RESOLUCIÓN DE PROBLEMAS DE DISEÑO Y ANÁLISIS DE ALGORITMOS

José Carlos Pendas

Problema 1

En una futurística ciudad llena de robots, cada robot pertenece a una facción identificada por un color específico. Las piezas de los robots fueron creadas por varios fabricantes distintos, y algunos robots tienen piezas hechas por fabricantes en común. Los robots de diferentes facciones están buscando formar una alianza estratégica multicolor. Esta alianza debe ser un grupo de robots donde todos tengan, dos a dos, alguna pieza hecha por el mismo fabricante, y lo más importante: debe haber al menos un robot de cada color en esta alianza. Dado un grupo de robots, determine si es posible o no formar una alianza multicolor.

Modelación

En el problema planteado anteriormente, nos percatamos que se puede construir un grafo de una forma que explicaremos próximamente, de manera que cada instancia de este sea equivalente a las del problema del conjunto independiente. Por lo tanto, si logramos resolver el problema del conjunto independiente, también habremos resuelto nuestro problema. Esto nos indica que el problema planteado es reducible al problema del conjunto independiente.

Modelado del Problema

Para modelar este problema en un grafo, seguimos los siguientes pasos:

1. **Vértices (robots)**: Cada vértice del grafo representa un robot.
2. **Aristas (relaciones entre robots)**: Para todo par de robots u y v:
    - Si los robots u y v tienen el mismo color, trazamos una arista entre ellos.
    - Si los fabricantes de las piezas de los robots u y v son disjuntos (no comparten ningún fabricante), trazamos una arista entre ellos.

Características del Conjunto de Nodos

Dado este modelo, un conjunto de nodos en el grafo cumple con las siguientes características:

- Para todo par de nodos u y v, si no existe una arista entre ellos, significa que los robots tienen colores diferentes y comparten al menos un fabricante en común.

Objetivo

El objetivo es encontrar un conjunto de vértices que cumpla con las condiciones planteadas, es decir, que para todo par de robots del conjunto exista al menos un fabricante compartido y que los robots sean de colores distintos. Si encontramos un conjunto de tamaño k, donde k es el número de colores, entonces este conjunto representa una alianza multicolor.

Conjunto Independiente

Nuestro problema se reduce a encontrar un conjunto independiente en el grafo que cumpla con las condiciones anteriores.Un conjunto independiente es un conjunto de vértices en un grafo tal que ninguno de sus vértices es adyacente a otro. Es decir, es un conjunto V de vértices tal que para ningún par de ellos existe alguna arista que los conecte. En otras palabras, cada arista en el grafo contiene a lo más un vértice en V. El tamaño de un conjunto independiente es el número de vértices que contiene.

Un conjunto independiente maximal es un conjunto independiente tal que, al añadir cualquier otro vértice, este deja de ser independiente. Por otro lado, el conjunto independiente máximo corresponde al mayor conjunto independiente definible sobre un grafo dado. El problema de encontrar un conjunto con estas características se llama problema del máximo conjunto independiente y es NP-completo.

Observación

Observemos que un conjunto independiente de vértices equivale a un conjunto de robots que cumplen con las características de no ser del mismo color y tener un fabricante en común. Si este conjunto tiene un tamaño k, entonces representa una alianza multicolor; de lo contrario, no existe una alianza multicolor. Además, es fácil notar que el

conjunto independiente de tamaño máximo debe tener como máximo k vértices.

¿Por qué? Porque contamos con k colores. Si el conjunto contiene más de k vértices, necesariamente habrá vértices conectados entre sí, ya que compartirían el mismo color, por el principio del palomar.

Transformación de Conjunto Independiente a Cobertura de Vértices

Para abordar el problema de conjunto independiente, recurriremos al problema de *vertex cover* (cobertura de vértices), que es uno de los problemas más emblemáticos en la categoría de NP-completos. Una cobertura de vértices de un grafo es un conjunto de vértices que incluye al menos uno de los extremos de cada arista del grafo. Este problema fue uno de los 21 problemas NP-completos identificados por Richard Karp y, por lo tanto, es un clásico en la teoría de la complejidad computacional, como vimos en clases.

Relación entre Conjunto Independiente y Cobertura de Vértices

Una propiedad fundamental de estos problemas es que un conjunto de vértices forma una cobertura de vértices si y sólo si su complemento es un conjunto independiente.

Demostración

Supongamos que tenemos un conjunto de cobertura de vértices $C \subseteq V$, y su complemento, $V \setminus C$, no es un conjunto independiente. Esto implica que existen al menos dos vértices u y v en $V \setminus C$ tales que $(u, v) \in E$, es decir, comparten una arista en el grafo.

Si esta arista (u, v) conecta vértices en $V \setminus C$, entonces la arista no está cubierta por ningún vértice en C, lo que contradice la definición de cobertura de vértices. Formalmente, una cobertura de vértices $C \subseteq V$ de un grafo no dirigido $G = (V, E)$ es un subconjunto de vértices tal que para cada arista $(u, v) \in E$, al menos uno de sus extremos pertenece a C. Es decir:

$$(u, v) \in E \Rightarrow u \in C \vee v \in C$$

Supongamos que $V \setminus C$ es un conjunto independiente y que $C \subseteq V$ es su complemento, pero que C no es un conjunto de cobertura de vértices. Esto significa que existe al menos una arista $(u, v) \in E$ tal que ninguno de los dos vértices u ni v pertenece a C. En otras palabras, tanto u como v pertenecen a $V \setminus C$.

Sin embargo, esto contradice la suposición de que $V \setminus C$ es un conjunto independiente, ya que, por definición, un conjunto independiente no puede contener vértices que compartan una arista. Si existe una arista (u, v) con ambos vértices en $V \setminus C$, entonces $V \setminus C$ no es independiente.

Por lo tanto, si $V \setminus C$ es un conjunto independiente, C debe ser una cobertura de vértices, es decir, debe cubrir todas las aristas del grafo.

En consecuencia, el número de vértices de un grafo $G = (V, E)$ es igual a la suma del tamaño de su conjunto mínimo de cobertura de vértices y el tamaño de un conjunto independiente máximo.

Esta relación nos permite transformar el problema de encontrar un conjunto independiente máximo en el problema de buscar una cobertura de vértices mínima, y viceversa. Esta equivalencia es fundamental para el estudio y la resolución de problemas NP-completos en teoría de grafos.

Solución

La solución a este problema consiste en encontrar una cobertura de vértices mínima. A partir de esta cobertura, determinamos su complemento. Si el complemento tiene tamaño k, entonces existe una alianza multicolor; de lo contrario, no existe tal alianza.

La variante de decisión del problema de cobertura de vértices es NP-completa, lo que implica que, a medida que el tamaño del grafo crece, el tiempo de ejecución de cualquier algoritmo exacto aumenta exponencialmente. Esto lo hace inviable para grafos grandes.

Fuerza Bruta

Una forma exacta de resolver el problema es mediante **fuerza bruta**, generando todos los subconjuntos posibles de vértices y verificando si cubren todas las aristas. Si se encuentra un subconjunto mínimo que cubra todas las aristas, se tiene la solución exacta. Sin embargo, este método tiene una **complejidad exponencial**, con 2^V subconjuntos para un grafo con V vértices, por lo que solo es práctico para grafos pequeños.

Algoritmos Aproximados

Dado que la fuerza bruta no es viable para grafos grandes, recurrimos a **algoritmos aproximados** que proporcionan soluciones razonablemente buenas en tiempos más cortos. A continuación, presentamos un algoritmo aproximado para la cobertura de vértices:

Algoritmo Aproximado de Cobertura de Vértices

El problema de la cobertura de vértices óptima es una versión de optimización de un problema NP-completo. Sin embargo, encontrar una cobertura de vértices que sea casi óptima no resulta demasiado difícil."

Algoritmo: APPROX-VERTEX_COVER (G: Grafo)

- $c \leftarrow \{\}$
- $E' \leftarrow E[G]$
- Mientras E' no esté vacío, hacer:
  - Sea (u, v) una arista arbitraria de E'
  - $c \leftarrow c \cup \{u, v\}$
  - Eliminar de E' toda arista incidente en u o v
- Retornar c

Análisis

Es fácil ver que el tiempo de ejecución de este algoritmo es $O(V + E)$, utilizando una lista de adyacencia para representar E'.

Teorema

`APPROX-VERTEX-COVER` es un algoritmo 2-aproximado de tiempo polinómico, es decir, el algoritmo tiene una cota de razón de 2.

Objetivo

Dado que este es un problema de minimización, estamos interesados en el menor valor de c/c^* posible. Específicamente, queremos mostrar que:

$$\frac{c}{c^*} \leq 2 = p(n)$$

En otras palabras, queremos demostrar que el algoritmo `APPROX-VERTEX-COVER` devuelve una cobertura de vértices que es, como máximo, el doble del tamaño de una cobertura óptima.

Demostración

Sean los conjuntos c y c^* los conjuntos resultantes de `APPROX-VERTEX-COVER` y `OPTIMAL-VERTEX-COVER`, respectivamente. Además, sea A el conjunto de aristas seleccionadas en la línea 4.

Dado que hemos añadido ambos vértices, obtenemos:

$$|c| = 2|A| \quad \text{pero} \quad |c^*| \leq |A| + 1.$$

Esto implica:

$$\Rightarrow \frac{c}{c^*} \leq 2.$$

Formalmente, como ningún par de aristas en A está cubierto por el mismo vértice de c^* (ya que, una vez que se selecciona una arista en la línea 4, todas las demás aristas incidentes en sus extremos se eliminan de E' en la línea 6), tenemos la siguiente cota inferior:

$$|c^*| \geq |A|$$

En la línea 4, seleccionamos ambos extremos, lo que da una cota superior en el tamaño de la cobertura de vértices:

$$|c| \leq 2|A|.$$

Dado que la cota superior es exacta en este caso, tenemos:

$$|c| = 2|A|.$$

Tomamos $|c|/2 = |A|$ y lo sustituimos en la ecuación anterior:

$$|c^*| \geq \frac{|c|}{2}.$$

Por lo tanto:

$$\frac{|c^*|}{|c|} \geq \frac{1}{2}.$$

Esto nos lleva a:

$$\frac{|c^*|}{|c|} \leq 2 = p(n),$$

lo que prueba el teorema.

Nota

La implementación en Python de este algoritmo está disponible en el repositorio asociado.

Problema 2

Ante la inminente visita de una figura importante, los habitantes de una ciudad entraron en pánico y decidieron reparar las calles por las que podría transitar. Sin embargo, no sabían con certeza cuál sería la ruta exacta, por lo que recurrieron a Max en busca de ayuda.

La ciudad está compuesta por n puntos importantes, conectados por calles cuya longitud es conocida. Se sabe que José, la figura importante, iniciará su viaje en uno de estos puntos (s) y lo terminará en otro (t). Los ciudadanos desean saber cuántas calles (aristas) participan en **algún camino de distancia mínima** entre todos los posibles pares de puntos s y t.

Observación

Para resolver este problema, inicialmente se pensó en una solución basada en programación dinámica que calcule el costo mínimo entre cada par de vértices. La idea es mantener una matriz que registre la menor distancia encontrada entre cada par de puntos, junto con el número de caminos que logran esa distancia mínima. Si se encuentra un nuevo camino con una menor distancia, se actualiza la solución reemplazando el camino anterior. Si se halla otro camino con la misma distancia mínima, este se suma al conjunto de caminos óptimos.

El primer enfoque consistió en modificar el algoritmo de **Floyd-Warshall**, que resuelve el problema de caminos mínimos en grafos con complejidad $O(V^3)$. Aunque este método es efectivo, su complejidad resulta ineficiente para grafos grandes. Por lo tanto, se propone un enfoque más eficiente basado en la búsqueda de caminos mínimos, que tiene una complejidad de $O(VE)$, mejorando el rendimiento en grafos dispersos. En el peor de los casos, este enfoque se equipara a $O(V^3)$, pero en la mayoría de los casos, es mucho más rápido.

Idea principal

El algoritmo de Floyd-Warshall utiliza la programación dinámica para encontrar el camino más corto entre todos los pares de nodos de un grafo. La clave está en construir una solución de manera incremental, refinando las distancias mínimas paso a paso hasta alcanzar la solución óptima.

Conceptos clave

Matriz de distancias: El algoritmo mantiene una matriz dist[i][j], donde cada celda dist[i][j] representa la distancia mínima entre los nodos i y j. Inicialmente, la matriz se llena con los pesos de las aristas que conectan directamente los nodos. Si no existe una arista directa entre i y j, se inicializa con infinito (∞).

Relajación de caminos: Para cada nodo k, el algoritmo revisa si el camino entre i y j puede mejorarse pasando a través de k. Si el camino que pasa por k resulta más corto que el existente entre i y j, se actualiza la distancia en la matriz de distancias. La fórmula utilizada para esta actualización es:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

Esta operación se repite para todos los pares de nodos i y j, considerando cada nodo k como intermediario, hasta que todas las combinaciones posibles hayan sido evaluadas.

■ Problema 3

Te dan una cadena S de longitud N y un número entero K.

Sea C el conjunto de todos los caracteres en S. La cadena S se llama *buena* si, para cada sufijo de S:

La diferencia entre las frecuencias de cualquier par de caracteres en C no excede K. En particular, si el conjunto C tiene un solo elemento, la cadena S es buena.

El objetivo es encontrar si existe una reordenación de S que sea buena. Si existen múltiples reordenaciones de este tipo, imprime la reordenación lexicográficamente más pequeña. Si no existe tal reordenación, imprime -1.

Nota que un sufijo de una cadena se obtiene eliminando algunos (posiblemente cero) caracteres desde el principio de la cadena. Por ejemplo, los sufijos de $S = abca$ son $\{a, ca, bca, abca\}$.

Observaciones

Para resolver este problema, hemos desarrollado una heurística *greedy* un poco compleja de entender, pero intentaremos explicarla de la mejor manera posible. Hemos probado la solución y creemos que es correcta, aunque su escritura se ha vuelto complicada. Estamos refinando la idea y la presentaremos en esta sección en breve

El algoritmo propuesto se basa en la idea de organizar los caracteres de menor valor lexicográfico de izquierda a derecha. En este contexto, una cadena se considera de primer orden si, para cualquier posición k en el rango de 0 a n, la subcadena que va desde la posición 0 hasta k tiene el menor valor lexicográfico posible. Para lograr esto, nuestro enfoque se divide en varias etapas.

Primero, se crea un arreglo C que contiene el conjunto de caracteres sin repeticiones, ordenados de

mayor a menor valor lexicográfico. Además, se define una tupla por cada carácter, que incluye tres elementos: 1. **value**: el carácter en cuestión, 2. **frequency**: la cantidad de veces que aún queda por colocar dicho carácter, 3. **max**: el índice del carácter en C que tiene el mayor valor lexicográfico y que, además, presenta la mayor frecuencia.

Estos datos nos permitirán implementar una heurística greedy.

El procedimiento inicia recorriendo C de forma inversa y evaluando cuántos caracteres se pueden insertar consecutivamente sin desbalancear la distribución de frecuencias con respecto al carácter de mayor frecuencia. Es fundamental asegurar que, si quedan caracteres por insertar de un mismo valor, no se genere una desventaja en la frecuencia mayor que $k + 1$ respecto al carácter con la mayor frecuencia.

Por ejemplo, consideremos la cadena "aaaaabbbbbccccc" y un valor de $k = 2$. En este caso, el arreglo C sería: $[(c, 5, 0), (b, 5, 0), (a, 5, 0)]$.

El resultado esperado sería "aabbcabcaabbcc". Al iniciar colocando dos caracteres 'a', el siguiente carácter debe ser una 'b'. Si se insertara una tercera 'a', quedarían dos 'a' para emparejarse con cinco 'c', lo que generaría un sufijo inválido.

El proceso se desarrolla de la siguiente manera: 1. Se inserta un carácter 'a' y se verifica si la cantidad restante de 'a' es suficiente para equilibrarse con el carácter de mayor frecuencia. Si no es posible, se inserta el carácter de mayor frecuencia disponible, en este caso, una 'b', y se repite el proceso hasta que todos los caracteres hayan sido insertados.

Este razonamiento greedy asegura que siempre se obtenga la menor ordenación lexicográfica posible de la cadena. Al mantener un seguimiento de las frecuencias, garantizamos que la cadena resultante sea óptima, pues se preserva un equilibrio adecuado entre los sufijos.

Para calcular la frecuencia de cada carácter en la cadena, se utiliza un diccionario. Cada vez que se repite un carácter, se incrementa un contador asociado. Posteriormente, se ordenan las claves del diccionario de mayor a menor frecuencia, formando así una lista de tuplas. A través de un recorrido de esta lista, se determina la frecuencia máxima alcanzada hasta la posición actual. Este proceso tiene una complejidad de $O(n \log n)$.

Finalmente, el algoritmo agrega caracteres de C y realiza operaciones en $O(n)$, lo que resulta en una complejidad total de $O(n + n \log n)$.

■ Observaciones