DAA

josé luis leiva fleitas- 412* & eduardo garcía maleta-411[1]

contents

1 corrupción

Problema:

Han pasado 20 años desde que Lidier se graduó de Ciencias de la Computación (haciendo una muy buena tesis) y las vueltas de la vida lo llevaron a convertirse en el presidente del Partido Comunista de Cuba. Una de sus muchas responsabilidades consiste en visitar zonas remotas. En esta ocasión debe visitar una ciudad campestre de Pinar del Río.

También han pasado 20 años desde que Marié consiguió su título en MATCOM. Tras años de viaje por las grandes metrópolis del mundo, en algún punto decidió que prefería vivir una vida tranquila, aislada de la urbanización, en una tranquila ciudad de Pinar del Río. Las vueltas de la vida quisieron que precisamente Marié fuera la única universitaria habitando la ciudad que Lidier se dispone a visitar.

Los habitantes de la zona entraron en pánico ante la visita de una figura tan importante y decidieron reparar las calles de la ciudad por las que transitaría Lidier. El problema está en que nadie sabía qué ruta tomaría el presidente y decidieron pedirle ayuda a Marié.

La ciudad tiene n puntos importantes, unidos entre sí por calles cuyos tamaños se conoce. Se sabe que Lidier comenzará en alguno de esos puntos (s) y terminará el viaje en otro (t). Los ciudadanos quieren saber, para cada par s, t, cuántas calles participan en algún camino de distancia mínima entre s y t.

1.1 Modelacion del problema

Representaremos la ciudad como un grafo dirigido $G = V, E$, donde los n puntos importantes de la ciudad estarán representados por n vértices (V) del grafo G y las calles que unen estos puntos estarán representadas por las aristas de dicho grafo (E). Entonces el problema lo analizaremos de la siguiente manera:

Sea $G = V, E$ un grafo,se quiere, para todo par de vértices s y t del grafo, determinar cuántas aristas participan en algún camino de distancia mínima entre s y t.

1.2 Solución 1: Utilizando el algoritmo de Dijkstra

Se le realiza al algoritmo la siguiente modificación: Sea $nodes[u][1]$ el costo de llegar desde origen hasta el vértice u. En cada nodo, en vez de guardar un padre, se guarda una lista de padres, de forma tal que cuando se haga $relaxuv$, si el costo computado hasta el momento de llegar a v desde el origen ($nodes[v][1]$), es igual al costo de ir del origen a u ($nodes[u][1]$) más el costo del arco (u, v) ($dist[u][v]$) entonces añadimos a la lista de padres de v ($nodes[v][2]$) el vértice u. Mediante estas listas de padres podemos reconstruir todos los caminos de costo mínimo que llegan a u. Se tendrá una matriz $solution_matrix$ donde $solution_matrix[i][j]$ almacenará la cantidad de arcos participantes en algun camino de costo mínimo entre los nodos i y j. Por cada vértice del grafo se ejecuta el algoritmo de Dijkstra con la modificación anterior. Cada vez que se termine una ejecución del algoritmo de Dijkstra para el origen de u, para cada par u, v se recorren todos los caminos de costo mínimo encontrados y se cuentan todas los arcos que participan en dichos caminos, los arcos se contaran solo una vez, pues a medida que se recorren los caminos ($path_seeker$) se iran marcando en una máscara booleana los arcos previamente contados. Terminado este proceso para cada uno de los vértices del grafo en $solution_matrix$ quedará almacenada la respuesta final.

Análisis de Correctitud

La modificación realizada en el bloque $relax$ en el algoritmo de Dijkstra no afecta la correctitud de este, pues no se modifica su comportamiento, solo agregamos información a los nodos. Luego podemos afirmar que se calculan correctamente los caminos de costo mínimo desde el orígen s hacia el resto de los nodos. Luego, el algoritmo calcula todos las caminos de costo mínimo entre cada par de nodos y cuenta los arcos que intervienen en los caminos una sola vez (gracias a la máscara booleana usada en $path_seeker$).

Análisis de Complejidad Temporal

La modificación realizada al algoritmo de Dijkstra no afecta su complejidad temporal. Luego por cada uno de los vertices v en $V(G)$:

- se realiza este algoritmo para obtener los caminos de costo mínimo desde v como origen hacia el resto, sea $n = |V(G)|$ esto tiene una complejidad temporal de $O(n^2)$, - por cada uno u del resto de vertices $(n-1)$ se reconstruyen los caminos de costo mínimo para actualizar $solution_matrix$, sea $m = |E(G)|$ esto tiene una complejidad de $O(mn^2)$ porque maximo pueden haber m caminos distintos desde v a u y maximo se visitaran n nodos, por cada uno de los $n-1$ nodos.

Luego la complejidad temporal es $O(n*(n^2+mn^2)) = O(mn^3) = O(|E||V|^3)$.

1.3 Solución 2: Utilizando el algoritmo de Floyd-Warshal

Realizamos el algoritmo de Floyd-Warshall a partir de la matriz de costos inicial ($dist$) para obtener la matriz de las distancias de costo mínimo entre todo par de vértices i, j, llamemos a esta matriz fw. Luego, para cada par de vértices i, j, con $i! = j$, hallamos todos los vértices k, tal que $fw[i, j] = fw[i, k] + fw[k, j]$, estos vértices pertencen a algún camino de costo mínimo de i a $j, alconjuntodeestosvrticesparacadapari, j$ llamémoslo $CCM(i, j)$. Luego, para cada par de vértices i, j distintos, hallamos el número de vértices p que cumplen que $fw[i, p] + dist[p, j] = fw[i, j]$, esta es la cantidad de aristas que llegan a j, pertenecientes a algún camino de costo mínimo proveniente de i, llamemos a este número $e(i, j)$. Luego hacemos un recorrido por cada par de vértices i, j, la cantidad de aristas que pertenecen a algún camino de costo mínimo de i a j es igual a la suma de las aristas que llegan a cada vértice k perteneciente a $CCM(i, j)$ en caminos de costo mínimo provenientes de i:

$S(i, j) = \sum e(i, k), \forall k \in CCM(i, j)$.

Análisis de Correctitud

Si una arista (x, y) pertence a algún camino de costo mínimo entre i y j, entonces: y es un vértice que pertence a algún camino de costo mínimo entre i y j y se cumple que: el costo mínimo de un camino de i a y sumado con el costo de un camino de costo mínimo de y a j es igual al costo de algún camino de costo mínimo de i a j.

Demostremos el enunciado anterior: el algoritmo de Floyd-Warshall calcula correctamente el costo del camino de costo mínimo entre todo par de vértices

i, j. Sea fw la matriz que devuelve el algoritmo de Floyd-Warshall. Todos los caminos de costo mínimo tienen el mismo costo por definición de camino de costo mínimo. Sea P algún camino de costo mínimo entre i y j en el cual participe el vértice y. Los caminos de costo mínimo poseen subestructura óptima, es decir, para cualquier par de vértices $v, w \in P$, el subcamino de P que va de v a w es un camino de costo mínimo entre v y w. Por tanto el subcamino de P de i a y es un camino de costo mínimo, luego el costo de dicho subcamino está almacenado en $fw[i, y]$. Análogamente se cumple para el subcamino de P que va de y a j y su valor está almacenado en $fw[y, j]$. Por tanto, $fw[i, y]$ es el costo del camino de costo mínimo que va de i a y, $fw[y, j]$ es el costo del camino de costo mínimo que va de y a j. Luego como $fw[i, y]$ es mínimo y $fw[y, j]$ es mínimo, su suma $(fw[i, y] + fw[y, j])$ es mínima. El costo del camino de costo mínimo entre i y j es $fw[i, j]$ y coincide con el costo de P. Luego en P se empieza en i, se pasa por y y luego se va a j, por tanto el costo de P es $fw[i, y] + fw[y, j]$ y como P es de costo mínimo entonces $fw[i, j] = fw[i, y] + fw[y, j]$, con $y \in P$. Luego queda demostrado que si y es un vértice que pertence a algún camino de costo mínimo entre i y j se cumple que el costo mínimo de un camino de i a y sumado con el costo de un camino de costo mínimo de y a j es igual al costo de algún camino de costo mínimo de i a j.

Luego, si (x, y) pertenece a algún camino de costo mínimo entre i y j, x, y pertenece a algún camino de costo mínimo entre i y y, pues todo subcamino de un camino de costo mínimo es de costo mínimo. Finalmente si (x, y) pertenece a algún camino de costo minimo entre i y y, entonces se cumple que la suma del costo de la arista (x, y) con el costo de un camino de costo mínimo de i a x debe ser igual al costo de un camino de costo mínimo de i a y.

Por lo cual: $y \in CCM(i, j)$ y $(x, y) \in e(i, y)$, y por lo tanto el algoritmo cuenta a (x, y).

Ahora demostremos que si el algoritmo cuenta una arista, esta pertence a algún camino de costo mínimo ente i y j: sea (p, j) una arista, si el algoritmo la cuenta, entonces esta arista cumple para algún vértice i, tal que exista al menos un camino de i a j, que $fw[i, p] + \mathtt{cost}[p, j] = fw[i, j]$, esto significa que el camino que empieza en i pasa por p y luega va a j es un camino de costo mínimo de i a j, pues su costo es $fw[i, p] + \mathtt{cost}[p, j]$ el cual coincide con el costo del camino de costo mínimo computado por el algoritmo de Floyd-Warshal, $fw[i, j]$. Luego la arista (p, j) participa en un camino de costo mínimo entre i y j.

Por último el algoritmo cuenta las aristas que pertenecen a caminos de costo mínimo entre i y j una sola vez pues el algoritmo cuenta las aristas que llegan a un vértice y revisa un vértice solo una vez.

Análisis de Complejidad Temporal

El algoritmo genera la matriz de los costos caminos de costo mínimo para todos los pares de vértices i, j utilizando el algoritmo de Floyd-Warshall, este tiene una complejidad temporal $O(|V|^3)$, luego por cada elemento de la matriz, o sea, par de vértices, (excepto los pares i, j donde i y j son iguales) el algoritmo recorre todos los vértices del grafo (excepto i), para reconocer cuales vértices pertenecen a caminos de costo mínimo entre i y j y cuántas aristas de la forma (k, j) pertenecen a caminos de costo mínimo de i a j, esta comprobación sucede en tiempo constante c, por lo cual este procedimiento tiene una complejidad temporal $O(|V|^3)$. Luego, para cada par de vértices i, j se recorren los vértices previamente calculados que pertenecen a caminos de costo mínimo de i a j (a lo sumo $|V|-1$) y se suma el número de aristas que llegan a cada uno de esos vértices pertenecientes a caminos de costo mínimo provenientes de i, este procedimiento tiene una complejidad temporal $O(|V|^3)$. Por tanto la complejidad temporal del algoritmo es $O(3|V|^3)$, que es $O(|V|^3)$.

2 banco

Eduardo es el gerente principal de un gran banco. Tiene acceso ilimitado al sistema de bases de datos del banco y, con unos pocos clics, puede mover cualquier cantidad de dinero de las reservas del banco a su propia cuenta privada. Sin embargo, el banco utiliza un sofisticado sistema de detección de fraude impulsado por IA, lo que hace que robar sea más difícil.

Eduardo sabe que el sistema antifraude detecta cualquier operación que supere un límite fijo de M euros, y estas operaciones son verificadas manualmente por un número de empleados. Por lo tanto, cualquier operación fraudulenta que exceda este límite es detectada, mientras que cualquier operación menor pasa desapercibida.

Eduardo no conoce el límite M y quiere descubrirlo. En una operación, puede elegir un número entero X y tratar de mover X euros de las reservas del banco a su propia cuenta. Luego, ocurre lo siguiente.

Si $X \leqslant M$, la operación pasa desapercibida y el saldo de la cuenta de Eduardo aumenta en X euros. De lo contrario, si $X > M$, se detecta el fraude y se cancela la operación. Además, Eduardo debe pagar X euros de su propia cuenta como multa. Si tiene menos de X euros en la cuenta, es despedido y llevado a la policía. Inicialmente, Eduardo tiene 1 euro en su cuenta. Ayúdalo a encontrar el valor exacto de M en no más de 105 operaciones sin que sea despedido.

Entrada: Cada prueba contiene múltiples casos de prueba. La primera línea contiene el número de casos de prueba $t(1 \leqslant t \leqslant 1000)$.

Para cada caso de prueba, no hay entrada previa a la primera consulta, pero puedes estar seguro de que M es un número entero y $0 \leqslant M \leqslant 1014$.

Salida: Para cada caso de prueba, cuando sepas el valor exacto de M, imprime una única línea con el formato "! M". Después de eso, tu programa debe proceder al siguiente caso de prueba o terminar, si es el último.

2.1 Modelación del problema

Dado que se conoce que $1 \leqslant M \leqslant 1014$ entonces podemos reducir el problema a buscar M en el conjunto P (ordenado) donde $P \subset Z \wedge P = [1 : 1014]$. Sea $is\_gt\_M(k)$ una función que determina si $k > M$ entonces se puede realizar una busqueda binaria en el conjunto P. Para realizar la búsqueda binaria se debe tener en cuenta que preguntar por un k específico no puede ser siempre en la mitad. Es necesario tener en cuenta que lo mejor que se puede preguntar siempre para asegurar descartar la mayor cantidad de soluciones posibles es la mitad del conjunto, pero dada la particularidad de que cada vez que la función $is\_gt\_M(k)$ responda $True$ la cantidad de dinero disponible disminuye en k y nuestra cantidad de dinero nunca debe ser menor que 0. Teniendo en cuenta que a es el último acierto conocido, en términos del problema carece de total sentido preguntar por un número k donde $k < a$ y sólo preguntamos por a si necesitamos recuperar dinero.

2.2 Solución: Búsqueda Lineal

El algoritmo busca el valor de M desde (lb) hasta (ub) y para cuando is_gt_M evalua $True$.

Parámetros :

-lb : Límite inferior del rango de búsqueda.

-ub : Límite superior del rango de búsqueda.

- $is\_gt\_M()$: Función que toma una suposicion y devuelve True si la esta es mayor que M y False en caso contrario

Variables internas:

-$fuel$: contador de iteraciones realizadas dureante la búsqueda.

3. Correctitud :

El algoritmo siempre encuentra M pues $lb <= M < ub$, pero no siempre en la cantidad de operaciones máxima. Si $0 <= M <= 1014$ y la cantidad de operaciones maxima a realizar es 105, entonces el algoritmo solo encontrará a M de forma correcta si está entre 0 y 104 o entre 1014 y 910 (si la búsqueda es reversa).

Análisis de Complejidad Temporal :

La complejidad temporal del algoritmo es $O(ub - lb)$ en general, porque en el peor de los casos necesita hacer esa cantidad de preguntas para encontrar a M.

2.3 Solución: Búsqueda Binaria

El objetivo principal del algoritmo es determinar el valor de M, que se encuentra entre un límite inferior (lb) y un límite superior (ub).

Parámetros :

-lb : Límite inferior del rango de búsqueda.

-ub : Límite superior del rango de búsqueda.

- $is\_gt\_M()$: Función que toma una suposicion y devuelve True si la esta es mayor que M y False en caso contrario

Variables internas:

-$money$: representa la cantidad de dinero actual.

-$fuel$: contador de iteraciones realizadas dureante la búsqueda.

1.Inicialización

Comenzamos con los siguientes valores:

- Saldo inicial : 1

- Límite inferior: 0.

2. Proceso de Búsqueda :

- Sea a último acierto conocido.
- Sea b último fallo conocido.
- Sea m cantidad de dinero disponible.
- Sea k número que vamos a comparar con M.

Entonces $k = min(m, (a+b)/2)$. Es decir si podemos preguntaremos por la mitad del conjunto de búsqueda restante, en otro caso preguntaremos por la cantidad de dinero que tenemos disponible. Como resultado si $a == b$ entonces $a == M$.

3. Correctitud :

La correctitud del algoritmo se basa en :

1. Invariantes Mantenidas : Durante cada iteración, los límites $('lb'y'ub')$ son ajustados correctamente según la respuesta de la función.

2. Salida Final : Al finalizar, el algoritmo devuelve un par $(lb, fuel)$ donde lb representa un valor que estamos buscando y $fuel$ es un contador de las iteraciones que fueron necesarias para llegar allí.

3. Convergencia : Dado que los límites están siendo ajustados adecuadamente y hay condiciones claras para salir del ciclo, sabemos que el algoritmo converge a la solución.

4. Complejidad temporal:

En general, la complejidad temporal puede ser considerada como :
$O(n)$ en el peor caso, donde n es la diferencia entre lb y ub.

$O(log(n))$ en casos más optimistas donde el rango se reduce significativamente en cada iteración.

3 rompiendo amistades

Por algún motivo, a José no le gustaba la paz y le irritaba que sus compañeros de aula se llevaran tan bien. Él quería ver el mundo arder. Un día un demonio se le acercó y le propuso un trato: "A cambio de un cachito de tu alma, te voy a dar el poder para romper relaciones de amistad de tus compañeros, con la única condición de que no puedes romperlas todas". Sin pensarlo mucho (Qué más da un pequeño trocito de alma), José aceptó y se puso a trabajar. Él conocía, dados sus k compañeros de aula, quiénes eran mutuamente amigos.

Como no podía eliminar todas las relaciones de amistad, pensó en qué era lo siguiente que podía hacer más daño. Si una persona quedaba con uno o dos amigos, podría hacer proyectos en parejas o tríos (casi todos los de la carrera son así), pero si tenía exactamente tres amigos, cuando llegara un proyecto de tres personas, uno de sus amigos debería quedar afuera y se formaría el caos.

Ayude a José a saber si puede sembrar la discordia en su aula eliminando relaciones de amistad entre sus compañeros de forma que todos queden, o bien sin amigos, o con exactamente tres amigos.

3.1 Modelación del problema

Representaremos los k compañeros de José como los vértices de un grafo no dirigido G = V,E de tamaño k, donde las relaciones de amistad entre ellos están representadas por las aristas entre los vértices del grafo. Entonces analizaremos el problema de la siguiente forma :

Es posible eliminar aristas del grafo G (sin eliminarlas todas) de manera tal que todos los vértices queden, o bien aislados, o con exactamente 3 vecinos.

También podemos verlo de la siguiente manera:

Dado un grafo G determinar si existe un subgrago G′ de G tal que G′ es regular de grado 3.

3.2 Solución: Usando Backtracking

El algoritmo hace justo lo que se explica en el apartado de modelación, elimina aristas (sin eliminar todas) y comprueba si en la version del grafo resultante, cada vértice tiene una vecindad de tamaño 3 o ninguna vecindad. Esto se logra explorando todos las versiones del grafo resultantes de quitar cualquier subconjunto de $E(G)$. La exploración consiste en recorrer por las aristas del grafo y, por cada una de ellas, explorar recursivamente el resto con la arista removida y luego explorar el resto de forma normal. Cada vez que explora recursivamente el grafo sin una arista, se comprueba si este grafo es solucion del problema, en caso afirmativo el algoritmo para, de lo contrario sigue buscando.

Análisis de Correctitud

Dada la naturaleza de exploración del algoritmo, este puede comprobar toda versión del grafo resultante de substraerle cualquier subconjunto de aristas de $E(G)$, y puesto que puede comprobar todas, comprobara específicamente aquellas donde cada vértice tenga un vecindad de tamaño 3, o donde ningún vértice tiene aristas incidentes.

Análisis de Complejidad Temporal

Como por cada arista a, el algoritmo explora recursivamente los grafos $G - a$ y G, la complejidad temporal es exponencial, $O(2^{|}E(G)|)$ en primera instancia. Pero la implementación del grafo g en el algoritmo usa una matriz de adyacencia para almacenar las aristas, y por tanto $g.traverse$ tiene una complejidad temporal $O(n^2)$ aunque produzca ($yield$) las $E(G)$ aristas del grafo. Por tanto la complejidad temporal $O(2^{(}n^2))$.

3.3 Np-Completitud

Para demostrar que un problema es $NP - \mathtt{Completo}$ hay que demostrar que:

1. El problema está en NP.

2. Se puede reducir un problema conocido como $\mathrm{NP}-\mathtt{Completo}$ a este problema en tiempo polinómico.

Paso1: Probar que el problema de encontrar un subgrafo regular de grado 3 (SR3) está NP

Dado un grafo G y un conjunto de aristas que supuestamente forman un subgrafo regular de grado 3, podemos verificar en tiempo polinomial si este subgrafo es regular de grafo 3. Para ello, recorremos todas las aristas del subgrafo y contamos el grado de cada vértice. Si todos los vértices tienen grado 3, entonces el subgrafo es regular de grado 3. Este procedimiento tiene un costo de $O(|V|+|E|)$, siendo |V| la cantidad de vértices y |E| la cantidad de aristas.

Paso2: Demostrar que SR3 es NP-Completo

Para demostrar que el problema SR3 es NP-completo, se necesita realizar una reducción desde un problema conocido como NP-completo. En este caso, se utilizará el problema de 3-coloración.

Problema de 3-coloración

Dado un grafo no dirigido $G = (V, E)$ determinar si existe una partición del conjunto de vértices V, de la forma $V1, V2, V3$ tal que no existen aristas entre los vertices que pertenecen a la misma partición Vi.

Definiciones:
- G : Grafo de entrada del problema de 3-coloración.
- $V(G)$: Conjunto de vértices del grafo G.
-$E(G)$: Conjunto de aristas del grafo G.
-$d(v_i)$: Grado del vértice i.
-K'_n :Grafo completo de n vértices al que se le quita una arista (este grafo tiene todos los vértices con grado n-1, excepto dos vértices que tienen grado n-2).
-3-partición : Dividir el conjunto de vértices en 3 conjuntos disjuntos.

Paso3: Transformación de la entrada

A partir del grafo G, se construye un nuevo grafo G' siguiendo los pasos siguientes:

1. **Ciclos** : Por cada vértice v_i en $V(G)$, se crean 3 ciclos (denotados como : C_i^1, C_i^2, C_i^3), cada uno con una longitud de $2 * d(v_i) + 1$. Los vértices de cada ciclo son denotados como $c_{ij}^h, 1 \leqslant i \leqslant n, 1 \leqslant d \leqslant 2 * d(v_i) + 1, 1 \leqslant h \leqslant 3$.

2. **Subgrafos** : Por cada arista $e_j \in E(G)$ se crean 3 subgrafos en G' (D_j^1, D_j^2, D_j^3), donde cada uno es un K'_4. Los dos vértices con grado 2 en cada subgrafo son denotados como x_i^h y y_j^h

3. **Conexiones** : Por cada arista entre los vértices v_s y v_t en G :
- Se seleccionan dos vértices c_{sa}^h y c_{sb}^h (c_{sa}^h y c_{sb}^h) que aún tengan grado 2 de los ciclos correspondientes a v_s y v_t, C_s^h (C_t^h).

- Por cada $1 \leqslant h \leqslant 3$, se agregan a G' las aristas $< c^h_{sa}, x^h_j >, < c^h_{sb}, y^h_j >$ $, < c^h_{ta}, x^h_j > y < c^t_{tb}, y^h_j >$

Una vez consideradas todas las aristas en el paso (3), se tiene que para cada ciclo C^h_i $(1 \leqslant i \leqslant n, 1 \leqslant h \leqslant 3)$ solo queda un vértice con degree 2. Nombremos dichos vértices como w^h_i.

4. **Construcción Adicional**: Por cada $1 \leqslant i \leqslant n$, se construye un subgrafo U_i, que es un K'_4 más un vértice al que denominaremos u_i, los vértices de grado 2 del K'_4 los denominaremos x_i y y_i, el vértice u_i se une a los restantes del $K4'$ mediante una arista $< x_i, u_i >$.
Se toman todos los grafos U_i y se agregan a G'. junto con las aristas $< u_i, w^1_i >, < y_i, w^2_i >$ y $< y_i, w^3_i >$.

5. **Ciclo final** : Se agrega el ciclo C' de longitud $2n$ a G', conformado por los vértices $a_{11}, ..., a_{1n}, a_{21}, ..., a_{2n}$ y se agregan las aristas $< a_{pi}, u_i >$ para $p = 1, 2$.

Paso4: Demostrar que G es 3-coloreable si y solo si G′ tiene un subgrafo regular de grado 3

$(\Rightarrow)$ Sea G un grafo 3-coloreable, y una 3-partición de $V(G)$ tal que en cada conjunto V_i queden solo vértices de un mismo color de G, entonces existe un grafo H, subgrafo inducido de G' que es 3-regular, ya que siempre podemos tomar los vértices de la siguiente forma:

1. Todos los vértices a_{ij} etán en H
2. Todos los vértices u_i están en H.
3. Si v_i de G está en el conjunto V_c de la tricoloración, entonces los vértices c_{ij} del ciclo C^c_i están en H.
4. Si $c \neq 1$ para el conjunto V_c al que pertenece v_i entonces los vértices de U_i pertenecen a H.
5. Si la arista e_j es adyacente al vértice v_i que está en el conjunto V_c, entonces los vértices de D^c_j adyacente a C^c_i están en H.

El subgrafo H existe para cualquier 3-partición de G, y cuando la 3-partición corresponde con una coloración se puede comprobar que todos los vértices en H tienen grado 3, en principio en G' todos los vértices son de grado 3 excepto los x^c_{jp} y y^c_{jp} de los D^c_{jp} que son de grado 4 pero en el subgrafo se hace una construcción a partir de una coloración y en G' los D^c_{jp} grafos reemplazan aristas, entonces los vértices x^c_{jp} y y^c_{jp} están conectados a vértices de círculos que no pertenecerán ambos a H, de donde obligatoriamente quedan en grado 3. De igual modo ocurre con los vértices u_i y y_i de los U_i , quienes tienen grado 4 cada uno en G, pero como H lo formamos considerando la 3-partición y cada vértice puede estar sólo en uno de los 3 conjuntos, entonces de las aristas que inciden en u_i solo se quedan las 2 que lo conectan al ciclo C' y la que se corresponde a si v_i está en el conjunto V_1 o no, y en los casos en los que se toma algún y_i como parte del conjunto H, en él solo permanecen las dos aristas que lo conectan al resto del U_i y solo una de las que indica si el vértice está en el conjunto V_2 o en el V_3 de la 3-partición. Por tanto en H, todos los vértices tienen grado 3.

($\Leftarrow$) Si G' contiene un subgrafo 3-regular H, entonces sobre H se cumplen las siguientes propiedades:

1. Todos los vértices a_{pi} y u_i pertenecen a H.
2. Por cada i, $1 \leqslant i \leqslant n$, exactamente uno de los ciclos C_i^h está en H.
3. Por cada i, $1 \leqslant i \leqslant n$, si C_i^h está en H, entonces ningún otro C_j^h para toda j tal que $< i, j > \in E(G)$.

Dem1. Supongamos que en H no está alguno de los u_i , como ese vértice no estaría, sus adyacentes en C' quedarían con grado 2 por lo que no podrían estar en H, esto rompería el ciclo C' produciendo que ninguno de sus vértices esté en H y estos a su vez al no poder estar en H, reducirían en 2 la cantidad de aristas de los restantes u i por lo que ninguno de los u_i podría estar en H. Como cada u_i es adyacente a un vértice de los C_i^1 , no se pudiera tener ningún vértice de los C_i^1 en H y la eliminación de estos vértices quita la posibilidad a los de D_j^1 de pertenecer a H. De igual modo cada u_i es adyacente al x_i de los U_i , de donde ningún vértice de los U_i podría estar en H ya que x_i pasaría a tener grado 2, al no poder estar, tampoco podrían u i1 y u i2 quienes al quitar x i quedarían con grado 2 y como esos tampoco estarían, se debe quitar también y_i , este último al estar conectado con un vértice de C_i^2 y con uno de C_i^3 hace que estos queden con grado 2 por lo que rompen los ciclos haciendo que ninguno de sus vértices puedan estar en H, esto finalmente provoca que los vértices de D_i^2 y D_i^3 tampoco puedan estar en H, quedando $H = \emptyset$. Por tanto todos los u_i tienen que estar en H y esto condiciona que todos los a_{pi} estén en H ya que, entre las 3 aristas que permanezcan incidiendo en u_i en el subgrafo cúbico una de ellas obligatoriamente está conectada a uno de los a_{pi} , como los a_{pi} todos tienen grado 3 y son adyacentes a otros dos $a_{p'i'}$, si uno de ellos está, todos tienen que estar.

Dem2. Como todos los u_i están en H y los a_{pi} , entonces para que u_i tenga grado 3 en el grafo inducido por H, es necesario que a H se agregue o bien el vértice de C_i^1 al que u_i es adyacente o el x_i , pero nunca ambos. En caso que se agregue el vértice adyacente en C_i^1 , entonces todo el ciclo se debe agregar ya que el grado 3 de cada uno de los vértices depende de los restantes. En caso que se agregara el y_i este para completar su grado 3 requiere que se agregue el resto de los vértices del U_i y que para completar el grado 3 de y_i se agregue o bien su adyacente en C_i^2 o en C_i^3, pero nunca ambos porque si no, pasaría a tener grado 4. Con la adición de uno de estos vértices a H se debe agregar todo el ciclo C_i^h al que pertenece por los mismos motivos que se agregaría C_i^1 en el caso anterior. De este modo se garantiza que para cada i, se agrega uno y solo uno de sus C_i^h ciclos.

Dem3.
Si a H pertenecen los vértices de C_i^h , como todos ellos tienen degree exactamente 3, todos sus vecinos tienen que pertenecer también a H, de donde los vértices de los D_j^h de las aristas e_j que inciden en v_i en G pertenecen a H y los vértices x_j^h y y_j^h están conectados a vértices de C_i^h y tienen que para alcanzar el grado 3 obligatoriamente incluir a todos los vértices de D_j^h ; por lo que si para algún v_k adyacente a v_i , el ciclo C_k^h también estuviese en H, sea $e_{j'}$ la arista que une a v_i con v_k , los vértices $x_{j'}^h$ y $y_{j'}^h$ tienen que tener tanto la arista que los conecta con uno de los vértices de C_i^h como la que los conecta con uno de los vértices de C_k^h, además de que obligatoriamente tienen que estar conectados a sus dos adyacentes en $D_{j'}^h$ porque al menos tienen que

incluir uno y esto condiciona que se agregue el otro al ser adyacentes entre ellos y tener grado exactamente 3, por lo que en ese caso x_j^h, y y_j^h, quedarían con grado, por tanto si C_i^h está en H, ningún C_k^h tal que v_k adyacente a v_i en G, puede estar en H.

Por la proposición (2) implica que el subgrafo H es una 3-partición de los vértices de G′ tal que el vértice v_i está en la partición c si $C_i^c \in H$. La proposición (3) asegura que vértices adyacentes estén en diferentes conjuntos en la partición, de donde si G′ tiene H como subgrafo cúbico, entonces G es 3-coloreable

3.4 Solución Backtrack

Se implementó una solución para buscar un subgrafo regular de grado 3 utilizando una estrategia Backtrack.

Caso base

La función `bt` tiene una condición base que verifica si `friendship_count` es cero. Si es así, significa que se ha encontrado una combinación válida (aunque no necesariamente un subgrafo regular), lo cual es correcto.

Condiciones para Subgrafo Regular

La comprobación

```
if all(len(edges) == 3 for edges in edges_list)
```

asegura que todos los vértices en el subgrafo actual tienen exactamente 3 conexiones. Esta condición es necesaria para confirmar que se ha encontrado un subgrafo regular de grado 3.

Eliminación y Restauración de Aristas

Al eliminar una arista y hacer una llamada recursiva, el algoritmo explora todas las combinaciones posibles de conexiones. La restauración posterior garantiza que el estado del grafo se mantenga para futuras exploraciones.

Manejo de Excepciones

El uso de `_SlnFound` como excepción para indicar que se ha encontrado un subgrafo regular asegura que el flujo del programa pueda interrumpirse adecuadamente al encontrar una solución válida.

Conclusión

El algoritmo es correcto bajo las condiciones establecidas, ya que verifica exhaustivamente todas las combinaciones posibles y asegura que cualquier subgrafo encontrado cumple con la propiedad de ser regular de grado 3. Sin embargo, su eficiencia puede ser un problema práctico debido a su complejidad exponencial.

Complejidad temporal

La complejidad temporal del algoritmo se puede analizar considerando las operaciones realizadas en la función `sln` y en la función `bt`.

Función `sln`

- Inicializa el grafo con k vértices y agrega m aristas (donde m es el número de amistades). Este proceso tiene una complejidad de $O(m)$.

Función `bt`

- En el peor de los casos, la función `bt` realiza una búsqueda exhaustiva a través de todas las combinaciones posibles de aristas. Esto implica que, en cada llamada recursiva, se exploran las aristas del grafo.
- La complejidad de esta búsqueda puede ser aproximada a $O(2^m)$ en el peor caso, ya que se pueden eliminar o no eliminar cada arista, lo que genera un árbol de decisiones exponencial.

Comprobaciones dentro de `bt`

- Las comprobaciones para determinar si todos los vértices tienen grado 3 (`all(len(edges) == 3 for edges in edges_list)`) requieren tiempo lineal en relación al número de vértices, es decir, $O(n)$.
- Sin embargo, dado que se hace dentro de un contexto recursivo, esto se ejecuta múltiples veces a lo largo de la búsqueda.

Resumen
La complejidad temporal del algoritmo es aproximadamente $O(m + 2^m)$, donde m es el número de aristas. Esto indica que el algoritmo es exponencial en relación con el número de aristas, lo cual puede ser ineficiente para grafos grandes.

3.5 Aproximación

Para poder encontrar una solución aproximada de un problema de decisión lo que podemos hacer es transformarlo en un problema de optimización, cuya solución óptima nos permita encontrar la solución del problema original. Para el caso del problema del SR3 un posible problema de optimización correspondiente sería el siguiente.

Problema de Optimización: Dado un grafo no direigido G, encontrar un subgrafo $G' \leqslant G$, con al menos un vértice de grado 3, que maximice el número de vértices con grado 0 o 3.

Sea $C*$ la solución óptima del problema para una instancia dada y sea C una solución hayada por un algoritmo de aproximación. Se dice que el algoritmo es una $p(n) - aproximacin$ si :

$$max(\frac{C}{C*}, \frac{C^*}{C}) \leqslant p(n)$$

En el caso de un problema de maximizar es suficiente con $\frac{C*}{C} \leqslant p(n)$

Sea p_1 la función de costo para el problema de optimización. Construyamos una solución aproximada del problema. Notemos que en este problema para que haya al menos un vértice con grafo 3 en G', tiene que haber al menos un nodo con grado mayor o igual que 3 en G. Sea v este vértice y x_1, x_2, x_3 tres de sus adyacentes. Entonces si tomamos el subgrafo G' inducido por v, x_1, x_2, x_3 tenemos que $p_1(G') \geqslant n-3$. Por tanto si $G^* \leqslant G$ es tal que $p_1(G^*)$ es máximo, tenemos que :

$$\frac{C^*}{C} = \frac{p_1(G^*)}{p_1(G')} = \frac{p_1(G*)}{n-3} \leqslant \frac{n}{n-3} = 1 + \frac{3}{n-3}$$

Por tanto si tomamos $p(n) = 1 + 3/(n+3)$, tenemos que esta solución es una $p(n) - aproximacin$. Podemos observar qye para $n \geqslant 6$, esta es una $2 - aproximacin$. En general para todo $k > 1$, existe $N \in \mathbb{N}$ tal que para todo $n \geqslant N$, esta solución es una $k - aproximacin$ del óptimo.