Trabajo Práctico N°2
Grupo *undefined*

Integrantes

- Luciano Martin Gamberale. Padrón: 105892.
- Alejandro Binker. Padrón: 107056.
- Gabriel La Torre. Padrón: 87796.

Fecha de entrega:

Lunes 20 de octubre del 2024

Índice

Parte 1: Las tareas del servidor público.

Enunciado

En una ciudad con estructura tipo damero (cuadrícula) trabaja un servidor público muy particular. Tiene asignado un rectángulo de nxm cuadras y debe seleccionar un subconjunto de ellas para realizar un trabajo. El mismo comienza en la cuadra más al sureste. Se puede mover a otra cuadra. Únicamente puede trasladarse a alguna cuadra que se encuentre al oeste o al norte de donde se encuentra (puede saltarse una o varias en esas direcciónes). Todas las cuadras tienen una ganancia posible si determina realizar una tarea allí. Desea obtener la mayor ganancia posible. Pero existe una restricción adicional, por cada nueva cuadra que seleccione la ganancia debe ser mayor a la anterior. Sino, no lo puede realizar. Ayude al empleado a resolver el problema.

Resolución

1) Resolver el problema utilizando programación dinámica. (incluya en su solución definición del subproblema, relación de recurrencia y pseudocódigo).

Inicialmente tenemos una matriz de nxm donde cada posición representa una cuadra. Sea i,j la fila y columna de una cuadra respectivamente, tenemos que para cada cuadra hay un valor entero que representa la ganancia de trabajar en esa cuadra llamemoslo M[i][j]. Queremos maximizar la ganancia pudiendo movernos a otras cuadras para trabajar.

Tenemos 2 restricciones:

Una nos dice que al movernos a una cuadra, esa cuadra debe tener una ganancia mayor a la anterior.

La otra nos dice que únicamente podemos movernos a alguna cuadra que se encuentre al oeste o al norte de donde se encuentra con la posibilidad de saltar una o varias cuadras. En otras palabras nos dice que podemos movernos a cualquier casillero de la submatriz que se encuentra al noroeste de donde estamos parados o más formalmente a un casillero de la submatriz que va de (0,0) a (i,j)

Uniendo las dos restricciones podemos ver gráficamente un ejemplo de los movimientos posibles dado una cuadra genérica:

70	200	30	100
100	20	150	50
50	110	80	30
100	40	10	50

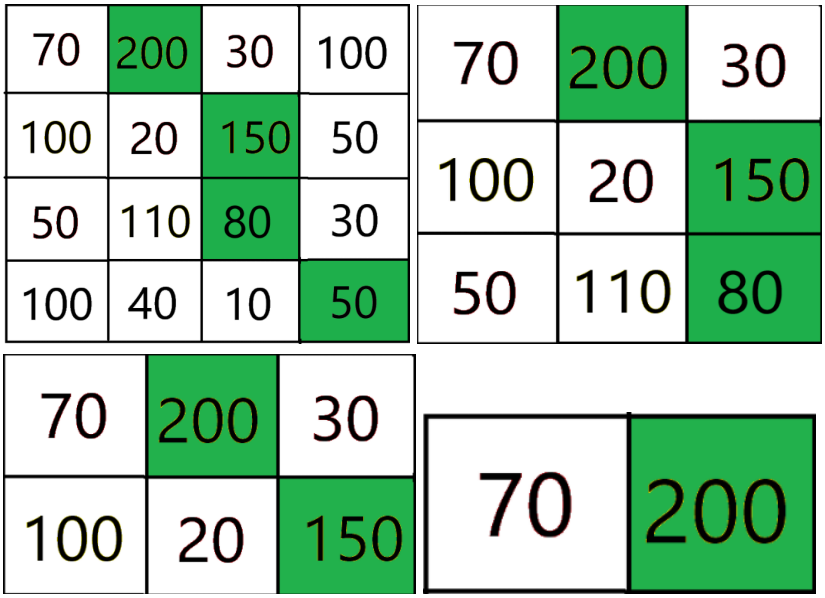
Cuadra verde: actual
Cuadras rojas: movimientos no válidos por ganancia menor
Cuadras amarillas: movimientos válidos
Cuadras grises: movimientos no válidos por estar fuera de la submatriz

En este caso, si quisiéramos calcular la máxima ganancia posible en 80 tendríamos que trabajar la actual (acumulamos 80) y ver por cada cuadra amarilla cual es la máxima ganancia que podría obtener si me moviera a esa cuadra. Vemos que la cuadra mas optima es la de 150 que tiene una ganancia máxima de 350 (150 -> 200) entonces la ganancia máxima que puedo obtener trabajando en 80 es 430 (80 -> 150 -> 200)

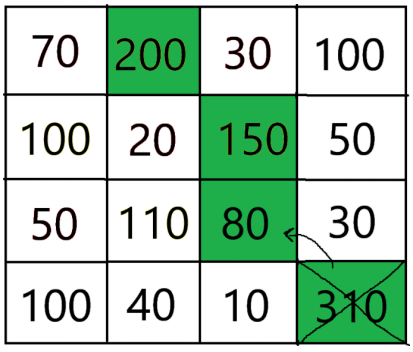
Ahora supongamos que queremos resolver la instancia de este problema (empezando en el sureste de la matriz) y estamos chequeando la máxima ganancia posible en 50, podrían ocurrir 3 cosas:

- Que 50 sea el número más alto en la matriz y sea mayor a cualquier camino posible en la submatriz al noroeste entonces la solucion seria 50
- Que haya un camino posible con mayor ganancia comenzando en una cuadra cualquiera de la submatriz al noroeste sin trabajar 50
- Que haya un camino posible con mayor ganancia que incluye trabajar 50

Podríamos recorrer todos los caminos posibles en la matriz pero no tendría sentido dado que hay problemas que incluyen calcular máximos en submatrices que ya aparecieron previamente. Gráficamente



Entonces nuestro subproblema es maximizar la ganancia en la cuadra donde estamos. De todos los elementos de la submatriz al noroeste de la calle actual nos vamos a quedar con el que tenga mayor ganancia máxima que podamos seleccionar. En caso de no poder seleccionar otra cuadra, lo mejor que podemos hacer es trabajar en la cuadra actual y finalizar.
Si no podemos saltar a ninguna cuadra desde (i,j) pero existe un camino óptimo que empieza en (k,l) con ganancia mayor que no lo incluye, entonces nuestro óptimo global no se encontrará en (i,j). Esto último se veria asi



Por lo tanto debemos quedarnos al final del algoritmo con el óptimo global siendo el mayor de todos los óptimos que calculamos.

Relación de recurrencia:
Sea el OPT(i,j) la ganancia máxima posible en la posición (i,j)
Para todo (k,l) < (i,j)

OPT(i,j) = MAX { M[i][j] + max(OPT(k,l)) si M[k][l] > M[i][j] , M[i][j]}

El resultado con la máxima ganancia posible de la ciudad será:

$Max_{i=1,\,j=1}^{n,m}$ {OPT(i,j)}

```
Sea OPT una matriz con el valor de las ganancias máximas posibles de cada cuadra
Sea recorridos una matriz con los recorridos a realizar para cada OPT
maximo_global = 0
recorrido_maximo = {}

desde i = 0 a n:
        desde j = 0 a m:
                maximo_local = matriz[i][j]
                recorrido_local= { (i,j) }

                desde k = 0 a i:
                        desde l = 0 a j:
                                si matriz[i][j] < matriz[k][l]l:
                                        valor_acumulado = OPT[k][l] + matriz[i][j]
                                        si valor_acumulado > maximo_local
                                                maximo_local = valor_acumulado
                                                recorrido_local = recorridos[k][l] U (i,j)


                OPT[i][j] = maximo_local
                recorridos[i][j] = recorrido_local
                si maximo_local >= maximo_global:
                 // Contemplamos que sea igual para priorizar los máximos más cercanos al sureste
                        maximo_global = maximo_local
                        recorrido_maximo = recorrido_local

Imprimir maximo_global
Imprimir recorrido_maximo
```

2) Analice la complejidad espacial y temporal de su propuesta

Complejidad temporal:

- Recorremos toda la matriz $O(nxm)$
- Por cada cuadra recorremos la submatriz al noroeste que en el peor de los casos será $O(nxm)$
- El resto de operaciones son $O(1)$.
- La complejidad será la multiplicación de ambas: $O((nxm)^2)$

Complejidad espacial:
- Para almacenar los óptimos necesitamos una matriz de nxm
- Para almacenar los recorridos para alcanzar cada optimo, debemos almacenar una matriz de nxm y en el peor de los casos tendremos un vector de n + m almacenado
- La complejidad espacial será la suma de ambas $O(nxm(n + m))$ + $O(nxm)$

3) De un breve ejemplo paso a paso de funcionamiento de su propuesta

Sea la matriz la siguiente:

300	100
80	90
90	120

Empezamos por la posición (0,0), como no tiene vecinos que formen la submatriz al noroeste, entonces el óptimo de (0,0) es 300 y nuestro mejor optimo es 300.

Seguimos con (0,1) y recorremos la submatriz. Como vemos que podemos seleccionar 300, entonces podemos recorrer 100 -> 300 y actualizamos nuestro máximo global con 400.

Seguimos con (1,0) con valor 80, recorriendo la submatriz vemos que solo hay un valor posible por chequear que es 300. Como es mayor podemos acumularlo y hacer 80 -> 300 con valor 380 pero como no supera nuestro máximo global no actualizamos.

Seguimos con (1,1) y podemos seleccionar 300 o 100. Como el óptimo de 100 era 400, entonces nos quedaremos con este ya que es el máximo de los posibles. Recorremos 90 -> 100 -> 300 y actualizamos nuestro máximo global con 490.

Seguimos con (2,0) y vemos que solo podemos seleccionar 300 entonces el óptimo será 390 con recorrido 90 -> 300.

Finalmente (2,1) solo podemos seleccionar 300 con una ganancia acumulada de 420 haciendo 120 -> 300. Vemos que no supera al camino con máximo global de 490 así que no lo actualizamos.
El algoritmo termina con un máximo global de 490 y un recorrido (1,1) -> (0,1) -> (0,0)

4) Programe su solución. Incluya las instrucciones de compilación y/o ejecución. Brinda 2 ejemplos para probar su programa.

Se desarrolló el programa en python. El mismo espera que en el mismo directorio esté el archivo con la matriz de ganancias para cada manzana/cuadra separados por coma. El programa se ejecuta por línea de comandos con la siguiente línea:
“python3 cuadras.py <cantidad_de_columnas> <cantidad_de_filas> <nombre_archivo_ganancias>”

5) Analice: ¿La complejidad de su propuesta es igual a la de su programa?

Para analizar la complejidad, omitimos la lectura del archivo de ganancias dado que en la propuesta no se tuvo en cuenta
- len(matriz) tiene una complejidad O(1)
- Inicializar la matriz de óptimos y recorridos tiene una complejidad de $O(nxm)$ cada uno
- Recorrer cada elemento de la matriz y por cada uno la submatriz es de $O((nxm)^2)$
- El resto de operaciones de asignación y condiciones son O(1)

- Sumando todas, la complejidad resultante es de $O((nxm)^2) + O(nxm)$. La diferencia con la complejidad de la propuesta es que en nuestro programa iteramos nxm veces para inicializar las matrices.

Parte 2: La fortaleza de la red de transporte

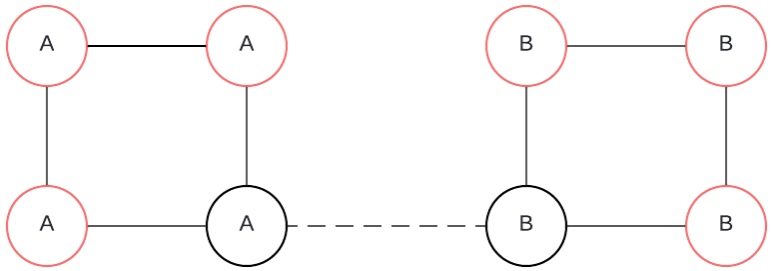
Enunciado

Contamos con la información de las rutas que unen un conjunto de "n" ciudades entre sí. Cada ruta es de doble mano. Comienza en una ciudad y finaliza en otra. Es posible utilizando una ruta o más llegar desde cualquier ciudad a otra. El ministerio de transporte quiere saber cual es el mínimo número de rutas que en caso de cortarlas por reparación provoque una desconexión entre alguna de las ciudades.

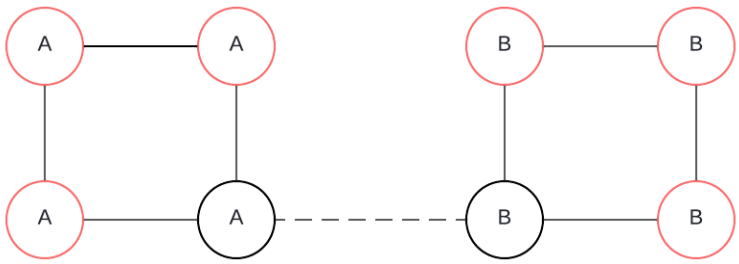
Resolución

1) Considerar la siguiente propuesta: "Aquella ciudad que cuenta con menor cantidad de rutas entrantes se debe considerar como la causante de debilidad de la red de transporte. Sus rutas adyacentes corresponden al mínimo buscado". Demostrar la optimalidad o invalidez de la afirmación.

Para demostrar que la propuesta planteada no es óptima, proponemos el siguiente ejemplo:



El ejemplo detalla un grupo de ciudades "A" unidos al grupo de ciudades "B" por una única ruta (la que se encuentra marcada en línea punteada), lo cual implicaría que el corte de la misma provoque desconexiones entre las ciudades de ambos grupos.
El resultado esperado de cualquier algoritmo sería entonces indicar que la debilidad se encuentra en la ruta de línea punteada.
Sin embargo, con la propuesta dada, indicaríamos que las ciudades que debilitan la red de transporte son las marcadas en color rojo, por tener la menor cantidad de rutas entrantes (cantidad = 2). Esto nos llevaría a pensar que tenemos que cortar al menos dos rutas para desconectar esta red de ciudades, lo cual contradice el resultado esperado.



Por lo tanto, invalidamos la propuesta.

2) Independientemente del punto anterior se solicita generar una propuesta mediante redes de flujo que solucione el problema. Explicar la idea de esta.

Para comenzar vamos a desglosar el problema

- “n” ciudades
- “m” rutas entre ciudades
- Cada ruta es doble mano
- Las ciudades están interconectadas, directamente o a través de otras (desde cualquier ciudad se puede llegar a otra)
- Nos piden el mínimo número de rutas que al cortarlas produzcan alguna desconexión

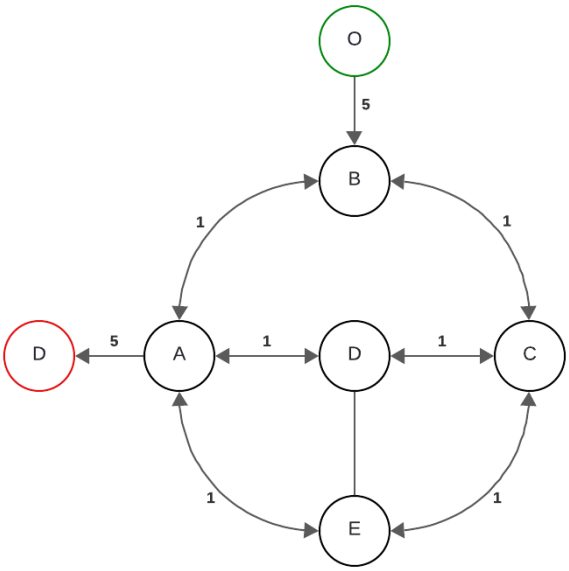
Voy a realizar la siguiente reducción para llevar el problema de las rutas a un problema de flujo mínimo:

- las “n” ciudades serán “n” nodos de un grafo
- las “m” rutas serán “2 * m” aristas en el grafo (una de ida y otra de vuelta)
- cada uno de las aristas tendrá una capacidad unitaria

A través de esto, buscaremos el corte mínimo entre dos pares de nodos/ciudades. Esto nos dará como resultado el corte mínimo entre esas dos ciudades nomás, lo cual no asegura que sea el mínimo número de rutas que separarían el el grafo. Para eso deberemos, a partir de una ciudad, calcular el camino mínimo con todas las demás. Para esto, una ciudad siempre será origen, e iremos cambiando el destino.

Agregaré un nodo más que lo llamaré “O” (origen) y otro que se llame “D” (destino). El nodo origen será nuestra fuente e irá conectado al nodo que elijamos como fijo. Y el destino será el sumidero que irá conectado al nodo que irá permutando para ir encontrando los distintos flujos mínimos. Dado que necesitamos que el eje que une a la fuente/sumidero con los nodos ruta, no afecten el flujo máximo, haremos que la capacidad de esos ejes sea igual a la cantidad de nodos (“n”), siendo que en el peor de los casos, un nodo podría tener “n-1” caminos para llegar a otro.

El criterio que vamos a escoger para elegir el nodo que estará conectado a la fuente es aquel que tenga menor identificador.
Esquematizado, para una única combinación de par de nodos, con el ejemplo brindado sería algo así:



Luego, con los mínimos de todas las ciudades, escogeremos nuevamente el mínimo de todos ellos (mínimo de mínimos), que representará la cantidad de rutas mínimas que al cortarlas producen una desconexión.

3) Presentar pseudocódigo

```
main:

    rutas las leo del archivo de rutas.txt y las guardo en una lista

    minimoNumeroRutasQueGeneranDesconexion(rutas)

//============== FUNCIÓN PRINCIPAL ==============//

fn minimoNumeroRutasQueGeneranDesconexion(rutas):

    nodos = extraerNodos(rutas)

    mapeoNodoConId = genero un diccionario con los nodos como clave y un id ascendente que empieza con valor uno
    agrego el origen con id 0
    agrego el destino con id = tamaño de nodos + 1

    idNodos = lista con todos los ids de los nodos que están en mapeoNodoConId

    aristas = generoAristasBidireccionales(rutas, mapeoNodoConId)

    flujosMaximos = lista vacía

    por cada nodo, id en mapeoNodoConId:
        si id = destino o id = origen:
            continuar a la próxima iteración
        si id = 1:
            agrego una arista desde origen hacia id con capacidad igual al tamaño de los nodos
            continuar a la próxima iteración

        agrego una arista desde id hacia destino con capacidad igual al tamaño de los nodos
        flujoMaximo = fordFulkerson(idNodos, aristas, origen, destino)
        remuevo la arista que va desde id hacia destino
```

```
        agrego flujoMaximo a flujosMaximos

    retorno el mínimo de los flujosMaximos

//============== FORD - FULKERSON ==============//

fn fordFulkerson(nodos, aristas, origen, destino):

    flujoMaximo = 0
    cantidadNodos = longitud de la lista de nodos

    flujo = inicializo una matriz de flujo de ceros de tamaño cantidadNodos x cantidadNodos

    // Crear grafos residuales:
    residualForward, residualBackward = crearGrafoResidual(cantidadNodos, aristas)

    caminoDFS = DFS(residualForward, residualBackward, origen, destino, cantidadNodos)

    mientras haya un camino desde origen a destino en el grafo residual (caminoDFS):

        cuelloBotella = calcularCuelloBotella(caminoDFS, residualForward, residualBackward, origen, destino)

        actualizo el flujo y los grafos residuales con el camino de aumento y el cuello de botella

        flujoMaximo += cuelloBotella

        caminoDFS = DFS(residualForward, residualBackward, origen, destino, cantidadNodos)

    retorno flujoMaximo

//============== CREAR GRAFO RESIDUAL ==============//

fn crearGrafoResidual(cantidadNodos, aristas):

    residualForward = matriz de ceros de tamaño cantidadNodos x cantidadNodos
    residualBackward = matriz de ceros de tamaño cantidadNodos x cantidadNodos

    para cada arista (u, v, capacidad) en aristas:
        residualForward[u][v] = capacidad
        residualBackward[v][u] = 0

    retorno residualForward, residualBackward

//============== CALCULAR CUELLO DE BOTELLA ==============//

fn calcularCuelloBotella(caminoDFS, residualForward, residualBackward, origen, destino):

    cuelloBotella = infinito
    nodoActual = destino

    mientras nodoActual != origen:
        nodoAnterior, tipoArista = caminoDFS[nodoActual]

        si la arista es hacia adelante (ARISTA_FORWARD):
            cuelloBotella = mínimo entre cuelloBotella y residualForward[nodoAnterior][nodoActual]
        si la arista es hacia atrás (ARISTA_BACKWARD):
            cuelloBotella = mínimo entre cuelloBotella y residualBackward[nodoActual][nodoAnterior]

        nodoActual = nodoAnterior

    retorno cuelloBotella

//============== ACTUALIZAR FLUJO ==============//
```

```
fn actualizarFlujo(flujo, residualForward, residualBackward, caminoDFS, cuelloBotella, origen, destino):

    nodoActual = destino

    mientras nodoActual != origen:
        nodoAnterior, tipoArista = caminoDFS[nodoActual]

        si la arista es hacia adelante (ARISTA_FORWARD):
            flujo[nodoAnterior][nodoActual] += cuelloBotella
            residualForward[nodoAnterior][nodoActual] -= cuelloBotella
            residualBackward[nodoActual][nodoAnterior] += cuelloBotella

        si la arista es hacia atrás (ARISTA_BACKWARD):
            flujo[nodoActual][nodoAnterior] -= cuelloBotella
            residualForward[nodoActual][nodoAnterior] += cuelloBotella
            residualBackward[nodoAnterior][nodoActual] -= cuelloBotella

        nodoActual = nodoAnterior

//============== DFS ==============//

fn DFS(residualForward, residualBackward, origen, destino, cantidadNodos):

    nodosVisitados = lista de pares (NO_VISITADO, ARISTA_NO_VISITADA) de longitud cantidadNodos
    marco el origen como visitado (origen, ARISTA_ORIGEN)

    retorno DFSRecursivo(residualForward, residualBackward, origen, destino, cantidadNodos, nodosVisitados)

fn DFSRecursivo(residualForward, residualBackward, nodoActual, destino, cantidadNodos, nodosVisitados):

    si nodoActual es igual a destino:
        retorno nodosVisitados

    para cada nodo i en la lista de nodos:
        si el nodo i no ha sido visitado y hay capacidad hacia adelante desde nodoActual a i:
            marco el nodo i como visitado desde nodoActual por arista hacia adelante
            si un camino al destino se encuentra desde el nodo i:
                retorno el camino
            si no, desmarco el nodo i como no visitado

        si el nodo i no ha sido visitado y hay capacidad hacia atrás desde nodoActual a i:
            marco el nodo i como visitado desde nodoActual por arista hacia atrás
            si un camino al destino se encuentra desde el nodo i:
                retorno el camino
            si no, desmarco el nodo i como no visitado

    retorno nodosVisitados
```

4) Realizar un análisis de optimalidad.

Para demostrar la optimalidad de nuestra solución, partimos del hecho de que hemos convertido el problema de rutas bidireccionales en un problema de corte mínimo en un grafo dirigido. Cada arista bidireccional será convertida en dos aristas dirigidas. Este problema lo resolvemos aplicando el algoritmo de Ford-Fulkerson para hallar el flujo máximo entre nodos.

Dado que queremos encontrar la cantidad mínima de aristas a cortar para generar una desconexión, buscamos el flujo mínimo que conecte un nodo con todos los demás nodos. Sin embargo, podemos reducir la cantidad de pares de nodos para los que calculamos Ford-Fulkerson, ya que:

Dado que queremos encontrar la cantidad mínima de aristas a cortar para generar una desconexión, buscamos el flujo mínimo que conecte un nodo con todos los demás nodos. Sin embargo, podemos reducir la cantidad de pares de nodos para los que calculamos Ford-Fulkerson, ya que:
- *Simetría de las aristas bidireccionales:* dado que todas las aristas son bidireccionales (lo que implica que son equivalentes en ambas direcciones), el flujo máximo entre un par de nodos es simétrico. Por lo tanto, el flujo máximo entre un par de nodos $a \to b,\ a, b \in G$ es el mismo que entre $b \to a$. Esto nos permite calcular cada par de nodos una única vez, optimizando el cálculo.
- *Prueba de optimalidad:* Supongamos, por contradicción, que existe un par de nodos $b, c \in G$ cuyo flujo es menor que el mínimo que encontramos entre un nodo $a \in G$ y todos los demás nodos. Esto implicaría que el flujo entre $b \to c$ es independiente de los caminos que pasan por a, lo cual es imposible dado que todos los nodos están interconectados por construcción del grafo.

Demostración de "*Prueba de optimalidad*":

Hipótesis: supongo que hay otro par de nodos $b, c \in G$, cuya desconexión requiere un número menor de aristas que la desconexión de cualquier nodo $a \in G$ con todos los demás nodos.

Absurdo: como dato tenemos que si o si, $a, b, c \in G$ están interconectados (porque las rutas tienen que tener un camino entre ellas por enunciado). Si seguimos la hipótesis previamente mencionada, significa que el flujo $b \leftrightarrow c$ sería menor que el flujo entre a y cualquier nodo en el grafo. Sin embargo, dado que todas las rutas entre nodos están interconectadas por enunciado, cualquier camino $b \leftrightarrow c$ puede descomponerse en caminos que pasan por otros nodos, incluido a. Esto significa que el corte mínimo entre $b \leftrightarrow c$ está limitado por los cortes mínimos entre a y los demás nodos. Por lo tanto, no puede haber un corte más pequeño entre $b \leftrightarrow c$ que el que ya se encontró entre a y otro nodo.

Absurdo: Otra manera de verlo es suponiendo que el mínimo número de aristas a cortar para desconectar el grafo no está en el camino que conecta a con el resto de los nodos de G, sino que está en un camino entre otros dos nodos, digamos $b \leftrightarrow c$. Esto significa que la capacidad mínima (número de aristas a cortar) en el camino $C(b \leftrightarrow c)$ es menor que la capacidad mínima en cualquier camino que conecta a con otros nodos, es decir: $C(b \leftrightarrow c) < C(a \leftrightarrow c)$

Esto implica que el camino $b \to c$, es decir, $C(b \leftrightarrow c)$ lo podemos dividir en:
- $\subseteq a$: esto se va a ver limitado por el mínimo entre $C(a \leftrightarrow c, \not\subseteq b)$ y $C(a \leftrightarrow b, \not\subseteq c)$
- $\not\subseteq a$: $C(b \leftrightarrow c, \not\subseteq a)$
También puedo hacer algo similar con $C(a \leftrightarrow c)$:
- $\subseteq b$: esto se va a ver limitado por el mínimo entre $C(a \leftrightarrow b, \not\subseteq c)$ y $C(b \leftrightarrow c, \not\subseteq a)$
- $\not\subseteq b$: $C(a \leftrightarrow c, \not\subseteq b)$

Por *hipótesis* tenemos que:
$C(b \leftrightarrow c) < C(a \leftrightarrow c)$
$min(C(a \leftrightarrow c, \not\subseteq b), C(a \leftrightarrow b, \not\subseteq c)) + C(b \leftrightarrow c, \not\subseteq a) < min(C(a \leftrightarrow b, \not\subseteq c), C(b \leftrightarrow c, \not\subseteq a)) + C(a \leftrightarrow c, \not\subseteq b)$

veo si siempre se cumple la condición:
$C(b \leftrightarrow c, \not\subseteq a) < C(a \leftrightarrow c, \not\subseteq b) < C(a \leftrightarrow b, \not\subseteq c)$
$C(a \leftrightarrow c, \not\subseteq b) + C(b \leftrightarrow c, \not\subseteq a) < C(b \leftrightarrow c, \not\subseteq a) + C(a \leftrightarrow c, \not\subseteq b)$
Esto es un absurdo porque la suma de las mismas dos cantidades en lados opuestos de la desigualdad debería ser igual, no menor.

La contradicción implica que no puede haber camino $C(b \leftrightarrow c)$ con un corte mínimo menor que el camino $C(a \leftrightarrow c)$. Por lo tanto, el camino mínimo se encuentra necesariamente entre a y el resto de los nodos, como habíamos afirmado al principio.

Luego, este número mínimo sería al que hay que “transformarlo” nuevamente en el problema original, pero esto es simplemente mantenerlo igual.

5) Realizar análisis de complejidad temporal y espacial. Considere las estructuras de datos que utiliza para llegar a estos.

Datos:
- "n" ciudades → "n" nodos
- "m" rutas → "$2 \times m$" aristas

Complejidades temporales:
- Lectura del archivo: $O(m)$, dado que se leen todas las rutas.
- Extracción de nodos: $O(m)$ para recorrer la lista de rutas y $O(n)$ para insertar con búsqueda lineal sin repeticiones, lo cual daría un complejidad de $O(m \times n)$.
- Generación de ids para nodos: $O(n)$.
- Generación de aristas: $O(m)$, dado que recorro todas las rutas.
- Cálculo de flujos máximos: $O(n \times F)$, dado que se ejecuta n veces Ford-Fulkerson $O(F)$.
- Búsqueda del mínimo flujo máximo: $O(n)$.
- Ford-Fulkerson $O(F)$:
    - Inicializar la matriz de flujo: $O(n^2)$.
    - Crear grafo residual: $O(n^2)$.
    - DFS: en el peor de los casos recorre n veces los n nodos para encontrar el camino de aumento, por lo tanto es $O(n^2)$.
    - Cálculo de cuello de botella: en el peor de los casos se recorren las m aristas para ver cuál es el mínimo, por lo tanto $O(m)$.
    - Actualizar flujo: nuevamente por cada arista, actualizo los flujos, por lo tanto $O(m)$.
    - Ejecución de búsqueda de caminos/actualización de flujos: podemos decir que la máxima cantidad de veces que se puede encontrar un camino de aumento, está limitado por un corte que deje por un lado la fuente y por el otro el sumidero. Esto se podría hacer aislando simplemente el nodo fuente, que tendrá una única arista de salida con capacidad máxima n, dado que en el peor de los casos, el flujo máximo podría ser de $n - 1$. Por lo tanto, esto se ejecutará como máximo n veces, en donde por cada iteración tendremos adentro una complejidad de $O(n^2 + m)$, lo que dá como resultado $O(n^3 + n \times m)$
    - Por lo tanto $O(F) = O(n^3 + n \times m)$
- Por lo tanto, la complejidad temporal final es $O(n \times F) = O(n^4 + n^2 \times m)$.

Complejidad espacial:
- Rutas/Aristas: se guardan en una lista, $O(m)$.
- Ciudades/Nodos: se guardan en una lista, $O(n)$.
- Mapeo de nodos con id: se guarda en un diccionario, $O(n)$.
- Flujos máximos: se guardan en una lista, $O(n)$.
- Ford-Fulkerson:
    - Flujos: se guarda cada flujo entre nodos en una matriz, $O(n^2)$.
    - Grafo residual para adelante: se guarda cada capacidad disponible entre nodos en una matriz, $O(n^2)$.
    - Grafo residual para atrás: se guarda cada capacidad disponible entre nodos en una matriz, $O(n^2)$.
    - Resultado de DFS: lista con los nodos y referencias para ir construyendo el camino, $O(n)$.
- Por lo tanto, la complejidad espacial es $O(n^2 + m)$.

6) Programar la solución. Incluya la información necesaria para su ejecución. Compare la complejidad de su algoritmo con la del programa.

El programa fué desarrollado en Python. El mismo espera que en el directorio en el cual posicione el archivo "transporte.py", se encuentre un archivo con la información de las rutas. El archivo de rutas se espera que tenga el formato acordado por el enunciado. La manera de ejecutarlo es por línea de comando, con las siguientes especificaciones:
"python3 transporte.py <nombre_archivo_rutas>"

Algunas complejidades temporales utilizadas en el código:

- crear set( ): $O(1)$.
- agregar elemento a set, set.add(<x>): $O(1)$.
- crear list(<set>): $O(n)$ considerando que el set tiene n elementos.
- agregar elemento a lista, list.append(<x>): $O(1)$.
- acceder elemento en lista, list[<x>]: $O(1)$.
- tamaño de lista, len(<list>): $O(1)$.
- insertar elemento en diccionario, dict[<x>] = <y>: $O(n)$ considerando que el diccionario tiene n elementos.
- acceder elemento en diccionario, dict[<x>]: $O(n)$ considerando que el diccionario tiene n elementos.
- extraer valores de diccionario, dict.values( ): $O(n)$ considerando que el diccionario tiene n elementos.

Considerando las mismas en el desarrollo que habíamos realizado para el pseudo-código, quedaría algo de la siguiente manera:
Complejidades temporales:

- Lectura del archivo: $O(m)$, dado que se leen todas las rutas y el append es $O(1)$.
- Extracción de nodos: $O(m)$ para recorrer la lista de rutas y $O(n)$ para insertar en el set, lo cual daría un complejidad de $O(m + n)$.
- Generación de ids para nodos: $O(n)$ para recorrer los nodos y $O(n)$ para insertarlos dentro del diccionario, por lo tanto es $O(n^2)$.
- Obtención de id de nodos: $O(n)$.
- Generación de aristas: $O(m)$, dado que recorro todas las rutas.
- Cálculo de flujos máximos: $O(n \times F)$, dado que se ejecuta n veces Ford-Fulkerson $O(F)$.
- Búsqueda del mínimo flujo máximo: $O(n)$.
- Ford-Fulkerson $O(F)$:
    - Inicializar la matriz de flujo: $O(n^2)$.
    - Crear grafo residual: $O(n^2)$.
    - DFS: en el peor de los casos recorre n veces los n nodos para encontrar el camino de aumento, por lo tanto es $O(n^2)$.
    - Cálculo de cuello de botella: en el peor de los casos se recorren las m aristas para ver cuál es el mínimo, por lo tanto $O(m)$.
    - Actualizar flujo: nuevamente por cada arista, actualizo los flujos, por lo tanto $O(m)$.
    - Ejecución de búsqueda de caminos/actualización de flujos: podemos decir que la máxima cantidad de veces que se puede encontrar un camino de aumento, está limitado por un corte que deje por un lado la fuente y por el otro el sumidero. Esto se podría hacer aislando simplemente el nodo fuente, que tendrá una única arista de salida con capacidad máxima n, dado que en el peor de los casos, el flujo máximo podría ser de $n - 1$. Por lo tanto, esto se ejecutará como máximo n veces, en donde por cada iteración tendremos adentro una complejidad de $O(n^2 + m)$, lo que dá como resultado $O(n^3 + n \times m)$

    - Para el caso de Ford-Fulkerson, no hubo diferencias con respecto al pseudo codigo, dado que se utilizaron listas que contenían la misma complejidad algorítmicas para crearlas, accederlas y modificarlas.
    - Por lo tanto $O(F) = O(n^3 + n \times m)$
- Por lo tanto, la complejidad temporal final es $O(n \times F) = O(n^4 + n^2 \times m)$.

En cuanto a la complejidad espacial, es la misma que calculamos previamente, dado que coincide con lo indicado en la documentación.

7) ¿Es posible expresar su solución como una reducción polinomial? En caso afirmativo explique cómo y en caso negativo justifique su respuesta.

Si, de hecho para encarar el problema, fuimos realizando las siguientes reducciones:
- El problema consta de:
    - "n" *ciudades*, en donde todas se encuentran interconectadas.
    - "m" *rutas* entre ciudades, en donde cada ruta es doble mano.
- Nos piden el *mínimo número* de *rutas* que al cortarlas produzcan alguna *desconexión*.

A partir de esto, realizamos las siguientes *reducciones polinomiales*:
- las "n" *ciudades* serán "n" *nodos* de un grafo
- las "m" *rutas* serán "2 * m" *aristas* en el grafo (uno de ida y otro de vuelta)
- cada uno de las aristas tendrá una capacidad unitaria

Las reducciones son polinomiales, dado que dependen polinomicamente de la cantidad de ciudades y rutas de nuestro problema.

Ahora debemos resolver en el grafo resultante, cuál es la cantidad de aristas mínimas que deberíamos cortar para producir una desconexión. Para eso recurrimos a Ford-Fulkerson, en donde sabemos que a partir de dos nodos, podemos encontrar cuál es el flujo máximo, y con esto saber cuál es la cantidad mínima de nodos que habría que cortar para desconectar esos dos nodos.
Sin embargo, dado que nos piden una desconexión cualquiera en el grafo, como explicamos anteriormente, debemos aplicar Ford-Fulkerson entre un nodos vs. todos los demás.
Esto hace que nuestra complejidad sea, a lo sumo, "n" veces Ford-Fulkerson.

Parte 3: Un casting para el reality show

Nota: la re-entrega está en la sección siguiente.

Enunciado

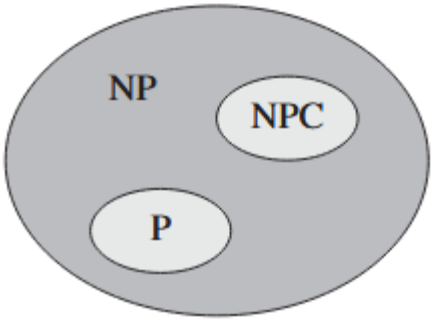
Contamos con un conjunto de "n" personas que conforman un grupo de un próximo reality show de supervivencia extrema. Algunas de esas personas se conocen entre sí y tienen una relación de amistad preexistente. Se desea separar a las personas en dos equipos con la condición que los que tienen amistad queden siempre en el mismo equipo. Para lograrlo se nos permite eliminar con mucho "j" personas que puedan resultar conflictivas para cumplir con el cometido. La producción del programa desea saber si dado un casting determinado es posible lograr los solicitado.

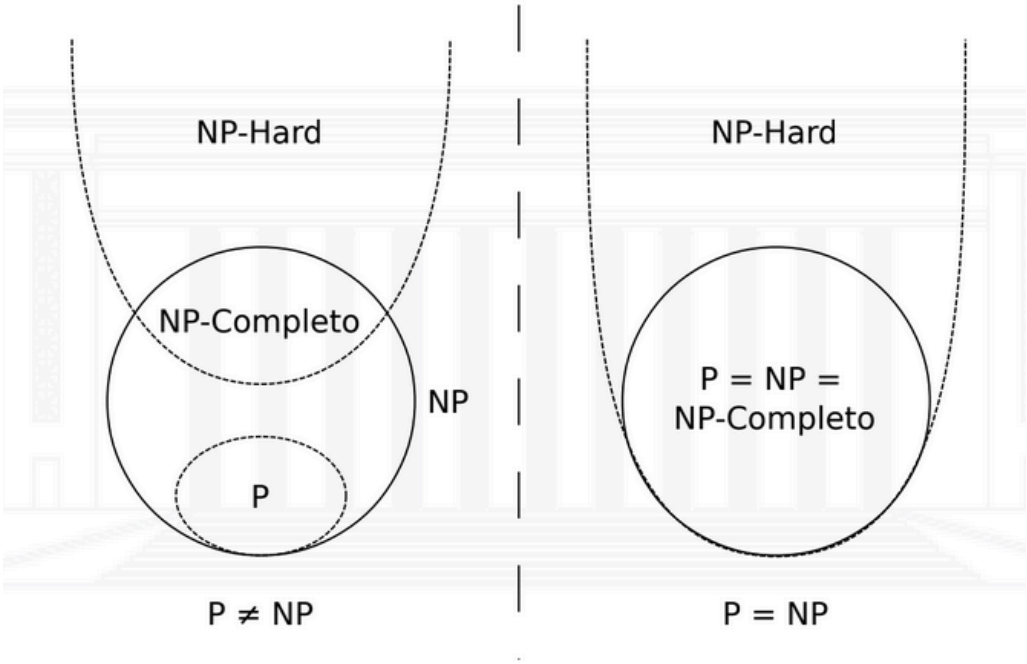
Resolución

1) Realice un análisis teórico entre las clases de complejidad P, NP, NP-H y NP-C y la relación entre ellos.

La clase P es un conjunto de problemas que pueden resolverse en tiempo polinomial de manera determinista. Mientras que, los problemas NP son problemas que no necesariamente pueden

resolverse en tiempo polinomial pero, dada una solución, pueden verificarse en tiempo polinomial. Esto incluye, entonces, a los problemas de clase P. Los problemas NP-Completo, son problemas que son tan difíciles de resolver como de verificar.
Un problema es NP-C si:
a) el problema pertenece a NP y
b) todos los problemas en PC se pueden reducir a él en tiempo polinomial.
Un problema es NP-Hard si b se cumple pero a, no.





2) Demostrar que, dada una posible solución que obtenemos, se puede fácilmente determinar si se puede cumplir o no con la tarea solicitada.

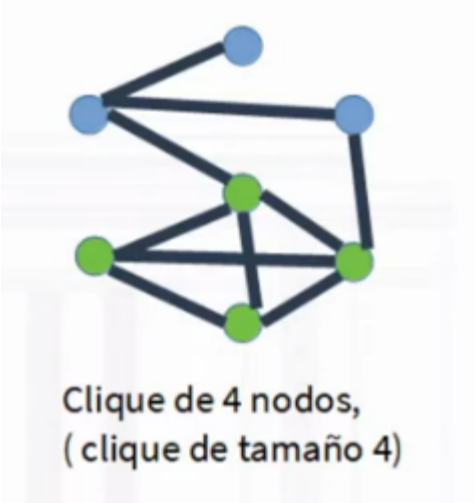
Dada una partición de 2 equipos. Representamos cada miembro como un nodo con sus amistades como aristas. Toma n iteraciones, donde n es el tamaño de la suma de los miembros de ambos equipos, determinar si alguno de los nodos tiene un amigo en otro equipo. Esta verificación es polinomial.

3) Demostrar que si desconocemos la solución la misma es difícil de resolver. Utilizar para eso el problema “Mínimum Node Deletion bipartite Subgraph” (suponiendo que sabemos que este es NP-C).

El “Minimum Node Deletion Bipartite Subgraph”, o su traducción “Subgrafo Bipartito por Eliminación Mínima de Nodos”, es un problema por el cual se busca, en un grafo no dirigido, es decir, cuyas aristas no tienen dirección, quitar la menor cantidad de nodos, de manera que el resultado sea un subgrafo bipartito. Entiéndase por un subgrafo bipartito, un grafo que puede dividirse en dos conjuntos disjunto cuyos vértices no tengan aristas entre los vértices de un mismo conjunto. Esto es análogo al problema del “Casting para el reality show”. Sea G=(V,E) un grafo cuyas aristas representan las amistades. Para resolver el problema del “Casting” de manera análoga al “Minimum Node Deletion Bipartite Subgraph”, construimos G’=(V,E) de manera que las aristas del grafo ahora representan quienes no tienen amistad. Por transitividad: Si sabemos que “Minimum Node Deletion Bipartite Subgraph”, a partir de ahora X, es NP-C y “el problema del casting”, a partir de ahora Y, puede transformarse en X. Y -> trans -> X -> sol(x) -> trans -> sol(y) => Y ≤ X Por lo que entonces “Minimum Node Deletion Bipartite Subgraph” es al menos NP-C.

4) Demostrar que el problema “Mínimum Node Deletion bipartite Subgraph” pertenece a NP-C. (Para la demostración puede ayudarse con diferentes problemas, recomendamos “Clique problem”)

Un Clique es un problema de decisión NP-C. El problema consiste en, dado un k mayor a 0, obtener un subgrupo del grafo de tamaño k en el que todos los vértices están conectados entre sí.
Por ejemplo, el visto en clase:



Mientras que, como explicamos anteriormente, el “Minimum Node Deletion Bipartite Subgraph” es un problema en el que, dado un k, queremos saber si se puede dividir el grafo en grupos cuyos vértices no **tengan conexiones** entre sí.

Para probar NP-C, primero, probemos que “Minimum Node Deletion Bipartite Subgraph” **ε** NP.

Dado G=(V,E) grafo
j la cantidad de integrantes del casting a descartar.
T certificado: 2 subconjuntos de nodos de V.
Puedo verificar en tiempo polinomial:

- La cantidad de nodos de T |T|= |V|-j.
- Cada nodo de T está conectado solo con nodos de su subconjunto.

Visto que podemos verificar una solución del problema en tiempo polinomial O(|T|) -> podemos afirmar que es NP.
Ahora falta demostrar que es NP-Completo:
Sabemos que el problema del Clique es NP-Hard, para demostrar que Minimum Node Deletion Bipartite Subgraph es NP-Completo tenemos que reducir polinomicamente desde el problema del clique a Minimum Node Deletion Bipartite Subgraph.
Dado G=(V,E) un grafo.
Si el grafo tiene un clique de tamaño k, entonces cualquier subconjunto de vértices de tamaño mayor o igual a k no puede ser parte de un grafo bipartito.
Sea J el número de nodos a eliminar. Tenemos que J = |V| - k.
En definitiva, cualquier problema de Clique con un valor k, se puede transformar en un problema de Minimum Node Deletion Bipartite Subgraph de valor J tal que J = |V| - k. Siendo esta una transformación en tiempo polinomial.

5) En base a los puntos anteriores a qué clases de complejidad pertenece el problema del "Casting del reality"? Justificar

Por lo explicado en el punto 3, podemos afirmar que el problema del casting es NP-C ya que es una transformación del problema de "Minimum Node Deletion Bipartite Subgraph".

6) Una persona afirma tener un método eficiente para responder el pedido cualquiera sea la instancia. Utilizando el concepto de transitividad y la definición de NP-C explique qué ocurriría si se demuestra que la afirmación es correcta.

Si se demuestra que el método eficiente de esta persona resuelve el problema del casting del reality en tiempo polinomial, y sabemos que este problema es NP-completo, entonces, según la transitividad, todos los problemas en NP podrían resolverse en tiempo polinomial.
Esto implicaría que P = NP, es decir, que la clase de problemas que se pueden resolver en tiempo polinomial es la misma que la clase de problemas cuyas soluciones se pueden verificar en tiempo polinomial.

7) Un tercer problema al que llamaremos X se puede reducir polinomialmente al problema de "Casting del reality", qué podemos decir acerca de su complejidad?

Al acotar polinomialmente un problema al problema del "Casting del reality" podemos decir que ese problema es al menos tan difícil como el "Casting del reality" por lo que es, al menos de clase NP-C.

Referencias

- Introduction to Algorithms, third edition. Cormen, Leiserson, Rivest, Stein. The MIT Press. Capítulo 34.
- Clase de Reducciones polinomiales, Teoría de Algoritmos, Ing. Víctor Daniel Podberezski.
- Documentación sobre la complejidad de las operaciones de una lista en Python

Parte 3 (Primera re-entrega): Un casting para el reality show

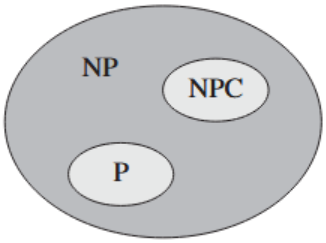
Enunciado

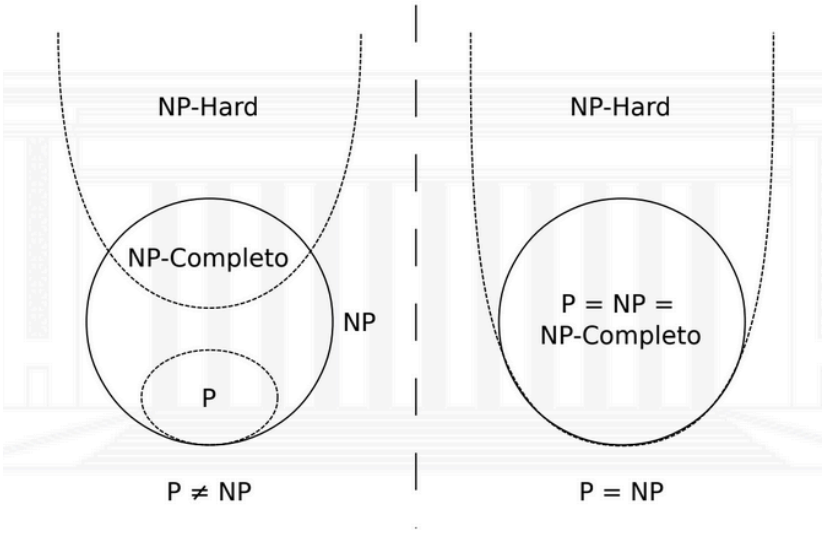
Contamos con un conjunto de “n” personas que conforman un grupo de un próximo reality show de supervivencia extrema. Algunas de esas personas se conocen entre sí y tienen una relación de amistad preexistente. Se desea separar a las personas en dos equipos con la condición que los que tienen amistad queden siempre en el mismo equipo. Para lograrlo se nos permite eliminar con mucho “j” personas que puedan resultar conflictivas para cumplir con el cometido. La producción del programa desea saber si dado un casting determinado es posible lograr los solicitado.

Resolución

1) Realice un análisis teórico entre las clases de complejidad P, NP, NP-H y NP-C y la relación entre ellos.

La clase P es un conjunto de problemas que pueden resolverse en tiempo polinomial de manera determinista. Mientras que, los problemas NP son problemas que no necesariamente pueden resolverse en tiempo polinomial pero, dada una solución, pueden verificarse en tiempo polinomial. Esto incluye, entonces, a los problemas de clase P. Los problemas NP-Completo cumplen
a) el problema pertenece a NP, es decir, puede verificarse en tiempo polinomial y
b) todos los problemas en NP se pueden reducir a él en tiempo polinomial.
Un problema es NP-Hard si b se cumple pero a, no.





2) Demostrar que, dada una posible solución que obtenemos, se puede fácilmente determinar si se puede cumplir o no con la tarea solicitada.

Dada una partición de 2 equipos. De cada miembro del equipo tenemos sus amigos.

El pseudocódigo para el certificador sería:

```
Sea O el conjunto original de miembros del casting
Sea A, B, C dos equipos y un conjunto de eliminados propuestos como solución
```

```
Si len(C) ≥ j  # j es el valor máximo que nos permitían eliminar
        Devolver Falso
Si len(A) + len(B) + len(C) != len(O)
        Devolver Falso
Por cada miembro de C:
        Si no pertenece_al_conjunto_original(miembro)
                Devolver Falso
Por cada miembro del equipo A:
        si no pertenece_al_conjunto_original(miembro) o tiene_amigos_en(miembro, B):
                Devolver Falso
Por cada miembro del equipo B:
        si no pertenece_al_conjunto_original(miembro) o si tiene_amigos_en(miembro, A):
                Devolver Falso
Devolver Verdadero  # Verificamos la solución.

def tiene_amigos_en(miembro, equipo):
        amigos = miembro.amigos
        por cada amigo en amigos:
                si amigo en equipo
                        devolver True
        False

def pertenece_al_conjunto_original(miembro):
        si miembro en O
                devolver True
        False
```

Toma n iteraciones, donde n es el tamaño de la suma de los miembros de ambos equipos, recorrer a todos los miembros. Luego, por cada miembro son, como máximo, “n” iteraciones para revisar cada amigo del miembro para verificar que no está en el equipo contrario.
Esto sería de $O(n^2)$ por lo cual podemos afirmar que el certificador es polinomial, y con esto afirmar que este problema pertenece a NP, siendo que es un problema de decisión y además tiene un certificador polinomial.

3) Demostrar que si desconocemos la solución la misma es difícil de resolver. Utilizar para eso el problema “Mínimum Node Deletion bipartite Subgraph” (suponiendo que sabemos que este es NP-C).

El “Minimum Node Deletion Bipartite Subgraph”, o su traducción “Subgrafo Bipartito por Eliminación Mínima de Nodos”, es un problema por el cual se busca:

Dado un grafo no dirigido $G = (V, E)$, el objetivo es determinar el **mínimo número de nodos** que deben eliminarse del grafo para que el subgrafo restante sea **bipartito**.

Algunos conceptos claves son:

Grafo bipartito:
Un grafo es bipartito si sus nodos pueden dividirse en dos conjuntos U y W tal que no haya aristas entre nodos del mismo conjunto (es decir, todas las aristas conectan un nodo de U con uno de W).

Eliminar nodos:
Se busca eliminar el menor número posible de nodos de G para garantizar que el subgrafo resultante sea bipartito.

Dado que lo que buscaremos es demostrar que el problema de “Un casting para el reality show” es NP-Hard, para demostrar que es “difícil de resolver”, propongo reducir el problema de “Minimum Node Deletion Bipartite Subgraph” (X) al problema de “Un casting para el reality show” (Y):

$X \leq_p Y$

Para esto, re-escribiremos un poco el problema de “Minimum Node Deletion Bipartite Subgraph” para que sea un problema de decisión:

Dado un grafo no dirigido G=(V,E), ¿Se pueden eliminar como máximo j nodos para que el subgrafo restante sea bipartito?

Ahora lo que resta, es buscar una reducción polinomial para cumplir con lo siguiente:
$X \rightarrow transf.\ polinomial \rightarrow Y \rightarrow Caja\ negra\ que\ resuelve(Y) \rightarrow S_y \rightarrow transf.\ polinomial \rightarrow S_x$

Partimos de:
$G = (V, E)$: grafo no dirigido
j: máxima cantidad de nodos a eliminar

Una posible reducción podría ser que cada vértice del grafo original se convierta en una persona. Luego, con los ejes del grafo, lo que haremos es convertir cada eje entre dos nodos, en una relación de amistad entre personas.
Por último, el valor j de máxima cantidad de nodos a eliminar, se convertirá en el j de máxima cantidad de personas a eliminar.

Planteo el pseudocódigo de la transformación para demostrar que es polinomial:

```
G = (V, E)
j

P: personas
R: relaciones entre personas
j' = j

por cada v de V:
        agrego una persona a P

por cada (u, v) de E:
        agrego una relación a R entre Persona u y Persona v
```

Como vemos la complejidad temporal es $O(V + E)$ debido a que se tienen que iterar los nodos y aristas por separados para convertirlos en personas y relaciones. Además, la complejidad espacial es de $O(V + E)$ también, debido a que las personas tendrán una complejidad espacial $O(V)$ y las relaciones de $O(E)$.

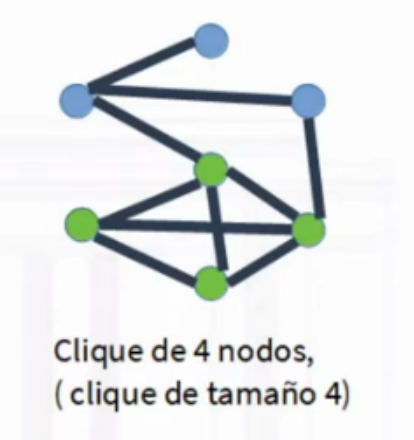
Luego, estas personas y relaciones, se las envío al problema de “Un casting para el reality show” que me va a indicar si efectivamente se pueden eliminar como mucho “j” personas. Este resultado, también debe transformado en el resultado del “Minimum Node Deletion Bipartite Subgraph”, pero esta transformación es $O(1)$ debido a que se debe devolver el mismo resultado, sin realizar modificaciones.

Con todo lo mencionado anteriormente, podemos afirmar que se puede reducir un problema NP-Hard a del casting, demostrando que el problema del casting es tan complejo como un problema NP-Hard.

4) Demostrar que el problema “Mínimum Node Deletion bipartite Subgraph” pertenece a NP-C. (Para la demostración puede ayudarse con diferentes problemas, recomendamos “Clique problem”)

Un Clique es un problema de decisión NP-C. El problema consiste en, dado un k > 0, obtener un subgrupo del grafo de tamaño k en el que todos los vértices están conectados entre sí.

Por ejemplo, el visto en clase:



Mientras que, como explicamos anteriormente, el “Minimum Node Deletion Bipartite Subgraph” es un problema en el que, dado un k, queremos saber si se puede dividir el grafo en 2 grupos cuyos vértices **no tengan conexiones** entre sí.
Para probar NP-C, primero, probemos que “Minimum Node Deletion Bipartite Subgraph” ∈ NP:

El pseudocódigo de esto sería:

```
fn certificador(G, j, U, W, T):
        G=(V, E): matriz de adyacencias
        j:cantidad máxima de nodos a eliminar

        Sea U, W conjuntos bipartitos
        Sea T un conjunto de nodos eliminados

        si tamaño(V) != tamaño(U) + tamaño(W) + tamaño(T):
                retorno Falso

        si tamaño(T) > j:
                retorno Falso

        por cada u de U:
                si u no pertenece a V
                        retorno Falso

        por cada w de W:
                si w no pertenece a V
                        retorno Falso

        por cada t de T:
                si t no pertenece a V
                        retorno Falso

        por cada u de U:
```

```
                por cada v de G[u]:
                        si v pertenece a U:
                                retorno Falso

        por cada w de W:
                por cada v de G[w]:
                        si v pertenece a W:
                                retorno Falso

        retorno Verdadero
```

Complejidad temporal: $O(V^3)$. Visto que podemos verificar una solución del problema en tiempo polinomial, **podemos afirmar que es NP**.

Ahora falta demostrar que es NP-Hard:

Propiedad de los Cliques y Bipartición
Un clique de tamaño k (subgrafo completo con k nodos) nunca puede ser bipartito si $k > 2$. Esto se debe a que:

- En un grafo bipartito, no puede haber ciclos impares.
- Un clique tiene todos los nodos conectados entre sí, y por lo tanto contiene ciclos impares para $k > 2$.

Para hacer un clique bipartito, debes reducirlo lo máximo posible, es decir, dejar solo dos nodos conectados. Por lo tanto, en un clique de tamaño k:

Se necesitan eliminar $j = k - 2$ nodos para que el subgrafo resultante sea bipartito.

En un grafo $G = (V, E)$ con cliques, para garantizar la bipartición después de eliminar nodos:
Por cada clique de tamaño k en G, debes eliminar al menos $k - 2$ nodos de ese clique para convertirlo en bipartito.
Esto significa que el número total de nodos a eliminar para bipartir todo el grafo dependerá del tamaño del mayor clique presente en G.

Sabemos entonces que SIEMPRE si tenemos un clique de tamaño k, al menos hay que eliminar k-2 nodos para poder generar una bipartición.

Esta es la transformación propuesta:

```
Sea G' = el grafo G de la instancia de Minimum Node Deletion Bipartite Subgraph
Sea j = k - 2

// Si el resultado del algoritmo que resuelve Minimum Node Deletion Bipartite Subgraph es verdadero
entonces hay un clique de tamaño k o mayor
```

Dado que pudimos reducir polinomialmente el problema de "Clique" al problema de "Mínimum Node Deletion bipartite Subgraph", podemos afirmar que "Mínimum Node Deletion bipartite Subgraph" pertenece a NP-Hard.

Y dado que pertenece a NP y además a NP-Hard, podemos afirmar que pertenece a NP-C.

5) En base a los puntos anteriores a qué clases de complejidad pertenece el problema del "Casting del reality"? Justificar

Debido a que en el punto 2, demostramos que el problema "Casting del reality" pertenece a NP porque podemos certificar la solución en tiempo polinomial y es un problema de decisión, y que además por el punto 3, demostramos que pertenece a NP-Hard, al haber reducido un problema NP-Hard a este mismo, podemos afirmar que pertenece a NP-C.

6) Una persona afirma tener un método eficiente para responder el pedido cualquiera sea la instancia. Utilizando el concepto de transitividad y la definición de NP-C explique qué ocurriría si se demuestra que la afirmación es correcta.

Comenzaremos enunciando lo que entendemos por "transitividad":

Dado un problema A que se puede reducir polinomialmente a un problema B. Es decir, $A \leq_p B$. Y dado que el problema B se puede reducir polinomialmente a un problema C: $B \leq_p C$. Entonces se puede afirmar que a A se puede reducir polinomialmente a C: $A \leq_p C$.

Ahora, en base al enunciado y al conocimiento en los problemas P, NP, NP-C y NP-H podemos decir que si al menos un problema de los más difíciles (en NP-C), que en nuestro caso sería "Casting del reality", se puede resolver en tiempo polinomial, entonces todos los problemas en NP se podrían resolver en tiempo polinomial.
Esto se explica por la definición de transitividad de la reducción polinomial y la definición misma de NP-H. Podemos en un encadenamiento de razonamientos demostrarlo:
Si llamamos X al problema de "Casting del reality" y Y al problema "eficiente" con resolución polinomial:

$$X \in NP - C,\ Y \in P,\ si\ X \leq_p Y$$
$$\Rightarrow \forall Z \in NP,\ Z \leq_p X \leq_p Y$$
$$\Rightarrow \forall Z \in NP,\ Z \leq_p Y$$
$$\Rightarrow \forall Z \in NP,\ Z \in P \Rightarrow NP = P$$

Esto implicaría que P = NP, es decir, que la clase de problemas que se pueden resolver en tiempo polinomial es la misma que la clase de problemas cuyas soluciones se pueden verificar en tiempo polinomial.

7) Un tercer problema al que llamaremos X se puede reducir polinomialmente al problema de "Casting del reality", qué podemos decir acerca de su complejidad?

No podemos afirmar nada acerca de su complejidad, dado que cualquier problema se puede complejizar tanto como uno quiera. Si fuera al revés la reducción polinomial, sí se podría afirmar que X pertenece a NP-Hard, dado que el problema de "Casting del reality" pertenece a NP-Hard.

Parte 3 (Segunda re-entrega): Un casting para el reality show

Enunciado

Contamos con un conjunto de "n" personas que conforman un grupo de un próximo reality show de supervivencia extrema. Algunas de esas personas se conocen entre sí y tienen una relación de

amistad preexistente. Se desea separar a las personas en dos equipos con la condición que los que tienen amistad queden siempre en el mismo equipo. Para lograrlo se nos permite eliminar con mucho "j" personas que puedan resultar conflictivas para cumplir con el cometido. La producción del programa desea saber si dado un casting determinado es posible lograr los solicitado.

Resolución

2) Demostrar que, dada una posible solución que obtenemos, se puede fácilmente determinar si se puede cumplir o no con la tarea solicitada.

En base a una instancia del problema con "n" personas, verificamos polinomialmente que en la solución, las personas fueron divididas correctamente las personas en dos equipos (es decir que ambos equipos deben tener al menos una persona), considerando que los que tienen amistad siempre estén en el mismo equipo. Además, también debo considerar que la cantidad de personas que fueron eliminadas sean igual o menos que "j".

Además de estas condiciones, previamente verificamos que las personas de la solución sean efectivamente las personas de la instancia del problema.

El pseudocódigo para el certificador sería:

```
fn certificador(P: personas, j: máxima cant personas a eliminar, E1: equipo 1, E2: equipo 2, J: personas eliminadas):

    si tamaño(E1) == 0 ó tamaño(E2) == 0:
        retorno Falso

    si tamaño(J) > j:
        retorno Falso

    si tamaño(E1) + tamaño(E2) + tamaño(J) != tamaño(P):
        retorno Falso

    por cada e1 de E1:
        si e1 no pertenece a P:
            retorno Falso

    por cada e2 de E2:
        si e2 no pertenece a P:
            retorno Falso

    por cada pj de J:
        si pj no pertenece a P:
            retorno Falso

    por cada e1 de E1:
        por cada e2 de E2:
            si e1 tiene relación de amistad con e2:
                retorno Falso

    retorno Verdadero
```

Análisis de complejidades:

- Complejidad temporal: $O(n^2)$
    - Las comparaciones de tamaños son operaciones $O(1)$.
    - Se itera por cada persona del equipo 1, lo cual podemos decir que en peor de los casos puede ser un número cercano a "n" (cantidad de personas), y a cada uno se le pregunta si está contenido en un grupo de "n" personas, lo cuál podría hacernos comparar con cada una de las personas. Por lo tanto, su complejidad es $O(n^2)$.

    - Se itera por cada persona del equipo 2, y se puede reutilizar la explicación anterior, aproximando la complejidad en $O(n^2)$ también.
    - Se itera por cada persona eliminada para ver si está contenida dentro del grupo de "n" personas, lo cual está acotado por "j" a iterar. Esto nos llevaría a pensar que la complejidad sería $O(j * n)$, pero siendo que "j" está acotado superiormente por "n", podemos afirmar que en el peor de los casos también es $O(n^2)$.
    - Se realiza doble iteración, por cada persona del equipo 1 contra cada persona del equipo 2, para verificar que no tengan una relación de amistad. Si decimos que el chequeo de la relación de amistad es $O(1)$, podemos aproximar esta doble iteración con $O(n^2)$, dado que se iteran "n" personas de un equipo, contra las "n" personas del otro.
- Complejidad espacial: $O(n)$
    - Las personas son "n", y la suma de los equipos más las personas eliminadas también, lo cual nos hace concluir que es $O(n)$.

Dado que demostramos que nuestro certificador es polinomial gracias al cálculo de complejidades, podemos afirmar que este problema pertenece a NP (es un problema de decisión con un certificador polinomial).

3) Demostrar que si desconocemos la solución la misma es difícil de resolver. Utilizar para eso el problema "Mínimum Node Deletion bipartite Subgraph" (suponiendo que sabemos que este es NP-C).

El "Minimum Node Deletion Bipartite Subgraph" es un problema de decisión en donde dado un grafo $G = (V, E)$ y un valor "j" entero positivo. Queremos determinar si es posible construir un grafo bipartito eliminando no más de "j" nodos.

Dado un grafo no dirigido $G = (V, E)$, el objetivo es determinar el **mínimo número de nodos** que deben eliminarse del grafo para que el subgrafo restante sea **bipartito**.

Algunos conceptos claves son:

Grafo bipartito:
Un grafo es bipartito si sus nodos pueden dividirse en dos conjuntos U y W tal que no haya aristas entre nodos del mismo conjunto.

Eliminar nodos:
Se busca eliminar el menor número posible de nodos de G para garantizar que el subgrafo resultante sea bipartito.

Dado que lo que buscaremos es demostrar que el problema de "Un casting para el reality show" es NP-Hard, para demostrar que es "difícil de resolver", propongo reducir el problema de "Minimum Node Deletion Bipartite Subgraph" (X) al problema de "Un casting para el reality show" (Y):

$X \leq_p Y$

Dado un grafo no dirigido $G = (V, E)$, ¿Se pueden eliminar como máximo "j" nodos para que el subgrafo restante sea bipartito?

Ahora lo que resta, es buscar una reducción polinomial para cumplir con lo siguiente:
$X \rightarrow transf.\ polinomial \rightarrow Y \rightarrow Caja\ negra\ que\ resuelve(Y) \rightarrow S_y \rightarrow transf.\ polinomial \rightarrow S_x$

Partimos de:

- $G = (V, E)$: grafo no dirigido

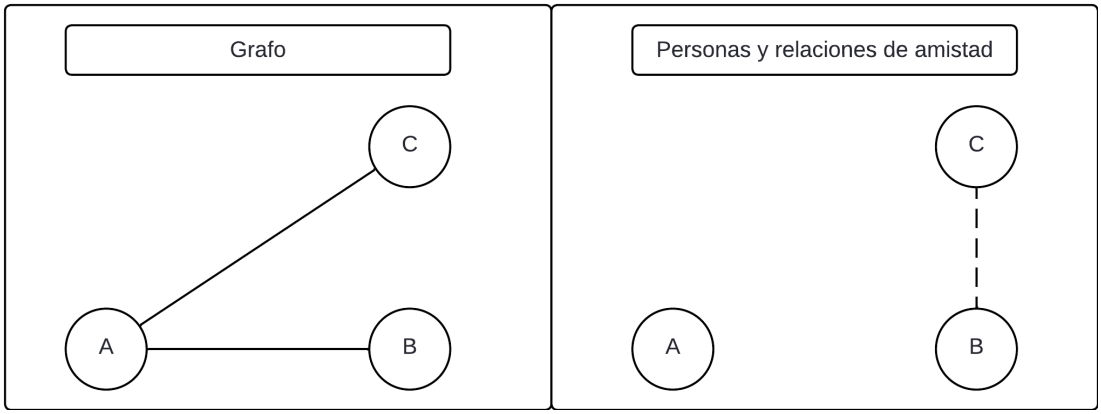
- j: máxima cantidad de nodos a eliminar
- El problema “Minimum Node Deletion Bipartite Subgraph” es NP-Hard por ser NP-C

Análisis general:

- Lo primero que notamos es que si dos nodos están unidos por una arista, estos mismos no pueden estar en el mismo conjunto luego de la partición final.
- Relacionándolo con el problema “Un casting para el reality show”, podemos pensar en un principio que cada nodo del problema “Minimum Node Deletion Bipartite Subgraph” puede transformarse en una persona del problema “Un casting para el reality show”.
- A su vez, como vimos antes, dado que dos nodos que se encuentran unidos por una arista, no pueden estar en el mismo conjunto, los podemos pensar como dos personas que necesariamente tienen que estar en dos equipos distintos, es decir, son “no amigos” o “rivales”.

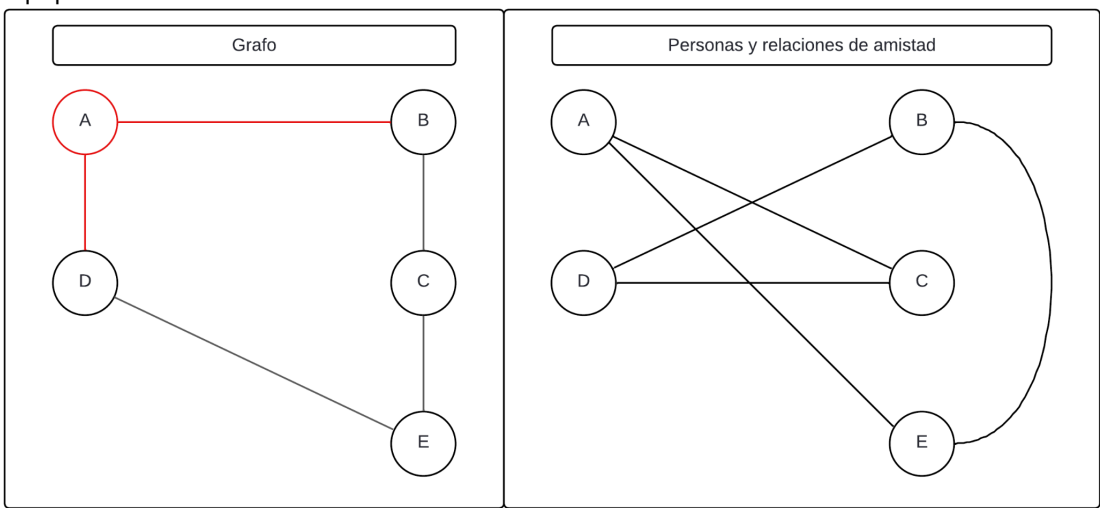
Análisis de una transformación que NO funciona:

- A partir de lo mencionado anteriormente, podemos notar que el grafo complementario del inicial, podría a llegar a darnos información de “amistades” entre personas.
- Por lo tanto podríamos plantear que cada nodo es una persona, y las aristas del grafo complementario son relaciones de amistad. A continuación muestro un caso donde funciona lo mencionado anteriormente:



Como vemos, pareciera existir una coherencia entre un grafo bipartito y dos equipos.

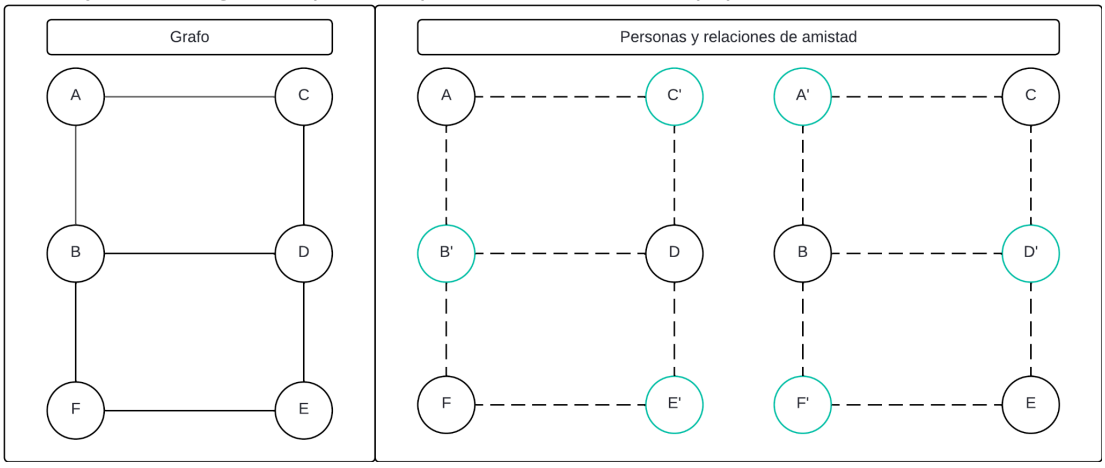
- Sin embargo, nos encontramos con el siguiente contraejemplo, en donde planteando que $j = j' = 1$, nos encontramos con que se podría eliminar un nodo (el marcado en rojo) y generar un grafo bipartito, pero no se puede eliminar una única persona y generar dos equipos:



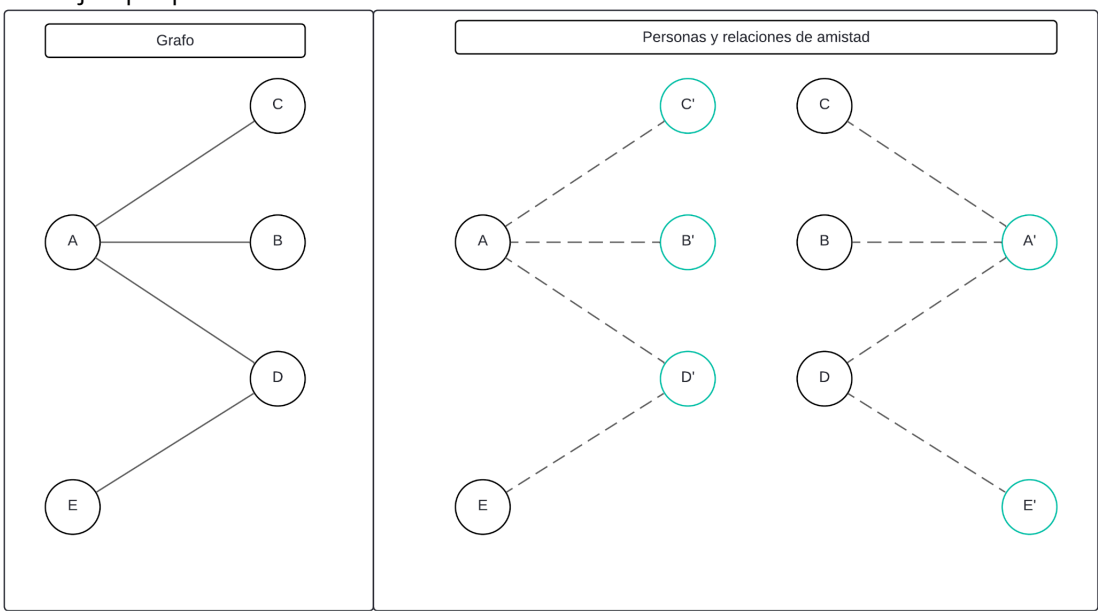
- Por lo tanto concluimos que utilizar simplemente el grafo complementario, no funciona.

Análisis para realizar una transformación posiblemente exitosa:

- La transformación que explicaré a continuación fue probada con distintos casos dando el resultado esperado, pero no pudimos demostrar matemáticamente que sirve para todos los casos.
- Utilizando lo explicado en “Análisis general” continuamos con otro análisis.
- Cabe destacar que estas “no amistades” o “rivalidades” tienen que dejar únicamente dos equipos, dado que si dejan un solo equipo sería absurdo, porque significa que hay un bucle en las rivalidades, y por ende, pensando en el problema inicial de los nodos, hay un ciclo, lo cual no nos permitiría hacer un grafo bipartito (a menos que podamos eliminar nodos/personas).
- Para representar estas rivalidades, podemos pensar que cada nodo "v" se va a transformar en dos personas: "v" y "v'". Además, cada arista $(u,\ v)$ hará una amistad entre la persona "v" y la "u'", y otra entre la persona "v'" y la "u".
- Esto va a generar que la persona "u" no tenga amistad con la persona "v" (que es lo que queríamos desde un comienzo), pero que "u" sí tenga amistad con "v'" (representando una “no amistad” o “rivalidad”). Esta enemistad es la clave de esta transformación, dado que nos va a quedar cada nodo unido a sus rivales.
- Dejo una imagen de ejemplo para que se entienda mejor la transformación de rivalidades, en donde podemos ver a la izquierda un grafo bipartito, que es transformado a personas con relaciones de amistad. Estas relaciones de amistad generan dos equipos claramente distinguibles, en donde podemos ver que los todos los nodos que iban a quedar en uno de los conjuntos del grafo bipartito, quedan en el mismo equipo.

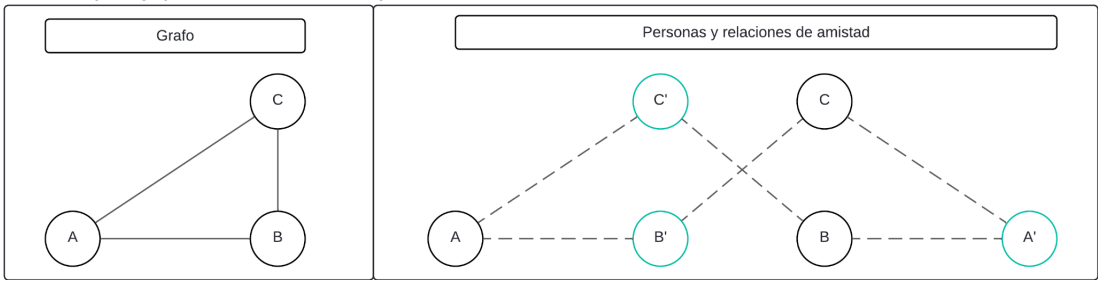


Otro ejemplo podría ser:



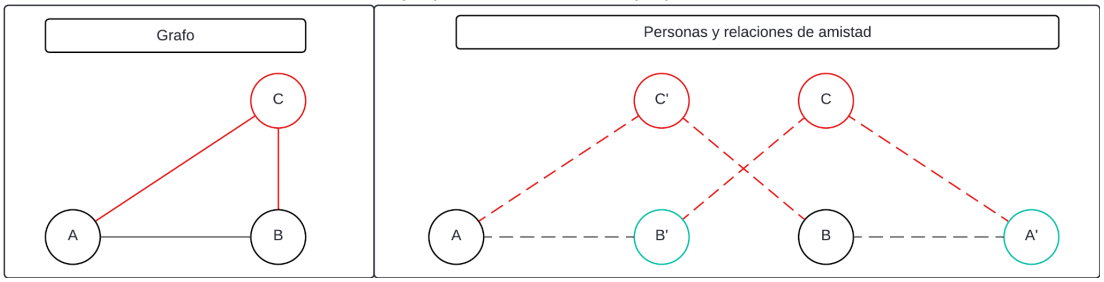
Esta bipartición pareciera dejar correctamente los nodos que esperamos que sean de la misma partición, como personas que están en el mismo equipo, debido a que estamos respetando la morfología original del grafo, quitando las personas conflictivas y reemplazandolas por su opuesto (su estado de “no-amigo” o “rival”).

- Podemos concluir entonces que cuando el grafo inicial es bipartito, esto siempre nos va a devolver dos equipos, y por ende la respuesta para ambos casos es “Verdadero” para todo valor de "j" mayor o igual a cero.
- Ahora debemos demostrar qué pasa cuando no es bipartito el grafo inicial y se deben eliminar como máximo "j" nodos. Si sabemos que hay "j" nodos a eliminar, podemos pensar que un nodo "u" perteneciente a los "J" tiene que estar unido a al menos un nodo de cada conjunto (dado que sino no sería conflictivo y podría estar en uno de ellos, y además esos nodos a los cuales está unido no pueden ir al otro conjunto), que traducido al problema de las personas y amistades, sería que la persona "u" tiene es rival de una persona "$e1$" del equipo 1 y una persona "$e2$" del equipo 2 (y a su vez estas personas también lo tienen como rival). Esto genera una contracción dado que la persona "$e1$" es rival de "$e2$", la persona "$e2$" rival de "u" y "u" de "$e1$" (ABSURDO). La contradicción se debe a que hay solo dos equipos, con lo cual, o sos rival de uno, o sos rival del otro, no se puede de ambos. Esta contradicción hace que el problema del “Un casting para el reality show” se deban eliminar las personas conflictivas ("u" y "u'").
- A modo de ejemplo, dejo una imagen de qué pasaría con un grafo de tres nodos (grafo con ciclo impar y por lo tanto no bipartito):

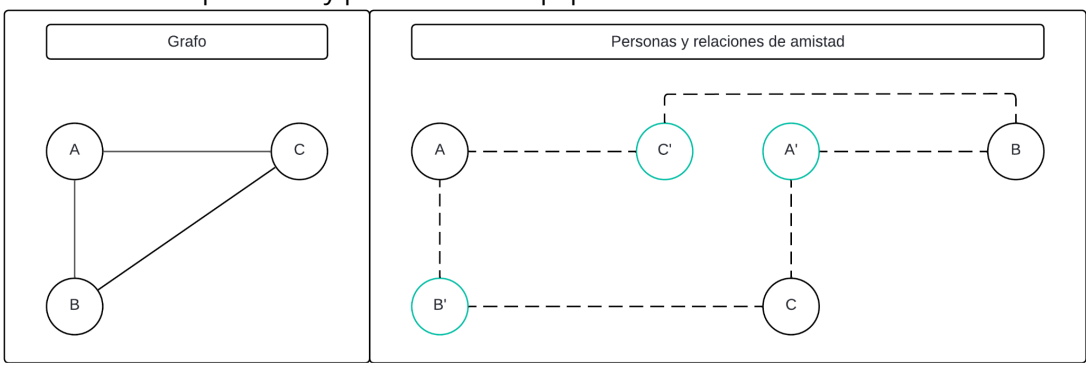


Vemos que es necesario eliminar uno de los nodos, por lo tanto con $j = 0$ ambos problemas retornarán Falso. Si planteamos que $j = 1$, lo que esperaríamos es que en el grafo se elimine cualquiera de los nodos para generar la bipartición, por ejemplo el nodo C.

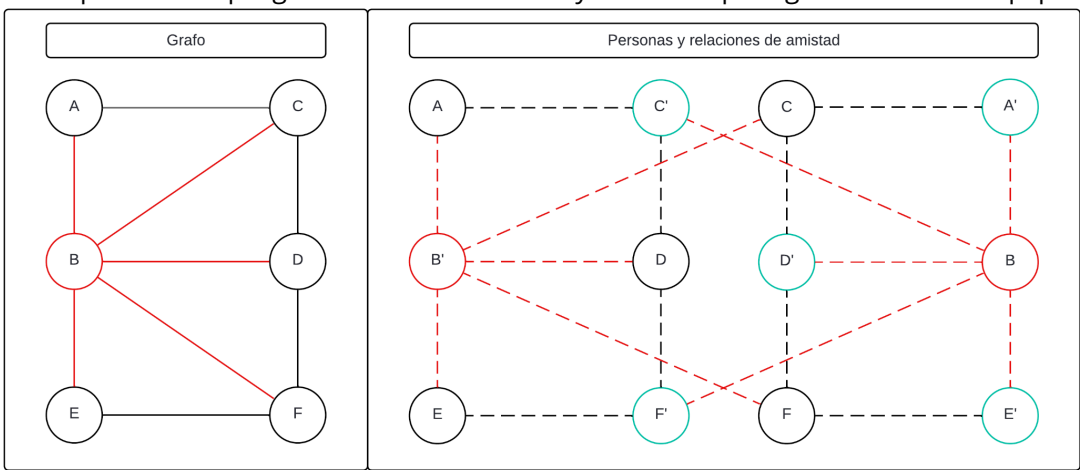
Para el problema “Un casting para el reality show” también es necesario eliminar relaciones de amistad para generar dos equipos, pero en un grado mayor a "j" dado que nosotros representamos los enemigos de una persona "u", pero también representamos a quien tiene de enemigo esa misma persona con "u'".
Entonces, para que tenga sentido eliminar esa información, planteamos que para el problema de las personas, la máxima cantidad de personas a eliminar: $j' = 2 * j$.
Con esta última transformación, vemos que se podrían eliminar "C" y "C'", para quitar la información de las enemistades y quedarían dos equipos nuevamente:



- Sin embargo, si entramos más en detalle, podemos ver que se podría eliminar cualquier par de nodos que rompan el ciclo que se forma dentro del equipo y esto generaría la desconexión de personas y por ende dos equipos.

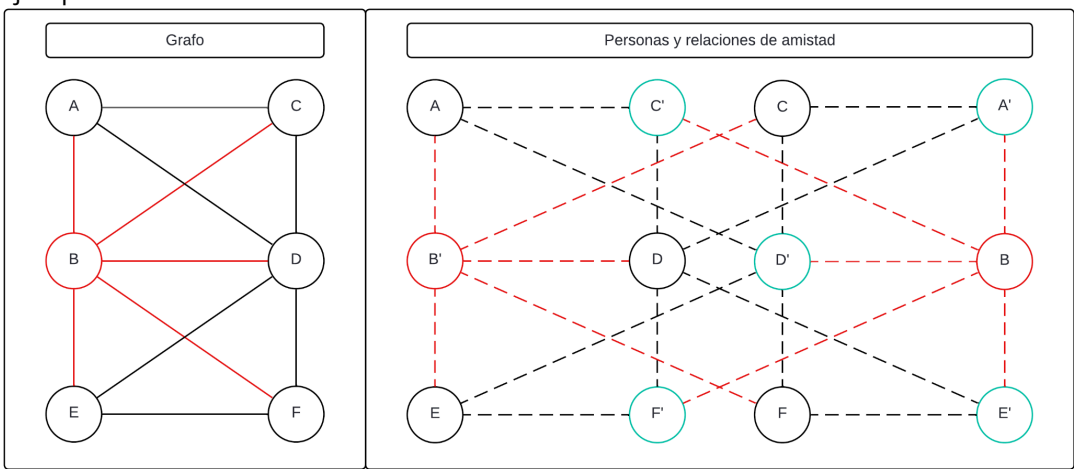


Otro ejemplo con más nodos para notar que aquellos nodos que pesan más en la bipartición del grafo inicial, son aquellas personas que también van a querer ser eliminadas más rápido, dado que son las que generan más amistades y conflictos para generar los dos equipos:



- Si bien esto no rompe lo explicado anteriormente, notamos que cuando hay un ciclo impar en el grafo inicial, se genera un bucle de personas en el problema del reality, que nos obliga a quitar dos personas para generar dos equipos (cosa que no pasaba con los grafos con ciclo par). Pero generando más casos, nos damos cuenta que esto no solamente funciona para $j = 1 \Rightarrow j' = 2$ para el caso Verdadero, sino que para el Falso también, siendo que se

mantienen los ciclos impares restantes como bucles en las personas para el siguiente ejemplo:



Además, si planteamos un $j = 2 \Rightarrow j' = 4$, ambos casos dan Verdadero, debido que luego de eliminar los dos nodos más conflictivos (que podrían ser B y B'), se podría proseguir por D, dejando dos equipos. Si bien en el problema de las personas y relaciones bastó con eliminar tres personas para generar dos equipos, esto no contradice que siga respetando la reducción correctamente.

Por todos los ejemplos mencionados anteriormente, me inclino a pensar que podría ser una reducción correcta y por ende la utilizo a continuación:

Planteo el pseudocódigo de la transformación en base a lo explicado anteriormente para demostrar que es polinomial:

```
fn transformacionPolinomial1(V: vértices, E: aristas, j: cantidad máxima de nodos a eliminar):

  P: personas
  A: amistades entre personas
  j': cantidad máxima de personas a eliminar

  por cada v de V:
    agrego una persona llamada "v"
    agrego una persona llamada "v'"

  por cada (u, v) de E:
    agrego una relación de amistad entre persona llamada "u" y "v'"
    agrego una relación de amistad entre persona llamada "u'" y "v"

  j' = 2 * j

  retorno (P, A, j')
```

Análisis de complejidades:

- Complejidad temporal: $O(V + E)$
  - La iteración de los vértices para crear personas tiene una complejidad $O(V)$.
  - La iteración de las aristas para crear las relaciones de amistad tiene una complejidad $O(E)$.
- Complejidad espacial: $O(V + E)$
  - Por cada vértice genero dos personas, por lo tanto la complejidad es $O(V)$.
  - Por cada arista genero dos relaciones de amistad, por lo tanto la complejidad es $O(E)$

Como vemos, la primera transformación para pasar del problema de “Minimum Node Deletion Bipartite Subgraph” a “Un casting para el reality show” es polinomial, por lo tanto estoy reduciendo un problema a otro polinomialmente.

Ahora me resta transformar la solución de “Un casting para el reality show” a la solución de “Minimum Node Deletion Bipartite Subgraph”.
Por todo lo explicado anteriormente, la transformación es directa:

```
fn transformacionPolinomial2(S: solución problema reality show):

   retorno S
```

Análisis de complejidades:

- Complejidad temporal: $O(1)$
- Complejidad espacial: $O(1)$

Dado que ambas transformaciones son polinomiales, puedo afirmar que reduje un problema NP-Hard al problema “Un casting para el reality show”, demostrando que el mismo es tan complejo como un problema NP-Hard.

4) Demostrar que el problema “Mínimum Node Deletion bipartite Subgraph” pertenece a NP-C. (Para la demostración puede ayudarse con diferentes problemas, recomendamos “Clique problem”)

Un Clique es un problema de decisión NP-Completo que consiste en que dado un grafo $G = (V, E)$ y un valor "k" entero positivo, queremos determinar si existe un subconjunto de nodos V' de V tal que estos conformen un subgrafo completo utilizando los ejes de E de tamaño "k" o mayor.

Mientras que, como explicamos anteriormente, el “Minimum Node Deletion Bipartite Subgraph” es es también un problema de decisión en donde dado un grafo $G = (V, E)$ y un valor "k" entero positivo. Queremos determinar si es posible construir un grafo bipartito eliminando no más de "k" nodos.

Para probar que el problema “Minimum Node Deletion Bipartite Subgraph” ∈ NP-C primero debemos demostrar que existe un certificador polinomial (dado que ya es un problema de decisión). En base a esto construiremos uno:

```
fn certificador(G, k, U, W, K):
   Sea G = (V, E): matriz de adyacencias
   Sea k: cantidad máxima de nodos a eliminar

   Sea U, W: conjuntos bipartitos
   Sea K: un conjunto de nodos eliminados

   si tamaño(U) == 0 ó tamaño(W) == 0:
      retorno Falso

   si tamaño(K) > k:
      retorno Falso

   si tamaño(U) + tamaño(W) + tamaño(K) != tamaño(V):
      retorno Falso

   por cada u de U:
      si u no pertenece a V
         retorno Falso

   por cada w de W:
```

```
    si w no pertenece a V
        retorno Falso

por cada vk de K:
    si vk no pertenece a V
        retorno Falso

por cada u de U:
    por cada vértice adyacente v de G[u]:
        si v pertenece a U:
            retorno Falso

por cada w de W:
    por cada vértice adyacente v de G[w]:
        si v pertenece a W:
            retorno Falso

retorno Verdadero
```

Análisis de complejidades:

- Complejidad temporal: $O(V^3)$
    - Las comparaciones de tamaños son operaciones $O(1)$.
    - Se itera por cada nodo de U, lo cual podemos decir que en peor de los casos puede ser un número cercano a "V", y a cada uno se le pregunta si está contenido en un grupo de "V" vértices, lo cuál podría hacernos comparar con cada uno de los nodos. Por lo tanto, su complejidad es $O(V^2)$.
    - Se itera por cada nodo de W, y se puede reutilizar la explicación anterior, aproximando la complejidad en $O(V^2)$ también.
    - Se itera por cada nodo eliminado para saber si está incluido en "V", lo cual está acotado por "k". Esto nos llevaría a pensar que la complejidad sería $O(k * V)$, pero siendo que "k" está acotado superiormente por "V", podemos afirmar que en el peor de los casos también es $O(V^2)$.
    - Se itera por cada arista adyacente de cada nodo de U, lo cual en la matriz de adyacencias tiene una complejidad de $O(V^2)$, y dado que además tenemos que preguntar si este adyacente está contenido en U, llegamos a una complejidad temporal de $O(V^3)$ para el peor de los casos en que U tenga un número de vértices cercanos a "V". Esto lo realizamos para asegurarnos que en cada conjunto, no haya dos nodos que son “vecinos”.
    - Lo mismo que se detalló anteriormente sucede para cada adyacente a cada nodo de W. Por lo tanto, podemos concluir que acá también la complejidad temporal es de $O(V^3)$.
- Complejidad espacial: $O(V^2)$
    - Se guardan los nodos y aristas en una matriz de adyacencias, por lo tanto $O(V^2)$
    - Los conjuntos U, W y K a lo sumo pueden tener "V" vértices cada uno, por lo tanto la complejidad está dada por $O(V)$.

Dado que es un problema de decisión y además podemos verificar una solución del problema en tiempo polinomial, podemos afirmar que el problema “Minimum Node Deletion Bipartite Subgraph” **pertenece a NP**.

Ahora demostraremos que también pertenece a NP-Hard a través de una reducción polinomial del problema del “Clique”:

Dado que lo que buscaremos es demostrar que el problema de “Minimum Node Deletion Bipartite Subgraph” es NP-Hard, propongo reducir el problema del “Clique” (X) al problema de “Minimum Node Deletion Bipartite Subgraph” (Y):

$X \leq_p Y$

Ambos problemas, por cómo los enunciamos previamente, son problemas de decisión.

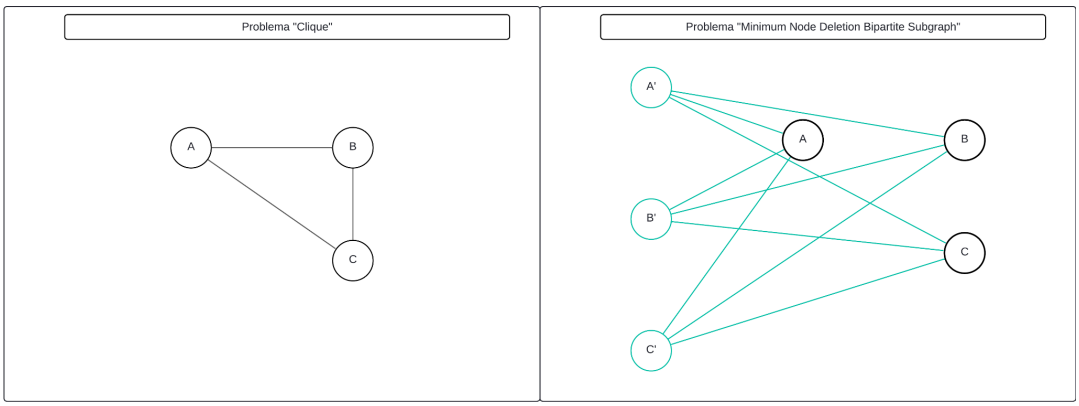
Ahora lo que resta, es buscar una reducción polinomial para cumplir con lo siguiente:
$X \rightarrow transf.\ polinomial \rightarrow Y \rightarrow Caja\ negra\ que\ resuelve(Y) \rightarrow S_y \rightarrow transf.\ polinomial \rightarrow S_x$

Partimos de:

- $G = (V, E)$: grafo no dirigido
- k: tamaño mínimo del *clique* (nodos de V que conforman un grafo completo)
- El problema del "Clique" es NP-Hard por ser NP-C

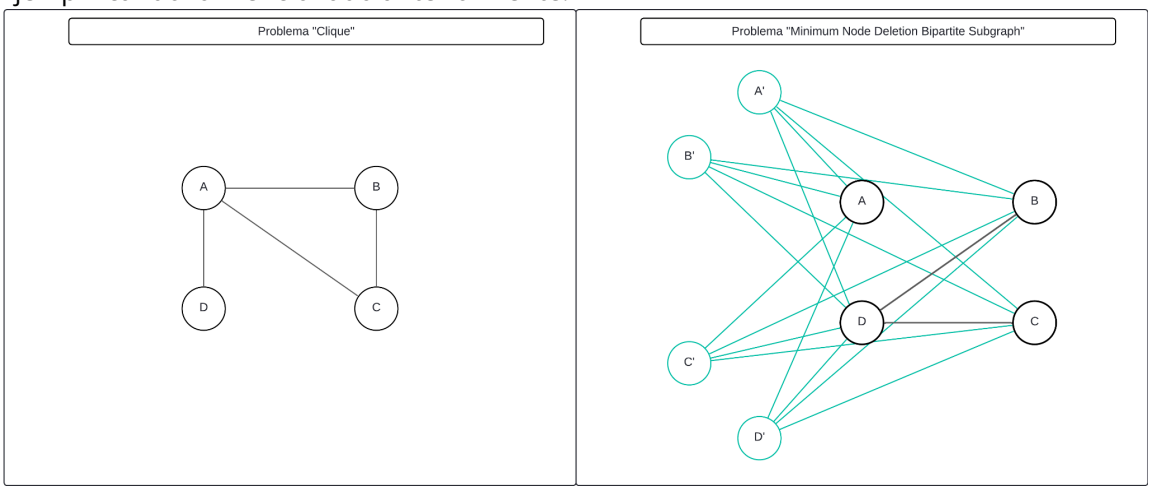
Análisis para realizar la transformación:

- Como objetivo, para poder determinar si existe un clique de tamaño "k" dentro del grafo "G", intentaremos dejar este clique de un lado de la partición, eliminando todos los nodos que no forman parte del clique. Para esto, en el problema de "Minimum Node Deletion Bipartite Subgraph" vamos tener que tener al menos los mismos nodos que en el problema del "Clique".
- Por un momento, supongo que el grafo "G" es un grafo completo, y por lo tanto un clique de tamaño $|V|$. Si quisiéramos resolver el problema del "Clique" con $k = |V|$, luego de transformarlo a una instancia del problema "Minimum Node Deletion Bipartite Subgraph", esperaríamos:
  - que el clique quede de un lado de la partición
  - que no se eliminen nodos, por lo tanto $k' = 0$ (k' parámetro que recibe el problema "Minimum Node Deletion Bipartite Subgraph")
- Para esto:
  - los nodos del clique cuando se transformen en la instancia del problema de "Minimum Node Deletion Bipartite Subgraph" no deben estar unidos por aristas para que queden del mismo lado de la partición.
  - debemos forzar mediante otros nodos, que estos nodos del clique queden del mismo lado. Para esto, por cada nodo v de V, crearemos un nodo v' que se tenga una arista contra todos los nodos pertenecientes a V.
- Ejemplificando, tendríamos algo como lo siguiente con un grafo completo de tres nodos y un $k = 3$:

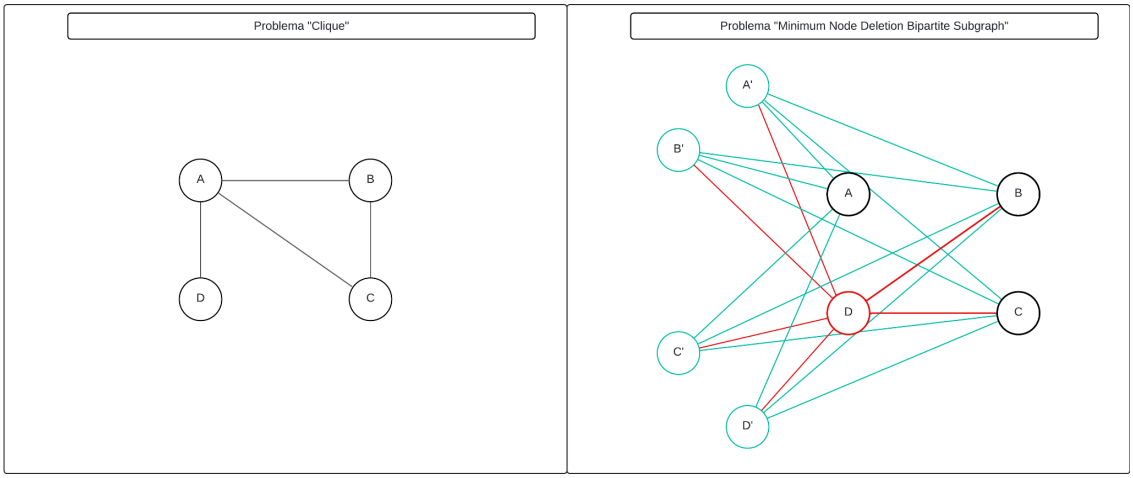


Como explicamos antes, se crearon los nodos adicionales, unidos a los originales, y además se hizo que el $k' = 0$.
Para cualquier instancia de G que sea grafo completo, esto funciona debido a que siempre queda un grafo bipartito con todos los nodos de G de un lado de la bipartición.

- Ahora debemos atacar el problema de cómo eliminar los nodos que no forman parte del clique. Dado que necesitamos eliminar estos nodos en el grafo bipartito, haremos que aquellos nodos que no estén unidos a otros en el grafo original, sí lo estén en el grafo de la transformación, es decir, trabajaremos con el grafo complementario de G, que lo llamaremos G'. Para lo explicado anteriormente, esto sigue funcionando, dado que el grafo complementario de un clique, es que estos nodos no tengan aristas que los una.
- Esta transformación es muy importante porque hace que aquellos nodos que no forman parte del clique, "estorben" para hacer la bipartición, y deban ser eliminados. Además, debido a que a los nodos auxiliares v' los unimos con todos los nodos de V, le generamos al problema "Minimum Node Deletion Bipartite Subgraph" la obligación de que tenga que eliminar preferentemente los nodos que no pertenecen al clique pero que sí pertenecen a V.
- En base a la explicación anterior, podemos deducir que siempre vamos a querer eliminar como máximo todos los nodos que no están en el clique, y en caso que se deban eliminar más, significa que el clique encontrado de G es de menor tamaño y por ende debe dar falso tanto el problema del "Clique" como también el de "Minimum Node Deletion Bipartite Subgraph". Para esto entonces planteamos $k' = |V| - k$, donde $|V|$ es la cantidad de nodos del grafo original G y "k" es el clique de tamaño mínimo a encontrar.
- Ejemplificando lo mencionado anteriormente:



Si en este caso, nos hubieran dado un $k = 4$, ambos problemas deberían decirnos que no es posible encontrar un clique de tamaño al menos cuatro. Luego de transformarlo al problema "Minimum Node Deletion Bipartite Subgraph", con un $k' = |V| - k = 4 - 4 = 0$, también nos indicaría que no es posible generar una bipartición sin eliminar al menos un nodo.
En cambio, si buscamos un clique de tamaño tres, es decir $k = 3$, y por lo tanto $k' = |V| - k = 4 - 3 = 1$, veríamos que sí se puede eliminar un nodo, y que nos quede el clique de un lado de la partición como esperábamos:



Además, el nodo más conveniente, es como dijimos, aquel que no pertenece al clique.

Planteo el pseudocódigo de la transformación en base a lo explicado anteriormente para demostrar que es polinomial:

```
fn transformacionPolinomial1(V: vértices, E: aristas, k: mínimo tamaño del clique):

   V': vértices para problema "Minimum Node Deletion Bipartite Subgraph"
   E': aristas para problema "Minimum Node Deletion Bipartite Subgraph"
   k': cantidad máxima de nodos a eliminar para problema "Minimum Node Deletion Bipartite Subgraph"

   // creo G': grafo complementario de G
   por cada v de V:
      agrego v a V'

   por cada u de V:
      por cada v de V:
         si v != u:
            agrego arista (u, v) a E'

   por cada (u, v) de E:
      elimino arista (u, v) de E'

   // nodos auxiliares
   por cada u de V:
      genero un nodo u' a partir de u
      agrego u' a V'
      por cada v de V:
         agrego arista (u', v) a E'

   k' = tamaño(V) - k

   retorno (V', E', k')
```

Análisis de complejidades:

- Complejidad temporal: $O(E * V^2 + V^2) = O((E + 1) * V^2) = O(E * V^2)$
    - La iteración de los vértices para crear vértices del grafo complementario tiene una complejidad $O(V)$.
    - La doble iteración a través de los vértices, para unir todos los nodos con todos los demás (excepto con él mismo) tiene una complejidad de $O(V^2)$. Esto se debe a que asumo que agregar una arista a E' se puede considerar como $O(1)$.
    - Luego, eliminar las aristas del grafo original, para generar el grafo complementario, itera a través de todas las aristas de "E" y elimina de entre "V^2" elementos (porque se genera esa cantidad de aristas para tener un grafo completo). Si consideramos

que la operación de eliminar debe recorrer todos los elementos para encontrar cuál borrar, es una operación $O(V^2)$. Por lo tanto, tendríamos una complejidad de $O(E * V^2)$.
    - Por último, la conexión entre todos los nodos auxiliares y los nodos del grafo original, tiene una complejidad de $O(V^2)$, debido a que se iteran lo "V" vértices auxiliares y se los conecta con los "V" vértices del grafo original.
    - El cálculo de k' es una operación $O(1)$.
- Complejidad espacial: $O(V + E)$
    - Por cada vértice genero dos vértices, por lo tanto la complejidad es $O(V)$.
    - Para generar el grafo complementario, tenemos que saber que un grafo completo de "n" vértices tiene $\frac{n*(n-1)}{2}$ aristas. Por lo tanto, si nuestro grafo tiene "E" aristas, podemos decir que el complementario tendrá "$\frac{V*(V-1)}{2} - E$" aristas (considerando que "V" es la cantidad de nodos del grafo inicial). En el peor de los casos, este E podría ser un número muy chico, y por lo tanto, deberíamos almacenar $O(\frac{V*(V-1)}{2})$ que sacando las constantes y aproximando, sería $O(V^2)$.
    - Por último debemos considerar las aristas de los nodos auxiliares, que están conectados a todos los nodos del grafo inicial, y por lo tanto nos dan una complejidad de $O(V^2)$, dado que debemos guardar que cada arista auxiliar, está conectada a todos los otros nodos del grafo inicial.

Como vemos, la primera transformación para pasar del problema del “Clique” a “Minimum Node Deletion Bipartite Subgraph” es polinomial, por lo tanto estoy reduciendo un problema a otro polinomialmente.

Ahora me resta transformar la solución de “Minimum Node Deletion Bipartite Subgraph” a la solución de “Clique”. Por todo lo explicado anteriormente, la transformación es directa:

```
fn transformacionPolinomial2(S: solución problema MNDBS):

  retorno S
```

Análisis de complejidades:
- Complejidad temporal: $O(1)$
- Complejidad espacial: $O(1)$

Dado que ambas transformaciones son polinomiales, puedo afirmar que reduje un problema NP-Hard al problema “Minimum Node Deletion Bipartite Subgraph”, demostrando que el mismo es tan complejo como un problema NP-Hard.

Y dado que pertenece a NP y además a NP-Hard, podemos afirmar que pertenece a NP-C.