Trabajo Práctico N°2
Grupo *undefined*

Integrantes

- Luciano Martin Gamberale. Padrón: 105892.
- Gonzalo Javier Iglesias. Padrón: 107213.
- Alejandro Binker. Padrón: 107056.
- Gabriel La Torre. Padrón: 87796.
- Santiago Nicolás Reynoso Dunjo. Padrón: 107051.

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

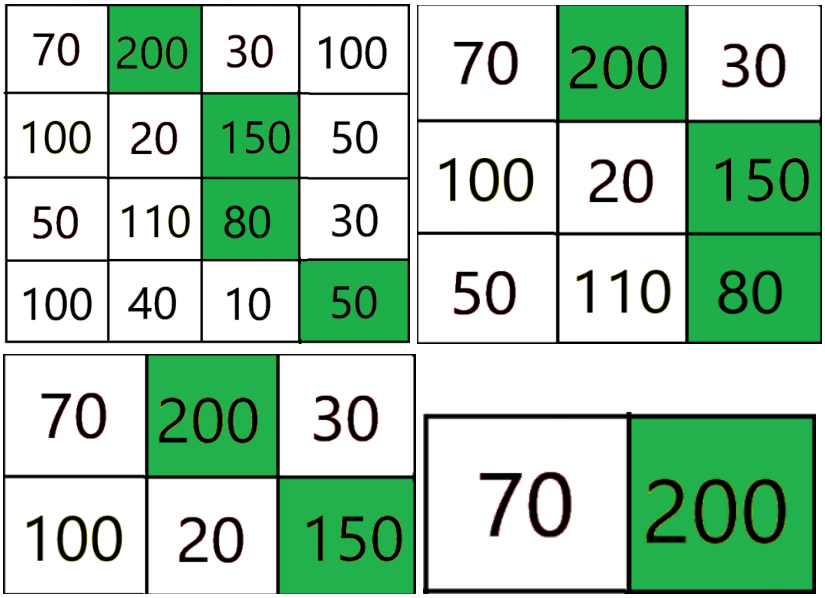
Cuadra verde: actual
Cuadras rojas: movimientos no válidos por ganancia menor
Cuadras amarillas: movimientos válidos
Cuadras grises: movimientos no válidos por estar fuera de la submatriz

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)

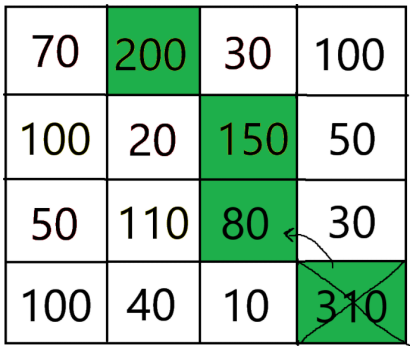
Ahora supongamos que queremos resolver la instancia de este problema (empezando en el sureste de la matriz) y estamos chequeando la máxima ganancia posible en 50, podrían ocurrir 3 cosas:

- 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

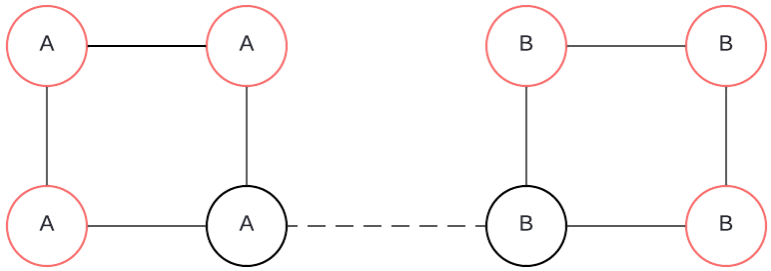
Enunciado

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.

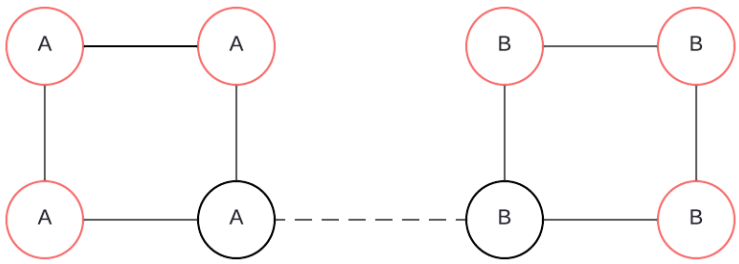
Resolución

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

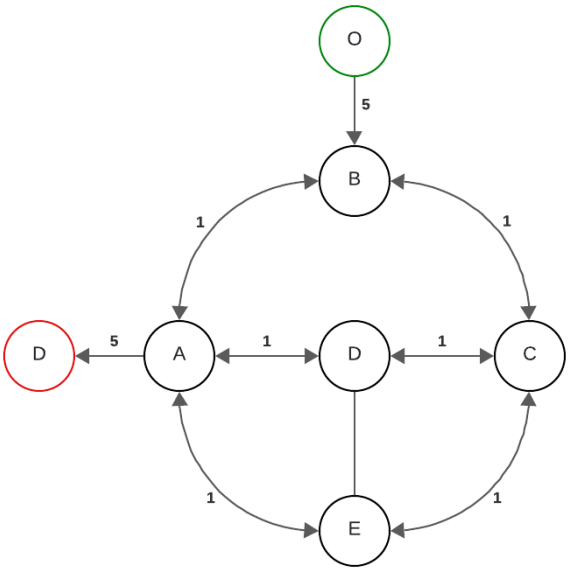
Voy a realizar la siguiente reducción para llevar el problema de las rutas a un problema de flujo mínimo:

- 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

Enunciado

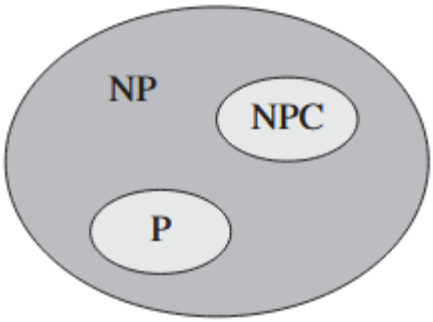
Contamos con un conjunto de "n" personas que conforman un grupo de un próximo reality show de supervivencia extrema. Algunas de esas personas se conocen entre sí y tienen una relación de amistad preexistente. Se desea separar a las personas en dos equipos con la condición que los que tienen amistad queden 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.

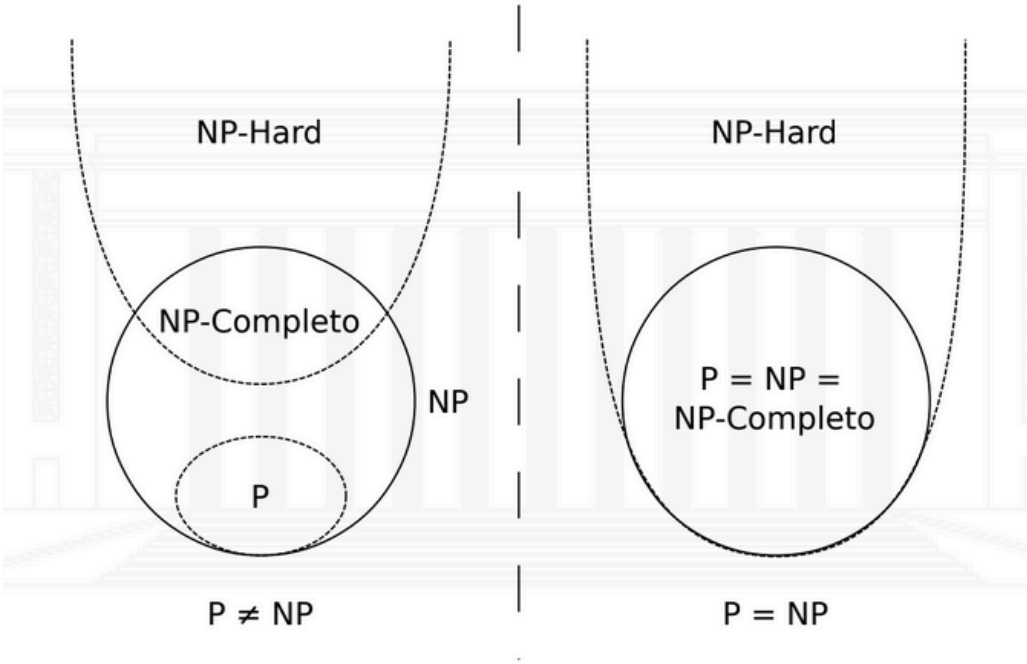
Resolución

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.

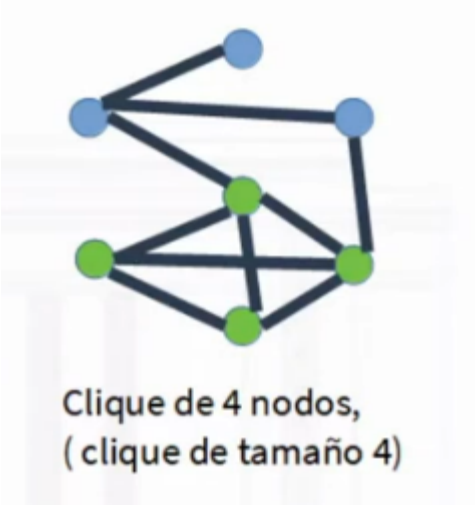
Dada una partición de 2 equipos. Representamos cada miembro como un nodo con sus amistades como aristas. Toma n iteraciones, donde n es el tamaño de la suma de los miembros de ambos equipos, determinar si alguno de los nodos tiene un amigo en otro equipo. Esta verificación es polinomial.

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:

a) La cantidad de nodos de T |T|= |V|-j.
b) 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