. Luego, para comprar unidades de las obtenibles en j, el algoritmo greedy tuvo necesariamente que comprar dicha unidad.

Pero el algoritmo greedy compra $b_j > s_j \geq 0$ unidades de las obtenibles en j, por lo que necesariamente compró dicha unidad. Esto contradice la hipótesis de que esa unidad no se compró, lo cual es absurdo. Luego la hipótesis es falsa, y no puede ser $p_i < p_j$.

Por tanto $p_i \geq p_j$ es cierto.

Tenemos entonces que $p_i \leq p_j$ y $p_i \geq p_j$, por lo que $p_i = p_j$.

Luego, construimos una nueva distribución S' a partir de S, disminuyendo s_i en $\min(s_i - b_i, b_j - s_j)$ y aumentando la s_j en la misma cantidad. Esta nueva distribución es factible porque:

$s_i - \min(s_i - b_i, b_j - s_j) \geq s_i - s_i + b_i = b_i \geq 0$, pues el greedy es factible.

$s_j + \min(s_i - b_i, b_j - s_j) \leq s_j + b_j - s_j = b_j \leq a_j$, pues el greedy es factible.

Dadas las dos primeras condiciones, esta nueva distribución S' solo podría infactibilizarse si posponer la compra de $\min(s_i - b_i, b_j - s_j)$ unidades desde el momento t_i hasta el momento t_j provocase que el contenedor se quedase vacío antes de alcanzar el minuto t_i, y esto es imposible dado que la solución greedy emplea al menos $s_i - b_i$ unidades menos de i (pues $b_i < s_i$), emplea a lo sumo la misma cantidad de unidades en el intervalo (i, j) que el óptimo (pues $\forall k, k \in (i, j), b_k \leq s_k$) y el greedy es factible, o sea, emplear $\min(s_i - b_i, b_j - s_j)$ unidades menos de i no provoca que se vacíe el contenedor antes.

El costo de S' es $\sum s_k \cdot p_k - \min(s_i - b_i, b_j - s_j) \cdot p_i + \min(s_i - b_i, b_j - s_j) \cdot p_j$. Pero como $p_i = p_j$ entonces esto es lo mismo que $\sum s_k \cdot p_k - \min(s_i - b_i, b_j - s_j) \cdot p_i + \min(s_i - b_i, b_j - s_j) \cdot p_i = \sum s_k \cdot p_k$, o sea, tiene el mismo costo que S'. Luego S' tiene el mismo costo que S y es también factible, por lo que S' es también óptimo.

Si se cumple que $s_i - b_i = \min(s_i - b_i, b_j - s_j)$, entonces ahora en S' se cumple que $\forall k, k \in (0, i)$, $s'_k = s_k = b_k$ y además $s'_i = s_i - (s_i - b_i) = b_i$. Luego S' cumple que $\forall k, k \in (0, i+1)$, $s'_k = b_k$, por lo que tiene un prefijo común con G más grande que el de S, que es el óptimo con el prefijo común más grande con G, lo cual es contradictorio.

Si se cumple que $b_j - s_j = \min(s_i - b_i, b_j - s_j)$, entonces ahora en S' se cumple que $\forall k, k \in (0, i)$, $s'_k = s_k = b_k$, por lo que $S' \in S_G$. Pero $s_i - b_i > s_i - \min(s_i - b_i, b_j - s_j) - b_i \geq 0$, por lo que se cumple $s_i - b_i > s'_i - b_i \geq 0$, y por tanto $|s_i - b_i| > |s'_i - b_i|$. Luego $S' \in S_G$ al igual que S y cumple que $|s'_i - b_i| < |s_i - b_i|$, cuando S era el de menor diferencia absoluta $|s_i - b_i|$ de entre todos los elementos de S_G, lo cual es contradictorio.

Como en ambos casos llegamos a una contradicción, entonces no puede ser $b_i < s_i$.

Sea $b_i > s_i$:

Si $b_i > s_i$, como $\sum_{k=0}^{n} b_k = \sum_{k=0}^{n} s_k = \max(m - C_0, 0)$ como está y $\forall k, k \in (0, i)$ se cumple $b_k = s_k$ (i es la primera posición en la que S y G difieren), entonces necesariamente tiene que existir un j, $i < j$, tal que $b_j < s_j$.

Sea j el primero de estos, entonces $\forall k, k \in (i, j)$ se cumple que $b_k \geq s_k$. Se cumple necesariamente que $p_i \geq p_j$:

Supongamos $p_i < p_j$, entonces, podríamos quitarle una unidad a s_j y dársela a s_i, en la distribución S. Es posible quitarle una unidad a s_j pues $s_j > b_j \geq 0$ y darle una unidad a s_i pues $s_i < b_i \leq a_i$. Esta nueva distribución S' sería factible pues solo podría infactibilizarse si adelantar la compra de una unidad desde el momento t_j al momento t_i provocase que el contenedor se desbordase antes de alcanzar el minuto t_j, y esto es imposible dado que la solución greedy emplea al menos una unidad más de i (pues $b_i > s_i$), emplea como mínimo la misma cantidad de unidades en el intervalo (i, j) que el óptimo (pues $\forall k, k \in (i, j), b_k \geq s_k$) y el greedy es factible, o sea, emplear una unidad más en i no provoca que el contenedor se desborde antes.

El costo de esta distribución S' sería $\sum_{k=0}^{n} s_k \cdot p_k + p_i - p_j < \sum_{k=0}^{n} s_k \cdot p_k$, o sea sería una distribución de menor costo que la distribución que teníamos, dado que $p_i < p_j$. Luego tendríamos una distribución factible y de menor costo que el óptimo S, lo cual es absurdo. Por tanto $p_i \geq p_j$ es cierto.

Se cumple también que $p_i \leq p_j$:

Supongamos $p_i > p_j$. Como las soluciones son iguales hasta este punto, entonces en el momento t_i ambos contenedores, el de la solución greedy y el de la solución óptima, están llenos hasta la misma capacidad.

Se cumple $\forall k, k \in (i, j), s_k > 0$, que $p_k \leq p_i$:

Supongamos $p_k > p_i$ para alguno de ellos, entonces, podríamos quitarle una unidad a s_k y dársela a s_i, en la distribución S. Es posible quitarle una unidad a s_k pues $s_k > 0$, y darle una unidad a s_i pues $s_i < b_i \leq a_i$.

Esta nueva distribución S' sería factible pues solo podría infactibilizarse si adelantar la compra de una unidad desde el momento t_k al momento t_i provocase que el contenedor se desbordase antes de alcanzar el minuto t_k, y esto es imposible dado que la solución greedy emplea al menos una unidad más de i que el óptimo (pues $b_i > s_i$), emplea como mínimo la misma cantidad de unidades en el intervalo (i, j) que este (pues $\forall k, k \in (i, j), b_k \geq s_k$) y el greedy es factible, o sea, emplear una unidad más en i no provoca que el contenedor se desborde antes de llegar a t_j, y por tanto, tampoco antes de llegar a t_k.

El costo de esta distribución S' sería $\sum_{l=0}^{n} s_l \cdot p_l - p_k + p_i < \sum_{l=0}^{n} s_l \cdot p_l$, o sea sería una distribución de menor costo que la distribución que teníamos, dado que $p_k > p_i$. Luego tendríamos una distribución factible y de menor costo que el óptimo S, lo cual es absurdo. Por tanto $p_k \leq p_i$ es cierto, $\forall k, k \in (i, j), s_k > 0$.

Por tanto $\forall k, k \in (i, j)$ tal que $p_k > p_i$ se cumple necesariamente que $s_k = 0$.

Como la solución S tuvo capacidad para comprar cada una de las unidades s_k en su momento, sin desbordamiento, entonces el greedy tiene capacidad para "tomar prestadas" tantas unidades como compra S en cada uno de estos momentos.

Como para todas las unidades que S compra en $[i, j)$, se cumple que son unidades más baratas que la unidad disponible en t_i, entonces el greedy toma prestadas, en este intervalo, tantas unidades más baratas que p_i como unidades compra S en él.

Pero como $b_i > s_i$ y $\forall k, k \in (i, j), b_k \geq s_k$, entonces S compra en este intervalo a lo sumo una unidad menos que G.

En consecuencia, G toma prestadas tantas unidades más baratas que p_i en este intervalo como unidades compra menos uno, a lo sumo.

O sea, G compra al menos una unidad más cara que p_i, de las obtenibles en este intervalo.

El algoritmo greedy llega al momento t_j con al menos una unidad más cara que p_i, la cual debe comprar por lo antes expuesto. Esto se cumple pues, para vender todas las unidades más caras que p_i es necesario vender primero todas las más baratas en este intervalo, que son tantas unidades como unidades compra S en él.

En este caso G consumiría en este intervalo más unidades que las que S compra en él, dado que emplea tantas unidades más baratas que p_i como unidades compra S y emplea todas las más caras que p_i, que son al menos una más pues $b_i > s_i$.

Luego, la solución S no compraría unidades suficientes para llegar al momento t_j, lo cual es contradictorio.

Como el greedy llega al momento t_j con al menos una unidad prestada más cara que p_i, y debe comprarla, entonces el greedy tuvo que haber comprado al menos todas las unidades obtenibles en j para poder comprar esta unidad:

De las unidades obtenibles en j se toman todas prestadas, pues si no hay capacidad para ello, entonces la unidad más cara que p_i sale del contenedor, pues esta unidad es más cara que $p_j \leq p_i$.

Mientras estas unidades de j estén prestada en el contenedor, no podrán comprarse unidades de precio superior a p_i, pues en cada iteración del greedy se compra la unidad más barata, y esta unidad es más cara que $p_j \leq p_i$.

Estas unidades solo serán sacada del contenedor si se compran o si son extraídas del contenedor por alguna unidad más barata.

Si son extraídas por unidades más barata, estas habrán sacado antes la unidad de precio superior a p_i, puesto que son más caras que dichas unidades y el algoritmo greedy sustituye las unidades más caras primero.

Luego, para comprar dicha unidad, entonces el algoritmo greedy tuvo necesariamente que comprar todas las unidades obtenibles en j.

Pero como el algoritmo greedy compra esa unidad, entonces compró todas las unidades obtenibles en j. Entonces se compraron $b_j = a_j > s_j$ de estas unidades, pero teníamos $b_j < s_j$, lo cual es absurdo. Luego la hipótesis es falsa, y no puede ser $p_i < p_j$. Por tanto $p_i \geq p_j$ es cierto.

Tenemos entonces que $p_i \leq p_j$ y $p_i \geq p_j$, por lo que $p_i = p_j$.

Luego, construimos una nueva distribución S' a partir de S, aumentando s_i en $\min(b_i - s_i, s_j - b_j)$ y disminuyendo s_j en la misma cantidad. Esta nueva distribución es factible porque:

$s_i + \min(b_i - s_i, s_j - b_j) \leq s_i + b_i - s_i = b_i \leq a_i$, pues el greedy es factible.

$s_j - \min(b_i - s_i, s_j - b_j) \geq s_j - s_j + bb_j = b_j \geq 0$, pues el greedy es factible.

Dadas las dos primeras condiciones, esta nueva distribución S' solo podría infactibilizarse si adelantar la compra de $\min(s_i - b_i, b_j - s_j)$ unidades desde el momento t_j hasta el momento t_i provocase que el contenedor se desbordase antes de alcanzar el minuto t_j, y esto es imposible dado que la solución greedy emplea al menos $b_i - s_i$ unidades más de i (pues $b_i < s_i$), emplea como mínimo la misma cantidad de unidades en el intervalo (i, j) que el óptimo (pues $\forall k$, $k \in (i, j)$, $b_k \geq s_k$) y el greedy es factible, o sea, emplear $\min(s_i - b_i, b_j - s_j)$ unidades más de i no provoca no provoca que el contenedor se desborde antes.

El costo de S' es $\sum s_k \cdot p_k + \min(b_i - s_i, s_j - b_j) \cdot p_i + \min(b_i - s_i, s_j - b_j) \cdot p_j$. Pero como $p_i = p_j$ entonces esto es lo mismo que $\sum s_k \cdot p_k - \min(b_i - s_i, s_j - b_j) \cdot p_i + \min(b_i - s_i, s_j - b_j) \cdot p_i = \sum s_k \cdot p_k$, o sea, tiene el mismo costo que S'. Luego S' tiene el mismo costo que S y es también factible, por lo que S' es también óptimo.

Si se cumple que $b_i - s_i = \min(b_i - s_i, s_j - b_j)$, entonces ahora en S' se cumple que $\forall k, k \in (0, i)$, $s'_k = s_k = b_k$ y además $s'_i = s_i + (b_i - s_i) = b_i$. Luego S' cumple que $\forall k, k \in (0, i+1)$, $s'_k = b_k$, por lo que tiene un prefijo común con G más grande que el de S, que es el óptimo con el prefijo común más grande con G, lo cual es contradictorio.

Si se cumple que $s_j - b_j = \min(b_i - s_i, s_j - b_j)$, entonces ahora en S' se cumple que $\forall k, k \in (0, i)$, $s'_k = s_k = b_k$, por lo que $S' \in S_G$. Pero $b_i - s_i > b_i - (s_i + \min(s_i - b_i, b_j - s_j)) \geq 0$, por lo que se cumple $b_i - s_i > b_i - s'_i \geq 0$, y por tanto $|s_i - b_i| > |s'_i - b_i|$. Luego $S' \in S_G$ al igual que S y cumple que $|s'_i - b_i| < |s_i - b_i|$, cuando S era el de menor diferencia absoluta $|s_i - b_i|$ de entre todos los elementos de S_G, lo cual es contradictorio.

Como en ambos casos llegamos a una contradicción, entonces no puede ser $b_i > s_i$.

Luego, necesariamente tiene que ser $s_i = b_i$ en S, pero i es la primera posición en la que todas las soluciones óptimas en S_G difieren de B, llegando nuevamente a una contradicción. Entonces no puede ser $B \notin S_G$ y por tanto B es óptimo.

Complejidad temporal

El algoritmo realiza m iteraciones. En cada una de estas se extrae la unidad de menor precio de la estructura, utilizando una estructura auxiliar que permite realizar esta operación en $O(logP)$, donde P representa la cantidad de precios distintos presentes en la estructura. No obstante, hay a lo sumo n precios diferentes, podemos concluir que $O(logP)$ es $O(logn)$. Por lo tanto, el tiempo total de estas operaciones es $O(m \cdot logn)$.

Aunque este número de operaciones puede ser elevado, es posible optimizarlo mediante un enfoque más eficiente. Observamos que no es necesario realizar las m iteraciones de manera explícita. Para cada $$ en el rango $[0,n)$, entre los momentos t_i y $t_{i + 1}$ el contenedor no se actualiza, excepto por la extracción de las unidades más baratas que hayan salido del contenedor desde el tiempo t_i . Así, podemos realizar iteraciones únicamente en los puntos t_i y extraer en ese momento las $t_{i} - t_{i - 1}$ unidades más económicas. Esto reduce el número de iteraciones a n.

Además, al agrupar las unidades por $< precio, frecuencia >$, en cada iteración podemos revisar los precios de menor a mayor, extrayendo total o parcialmente las unidades de cada precio hasta alcanzar $t_i - t_{i-1}$. Por lo tanto, podemos clasificar las extracciones de precios en dos tipos, totales y parciales.

En las extracciones totales, se eliminan todas las unidades de un precio determinado. Dado que hay a lo sumo n precios diferentes, el número de extracciones totales es $O(n)$. Cada extracción se realiza en $O(logn)$, lo que lleva a un costo total de $O(nlogn)$.

En las extracciones parciales, se eliminan solo una parte de las unidades de un precio específico. Estas extracciones son a lo sumo n, ya que se puede realizar una extracción parcial por cada iteración. Al igual que antes, el costo es $O(nlogn)$.

En consecuencia, el costo total de estas extracciones es $O(nlogn) + O(nlogn) = O(nlogn)$.

Adicionalmente, en cualquier momento, pueden llegar nuevas unidades disponibles, lo que permite sustituir hasta C unidades en el contenedor por C de estas nuevas unidades. En el peor de los casos, esta sustitución podría requerir $O(ClogC)$ si se realiza una por una. Sin embargo, al agrupar las unidades por $< precio, frecuencia >$, podemos revisar los precios disponibles en orden decreciente y sustituir total o parcialmente las unidades hasta completar p sustituciones.

Nuevamente, clasificamos estas operaciones en dos tipos, totales y parciales.

En las extracciones totales, al igual que antes, el número máximo es n, con un costo de $O(nlogn)$.

En las extracciones parciales, también son a lo sumo n, con un costo equivalente de $O(nlogn)$.

Por lo tanto, el costo total de las sustituciones es $O(nlogn) + O(nlogn) = O(nlogn)$.

Finalmente, el costo total del algoritmo se resume como $O(nlogn) + O(nlogn) = O(nlogn)$.

Pruebas

Además, se llevaron a cabo pruebas del algoritmo en la plataforma Codeforces. Inicialmente, se realizó un submit utilizando Python; sin embargo, esta versión no logró ejecutarse dentro del tiempo límite debido a la velocidad del lenguaje. Para mejorar el rendimiento, se tradujo el algoritmo a C++, lo que resultó en un submit exitoso que cumplió con las restricciones de tiempo.

Submit C++

Submit Python

"Binary Search" - Duff in Mafia

Duff es una de las jefas de la Mafia en su país, Andarz Gu. Andarz Gu tiene n ciudades (numeradas del 1 al n) conectadas por m carreteras bidireccionales (numeradas del 1 al m).

Cada carretera tiene un tiempo de destrucción y un color. La i-ésima carretera conecta las ciudades v_i y u_i, su color es c_i y su tiempo de destrucción es t_i.

La Mafia quiere destruir un emparejamiento en Andarz Gu. Un emparejamiento es un subconjunto de carreteras tal que ninguna de las dos carreteras en este subconjunto tiene un extremo en común. Pueden destruir estas carreteras en paralelo, es decir, el tiempo total de destrucción es el máximo de los tiempos de destrucción de todas las carreteras seleccionadas.

Quieren que se cumplan dos condiciones:

1. Las carreteras restantes forman una coloración propia.
2. El tiempo de destrucción de este emparejamiento se minimiza.

Las carreteras restantes después de destruir este emparejamiento forman una coloración propia si y solo si no hay dos carreteras del mismo color que tengan el mismo extremo; en otras palabras, los bordes de cada color deben formar un emparejamiento.

No hay programador en la Mafia. Por eso, Duff te pidió ayuda. Por favor, ayúdala a determinar qué emparejamiento destruir para satisfacer esas condiciones (o indicar que esto no es posible).

Entrada

La primera línea de entrada contiene dos enteros n y m ($2 \leq n \leq 5 \times 10^4$ y $1 \leq m \leq 5 \times 10^4$), el número de ciudades y el número de carreteras en el país.

Las siguientes m líneas contienen las carreteras. La i-ésima de ellas contiene cuatro enteros v_i, u_i, c_i y t_i ($1 \leq v_i, u_i \leq n$, $v_i \neq u_i$ y $1 \leq c_i, t_i \leq 10^9$ para cada $1 \leq i \leq m$).

Salida

En la primera línea de salida, imprime "Sí" (sin comillas) si es posible satisfacer la primera condición y "No" (sin comillas) en caso contrario.

Si es posible, entonces debes imprimir dos enteros t y k en la segunda línea, el tiempo mínimo de destrucción y el número de carreteras en el emparejamiento.

En la tercera línea, imprime k enteros distintos separados por espacios, los índices de las carreteras en el emparejamiento en cualquier orden. Las carreteras están numeradas desde uno en orden de su aparición en la entrada.

Si hay más de una solución, imprime cualquiera de ellas.

Reinterpretación del problema

Consideremos un grafo no dirigido G que consta de n nodos y m aristas. Cada arista e_i posee 2 valores c_i y t_i tal que c_i es el color correspondiente a la arista y t_i el tiempo que toma eliminarla. El objetivo es eliminar aristas de G en el menor tiempo posible en el menor tiempo posible, cumpliendo las siguientes restricciones:

- No se pueden eliminar dos aristas que compartan un nodo.
- No pueden quedar dos aristas del mismo color que compartan un nodo.
- El tiempo total para eliminar un conjunto E de aristas se define como $\max_{e_i \in E} t_i$

Si no es posible eliminar un conjunto de aristas que cumpla con estas condiciones, se debe imprimir "No". Si existe una solución, se debe imprimir "Sí", seguido del tiempo requerido y la cantidad de aristas a eliminar. Finalmente, se debe mostrar el conjunto de aristas a eliminar.

Propuesta de solución

Dado un conjunto de aristas E en el grafo, se pueden ordenar estas aristas según el tiempo t_i que toma eliminarlas, en orden no decreciente. Este orden nos permitirá analizar el tiempo de eliminación de manera eficiente.

Una vez que hemos ordenado las aristas, podemos aplicar un algoritmo de búsqueda binaria sobre este conjunto ordenado para determinar el tiempo mínimo necesario para eliminar un conjunto válido de aristas, teniendo en cuenta que **es posible eliminar aristas del prefijo de longitud i de forma correcta para obtener un grafo válido.**

Demostración

Si encontramos una solución s_i que permite eliminar aristas del prefijo p_i de longitud i podemos afirmar que esta solución también será válida para cualquier $j > i$. Esto se debe a que todas las aristas eliminadas pertenecen al conjunto p_i y y por lo tanto, también están incluidas en el conjunto p_j.

Luego al aplicar que es posible eliminar aristas del prefijo de longitud i de forma correcta para obtener un grafo válido, sobre la secuencia $1, 2, 3, \ldots, n$ se obtiene la secuencia $0, 0, \ldots, 0, 1, 1, \ldots, 1, 1$ sobre la que se puede aplicar búsqueda binaria para encontrar el primer elemento donde el predicado se cumple.

si en algún momento se detecta que existen más de una pareja de aristas del mismo color conectadas a un mismo nodo, se puede concluir que no existe solución válida. Esto es porque, según las restricciones, solo se puede eliminar una arista que incida en ese nodo. Por lo tanto, no se podrá cumplir la condición de no dejar dos aristas del mismo color en ese nodo, haciendo imposible encontrar una solución viable.

Dado un prefijo p del conjunto de aristas E definimos una variable booleana para cada arista $e_i \in E$, donde será True si la arista e_i será eliminado del grafo G, False si no es eliminada. El objetivo es determinar si existe una asignación de True o False para cada arista, de modo que se cumplan las restricciones del problema y se obtenga una solución válida. Este problema se puede modelar utilizando la técnica de 2-SAT de la siguiente manera:

- si dos aristas e_i y e_j comparten un nodo y tienen el mismo color, necesitamos eliminar al menos una de ellas para evitar que dos aristas del mismo color incidan en un mismo nodo. Esto lleva a la clausula lógica $e_i \vee e_j$ ya que necesitamos eliminar uno de ellos.
- si existe un subconjunto de aristas $e_1, e_2, \ldots, \ldots e_k$ con un nodo en común entonces $\forall i < j \quad \neg e_i \vee \neg e_j$ ya que si decidimos eliminar una arista e_i entonces necesitamos que el resto no sean eliminadas.

Con estas dos reglas, hemos convertido el problema de eliminar aristas en una instancia de 2-SAT, donde las cláusulas lógicas describen las restricciones sobre qué aristas pueden ser eliminadas.

Resolución mediante 2-SAT

El problema de 2-SAT consiste en encontrar una asignación de valores True o False para un conjunto de literales $a_1, a_2, \ldots, a_n$, de manera que se satisfagan m clausulas de la forma $a_i \vee a_j$, entonces el 2-SAT retorna True si existe una asignación de True para cada literal a_i de forma tal que todas las clausulas sean verdaderas, False en caso contrario.

Este problema se puede modelar como un grafo dirigido con las siguientes características:

Por cada literal a_i, se crean dos nodos en el grafo: uno para el literal a_i y otro para su complemento $\neg a_i$

Por cada claúsula de la forma $a_i \vee a_j$, sse puede reescribir como dos implicaciones: $\neg a_i \implies a_j$ y $\neg a_j \implies a_i$. Esto se modela en el grafo añadiendo dos aristas: la primera que va del nodo a_i al nodo $\neg a_j$ y otra de a_j a $\neg a_i$.

Notese que en caso de existir un camino en el grafo de a_i a a_j dada la transitividad de la implicación lógica podemos decir que $a_i \implies a_j$

Una vez que se ha construido este grafo, el siguiente paso es identificar las componentes fuertemente conexas. Si un literal a_i y su complemento $\neg a_i$ se encuentran en la misma componente fuertemente conexa, significa que hay una contradicción. Esto se debe a que existe un camino desde a_i hasta $\neg a_i$ y de $\neg a_i$ a a_i. por lo tanto $a_i \implies \neg a_i$ y $\neg a_i \implies a_i$ lo cual es una contradicción. En este caso, no existe una asignación válida para los literales, y el 2-SAT devolverá False.

Una vez identificadas las componentes fuertemente conexas, se construye un grafo dirigido acíclico (DAG), donde cada nodo representa una componente fuertemente conexa. Entre dos nodos a y b del DAG, existirá una arista (a, b) si hay una arista entre algún nodo de la componente correspondiente a a y algún nodo de la componente correspondiente a b en el grafo original.

El nuevo grafo obtenido es un grafo dirigido acíclico (demostrado en EDA). Luego posee un orden topológico (demostrado en EDA). Una vez obtenido un orden topológico podemos invertirlo y realizar las siguientes operaciones en orden:

Si una componente no tiene un valor asignado, se le asigna True a todos los nodos dentro de esa componente.

Una vez que a un nodo se le asigna un valor, se asigna el valor contrario a su complemento. Este valor se extiende a todos los nodos de la misma componente.

Si una componente se le asigna False, todas las componentes que tengan un camino hacia esa componente en el DAG también recibirán el valor False.

Este procedimiento garantiza que todas las asignaciones sean consistentes y que no haya contradicciones entre los nodos y sus complementos, ya que los literales a_i y $\neg a_i$ pertenecen a componentes diferentes.

Como este algoritmo siempre es posible podemos decir que el 2-SAT tiene solución si y solo si para todo nodo x se cumple que x y $\neq x$ pertenecen a distintas componentes fuertemente conexas.

Problema con la solución actual

Cuando se resuelve el problema para un prefijo p_i e las aristas, no se toma en cuenta el caso en el que existan dos aristas e_j, e_k tal que $e_j \in p_i$ y $e_k \notin p_i$ pero que e_j y e_k que compartan un nodo y tengan el mismo color. En este caso, para que exista una solución el prefijo p_i, e_j debe ser eliminado del grafo.

Para corregir esto, se puede agregar la cláusula $e_j \vee e_j$ en el 2-SAT, lo que fuerza la eliminación de $e_j del grafo y asegura que el prefijop_i cumpla con las restricciones.

Salida del programa

Al resolver el problema usando búsqueda binaria sobre los prefijos de las aristas y aplicando 2-SAT para verificar la validez de cada prefijo, se obtiene lo siguiente: si no se encuentra un prefijo que contenga una solución válida, el programa imprimirá "No". En caso de que exista una solución, el programa imprimirá "Yes", seguido del tiempo mínimo obtenido por la búsqueda binaria, junto con la cantidad de aristas a eliminar y las aristas correspondientes, de acuerdo con la asignación de True realizada por el 2-SAT en el prefijo válido.

Complejidad temporal

Para un conjunto de n literales y m cláusulas, el 2-SAT genera un grafo con $2n$ nodos y $2m$ aristas, y calcular las componentes fuertemente conexas toma $O(n+m)$. Revisar que un nodo y su complemento no pertenezcan a la misma componente se puede realizar en $O(n)$ operaciones.

La construcción del grafo de componentes también toma $O(n+m)$, y obtener su orden topológico tiene la misma complejidad.

Evaluar los literales y propagar la información requiere $O(n+m)$, ya que debemos asignarle valor a cada nodo y a lo sumo revisamos cada arista una única vez. Luego podemos resolver el predicado para un 2-SAT de n literales y m clausulas en tiempo $O(n+m)$.

Para el problema completo, se revisa que no haya más de dos pares de aristas del mismo color en cada nodo en $O(n+m)$. Antes de la búsqueda binaria, las aristas se ordenan por tiempo de eliminación en $O(m \log m)$. La búsqueda binaria hace $O(\log(m))$ evaluaciones, y cada evaluación del predicado genera $O(m^2)$ cláusulas, con una complejidad total de $O(n+m^2) \cdot \log m$.

Una forma de optimizar el número de cláusulas generadas en el predicado para el 2-SAT es evitar la creación de todas las posibles parejas de aristas que inciden en un mismo nodo. En su lugar, se pueden introducir k nuevos literales $p_1, p_2, \ldots, p_k$ para un conjunto de aristas $e_1, e_2, \ldots, e_k$ que comparten un nodo en común. Luego, se añaden las siguientes cláusulas:

$\neg e_i \vee p_i$ para cada i
$\neg p_{i-1} \vee p_i$ para todo $i \geq 1$
$\neg e_{i+1} \vee \neg p_i$ para cada i
Con este enfoque, se mantienen $O(m)$ literales, pero las cláusulas se reducen a $O(m)$ en lugar de $O(m^2)$. Esto reduce la complejidad del problema a $O((n+m)\log m)$.

Demostremos ahora que esta mejora es válida.

Sea un conjunto de aristas $e_1, e_2, e_3, \ldots e_k$ que inciden sobre un mismo nodo. Si no se elimina ninguna arista, las cláusulas (1) y (3) se cumplen, y la cláusula (2) se satisface si se asigna el valor True a todos los p_i. Si se elimina una sola arista e_i, se le asigna el valor True, y al resto de las aristas el valor False. Por la cláusula (1), p_i debe ser True, y asignando False a p_j para todo $j \neq i$, se cumplen las tres cláusulas para todos los i.

Si se intentan eliminar dos aristas e_i y e_j con $i < j$, sin pérdida de generalidad, la cláusula (1) implica que p_i es True. La cláusula (2) establece una relación de implicación entre los p_i, lo que significa que si p_i es True, entonces $p_{i+1}, p_{i+2}, \ldots, p_k$ también lo serán. Como $j > i$, p_{j-1} también será True. Sin embargo, por la cláusula (3), esto implica que e_j debe ser False, lo que genera una contradicción, demostrando que no se pueden eliminar dos aristas.

En conclusión, las nuevas cláusulas garantizan que a lo sumo se elimine una arista, lo que asegura el comportamiento deseado y valida la corrección del algoritmo.

Pruebas

Dada la complejidad del código, las pruebas realizadas en Codeforces presentaron dificultades significativas, particularmente debido a errores de tiempo de ejecución difíciles de diagnosticar. Para abordar estos problemas, se desarrolló un tester que se ejecutó 10,000 veces, produciendo respuestas correctas en todas sus ejecuciones.

Funcionamiento del Tester

A continuación, se describen sus componentes principales:

Generación de Entrada:

Se generan los valores de n y m, donde n varía entre 3 y 7 y m entre 3 y 15.
Las aristas se generan aleatoriamente. Para cada arista, se selecciona un primer vértice y un segundo vértice de manera que el segundo sea menor que el primero, evitando repeticiones. Además, cada arista recibe un color y un tiempo de eliminación aleatorios.

Pruebas de Salida:

Para verificar la salida del algoritmo, se calculan todas las posibles soluciones al problema, guardando el tiempo de la mejor solución si existe.

Se compara la respuesta del algoritmo con la respuesta correcta: si el algoritmo afirma que hay una solución, se valida que esto sea cierto. Si la solución existe, se comprueba que el tiempo de ejecución del algoritmo sea correcto y que la solución proporcionada sea válida.

Problema "NP Completo" - Planificación de Clases

Te han pedido organizar un seminario de nivel introductorio que se llevará a cabo una vez por semana durante el próximo semestre. El plan es que la primera parte del semestre consista en una serie de conferencias de invitados externos, y que la segunda parte del semestre se dedique a una serie de proyectos prácticos que realizarán los estudiantes.

Hay n opciones de conferencistas en total, y en la semana número i (para $i = 1, 2, \ldots, k$) un subconjunto L_i de estos conferencistas está disponible para dar una conferencia. Por otro lado, cada proyecto requiere que los estudiantes hayan visto ciertos materiales previos para poder completarlo con éxito. En particular, para cada proyecto j (para $j = 1, 2, \ldots, p$), hay un subconjunto P_j de conferencistas relevantes, de manera que los estudiantes deben haber asistido a una conferencia de al menos uno de los conferencistas del conjunto P_j para poder realizar el proyecto.

Dado estos conjuntos, ¿puedes seleccionar exactamente un conferencista para cada una de las primeras k semanas del seminario, de manera que solo elijas conferencistas que estén disponibles en su semana designada, y para que, para cada proyecto j, los estudiantes hayan asistido a una conferencia de al menos uno de los conferencistas del conjunto relevante P_j?

Demostración

El problema es en NP, ya que, dada una secuencia de ponentes, podemos verificar (a) que todos los ponentes están disponibles en las semanas en que están programados, y (b) que para cada proyecto, al menos uno de los ponentes relevantes ha sido programado.

En la planificación de conferencias, al igual que en muchos problemas de satisfacción de restricciones, podemos identificar dos fases conceptuales. En la primera fase, se seleccionan algunos ponentes de un conjunto de opciones, cerrando la puerta a otros; en la segunda fase, se determina si estas elecciones han dado lugar a una solución válida.

Específicamente, en el caso de la planificación de conferencias, la primera fase consiste en seleccionar un ponente para cada semana, mientras que la segunda fase implica verificar que se ha seleccionado un ponente relevante para cada proyecto. Esta estructura conceptual es característica de muchos problemas NP-completos, lo que nos permite buscar reducciones plausibles. A continuación, describiremos dos reducciones: primero de 3-SAT y luego de Vertex Cover. Ambas son suficientes por sí solas para demostrar la NP-completitud.

El problema 3-SAT es un caso canónico con la estructura de dos fases mencionada. Primero, se evalúan las variables, asignando a cada una un valor de verdadero o falso; luego, se revisan las cláusulas para comprobar si las elecciones realizadas las satisfacen. Este paralelismo con la planificación de conferencias sugiere una reducción natural que establece que 3\text{-SAT} \leq_P \text{Planificación de Conferencias}. En esta organización, la elección de los conferenciantes representa la asignación de valores a las variables, y la viabilidad de los proyectos corresponde a la satisfacción de las cláusulas.

Más concretamente, supongamos que se nos da una instancia de 3-SAT que consiste en cláusulas $C_1, \ldots, C_k$ sobre las variables $x_1, x_2, \ldots, x_n$. Construimos una instancia de planificación de conferencias de la siguiente manera. Para cada variable x_i, creamos dos conferenciantes z_i y z_i' que corresponderán a x_i y su negación. Comenzamos con n semanas de conferencias; en la semana i, los únicos dos conferenciantes disponibles son z_i y z_i'. Luego hay una secuencia de k proyectos; para el proyecto j, el conjunto de conferenciantes relevantes P_j consiste en los tres conferenciantes correspondientes a los términos de la cláusula C_j. Ahora, si hay una asignación satisfactoria ν para la instancia de 3-SAT, entonces en la semana i elegimos el conferenciante entre z_i, z_i' que corresponde al valor asignado a x_i por ν; en este caso, seleccionaremos al menos un ponente de cada conjunto relevante P_j. Inversamente, si encontramos una manera de elegir ponentes de tal manera que haya al menos uno de cada conjunto relevante, entonces podemos establecer las variables x_i de la siguiente manera: $x_i$se establece en 1 si se elige zi, y se establece en 0 si se elige z_i'. De esta manera, al menos uno de los tres variables en cada cláusula C_j se establece de manera que la satisface, y por lo tanto, esta es una asignación satisfactoria. Esto concluye la reducción y su prueba de correctitud.

De manera similar, la intuición detrás de la planificación de conferencias conduce a una reducción de Vertex Cover, que también presenta una estructura de dos fases: primero se selecciona un conjunto de k nodos del grafo de entrada, y luego se verifica que estas elecciones cubren todas las aristas.

Dada una entrada a Vertex Cover, que consiste en un grafo $G = (V, E)$ y un número k, creamos un conferenciante z_v para cada nodo v. Establecemos $l = k$, y definimos $L1 = L_2 = \ldots = L_k = z_v : v \in V$. En otras palabras, durante las primeras k semanas, todos los conferenciantes están disponibles. Después de esto, creamos un proyecto j para cada arista $e_j = (v, w)$, con conjunto $P_j = z_v, z_w$.

Ahora, si existe un Vertex Cover S de como máximo k nodos, entonces consideremos el conjunto de conferenciantes $Z_S = z_v : v \in S$. Para cada proyecto P_j, al menos uno de los conferenciantes relevantes pertenece a Z_S, ya que S cubre todas las aristas en G. Además, podemos programar todos los conferenciantes en Z_S durante las primeras k semanas. Por lo tanto, se sigue que hay una solución factible a la instancia de planificación de conferencias.

Inversamente, supongamos que hay una solución factible a la instancia de planificación de conferencias, y dejemos que T sea el conjunto de todos los conferenciantes que hablan en las primeras k semanas. Sea X el conjunto de nodos en G que corresponden a los conferenciantes en T. Para cada proyecto P_j, al menos uno de los dos conferenciantes relevantes aparece en t, y por lo tanto, al menos uno de los extremos de cada arista e_j está en el conjunto X. Así, X es una cubierta de vértices con como máximo k nodos.

Esto concluye la prueba de que Vertex Cover se reduce a Planificación de Conferencias en tiempo polinómico.