FACULTAD DE INGENIERÍA
Universidad de Buenos Aires

75.29 / 95.06 Teoría de Algoritmos

Trabajo Práctico Nº 2

Teoría de Logaritmos

Fecha de Entrega: Lunes 21 de Octubre de 2024

Integrantes:

Alumno	Padrón
Joaquin Andresen	102707
Federico Penic	102501
Facundo Pareja	99719
Rogger Aldair Paredes Tavara	97976

Índice

Parte 1: Las tareas del servidor público.

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

Definición de subproblema:

El subproblema se puede definir como la **máxima ganancia que podemos obtener al llegar a la celda** (i, j), considerando que:

- Podemos movernos desde las celdas adyacentes, es decir, desde la celda de **arriba** $(i - 1, j)$ o desde la celda **a la izquierda** $(i, j - 1)$, siempre y cuando el valor de la celda actual sea menor que el valor de las celdas adyacentes.
- El valor en la celda actual puede o no sumarse, dependiendo de si cumple la condición de decreciente.

La variable del subproblema sería $OPT(i, j)$, donde:

- $OPT(i, j)$ = máxima ganancia acumulada al llegar a la celda (i, j).

Relación de recurrencia:

$$OPT(i,j) = \begin{cases} M[i,j] & \text{si } M[i,j] > \max(OPT(i,j-1), OPT(i-1,j)) \\ \max(OPT(i,j), OPT(i,j-1), OPT(i-1,j)) & \text{si } M[i,j] \leq \min(OPT(i,j-1), OPT(i-1,j)) \end{cases}$$

Caso Base: $OPT(0,0) = M[0,0]$

Pseudocódigo

```
INICIO

Definir función LEER_MATRIZ_ARCHIVO(archivo):
  Abrir el archivo
  Leer cada línea y convertirla en una lista de enteros
  Devolver la matriz

Definir función MAX_GANANCIA(matriz, n, m, memo, manzanas):
  Si estamos en la esquina superior izquierda:
      Retornar valor en matriz[0][0] y manzanas[n][m]
```

```
  Si memo[n][m] no es cero:
      Retornar valor en memo[n][m] y manzanas[n][m]
  Inicializar variables para verificar los caminos desde arriba (top) y desde la izquierda (left)

  Si se puede mover a la izquierda:
    valor izquierda = Calcular la ganancia izquierda con MAX_GANANCIA(matriz, n, m-1, memo, manzanas)
  Si se puede mover hacia arriba:
    valor arriba = Calcular la ganancia superior con MAX_GANANCIA(matriz, n-1, m, memo, manzanas)

  Comparar el valor actual con los valores desde la izquierda y desde arriba:
    Si el valor actual es mayor, se guarda solo la posición actual
    Si los caminos anteriores tienen mayor valor, se suman y se guarda el mejor camino

  Actualizar memo y manzanas con el mejor resultado
  Devolver el valor máximo y las posiciones correspondientes

Definir función PRINCIPAL():
  Leer los parámetros de filas y columnas
  Leer la matriz desde el archivo manzanas
  Inicializar memo y manzanas vacias
  Calcular ganancia_total y manzanas_seleccionadas con MAX_GANANCIA
  Imprimir manzanas seleccionadas y ganancia total

FIN
```

2. Analice la complejidad espacial y temporal de su propuesta

Complejidad Temporal

- **Número total de subproblemas**: Hay $n \times m$ subproblemas, donde n es el número de filas y m es el número de columnas de la matriz.
- **Costo por subproblema**: En el peor caso, el bucle anidado que busca el mejor camino puede recorrer todas las celdas anteriores, lo que resulta en $O((n+1) \times (m+1))$ para cada celda. Este es el paso más costoso.

En resumen, la **complejidad temporal total** del algoritmo es $O(n^2 \times m^2)$, porque en cada celda (n, m) En el peor caso, podemos recorrer todas las celdas anteriores en la submatriz.

Complejidad Espacial

La complejidad espacial está dominada por las dos estructuras memo, paths y manzanas, que ocupan $O(n \times m)$ espacio.

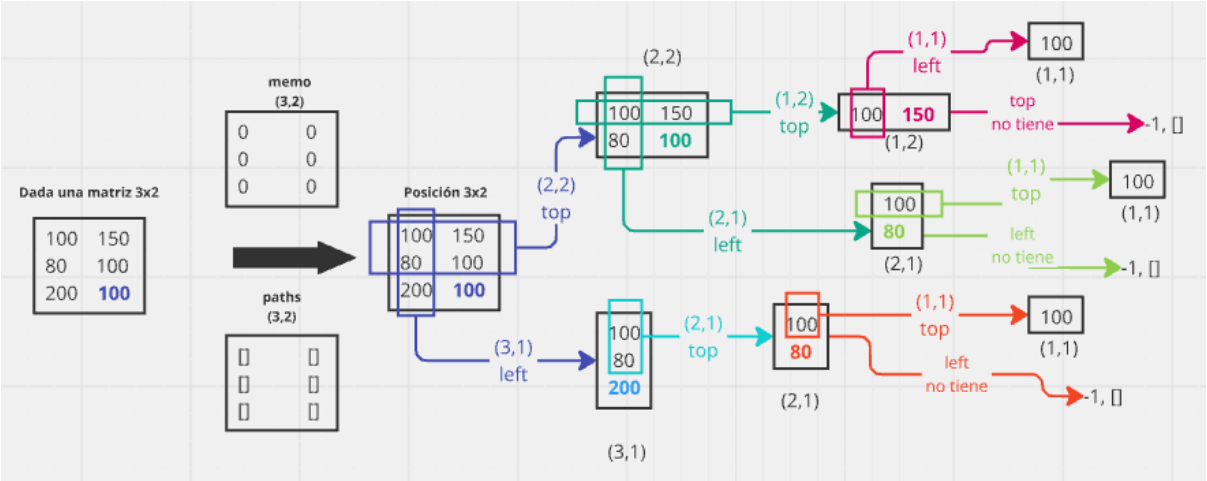
Por lo tanto, la **complejidad espacial** total es:

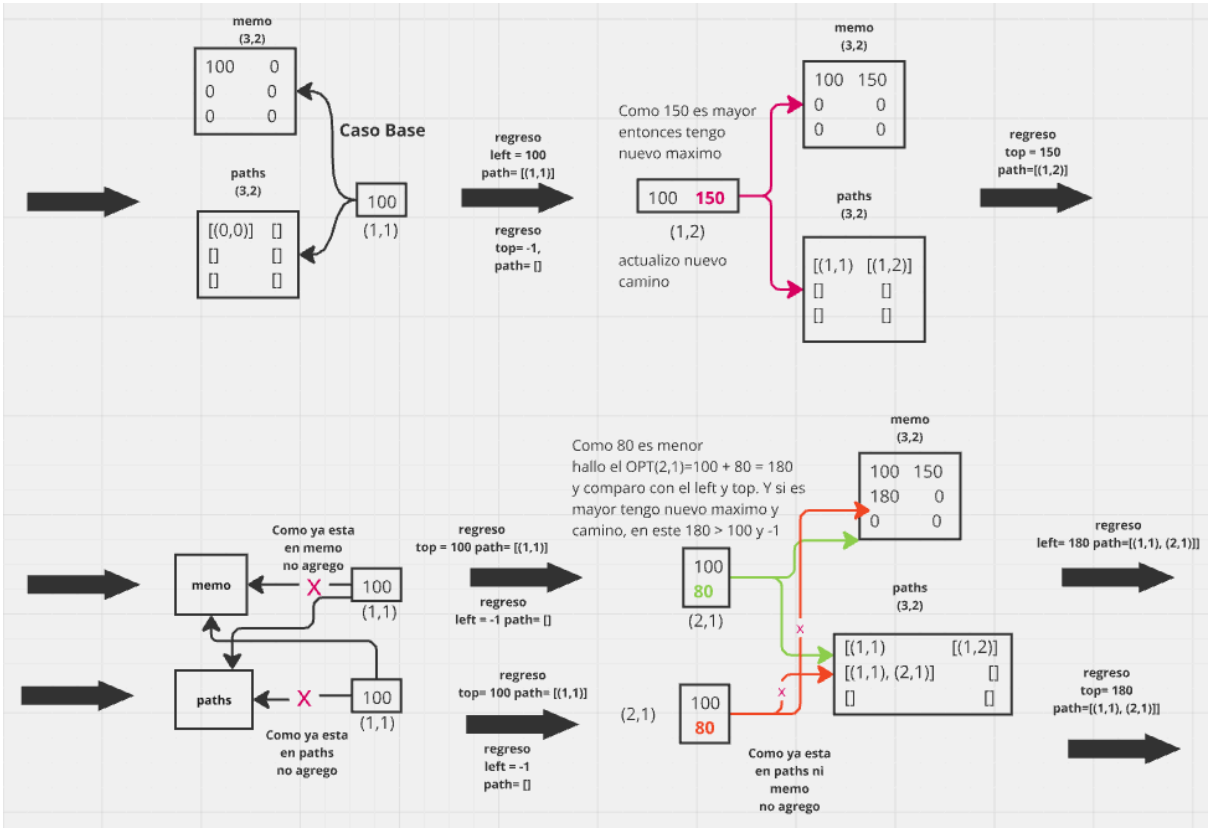
- $O(n \times m)$: por las matrices de memo y manzanas.
- $O(n \times m)$: para los caminos temporales en manzanas.

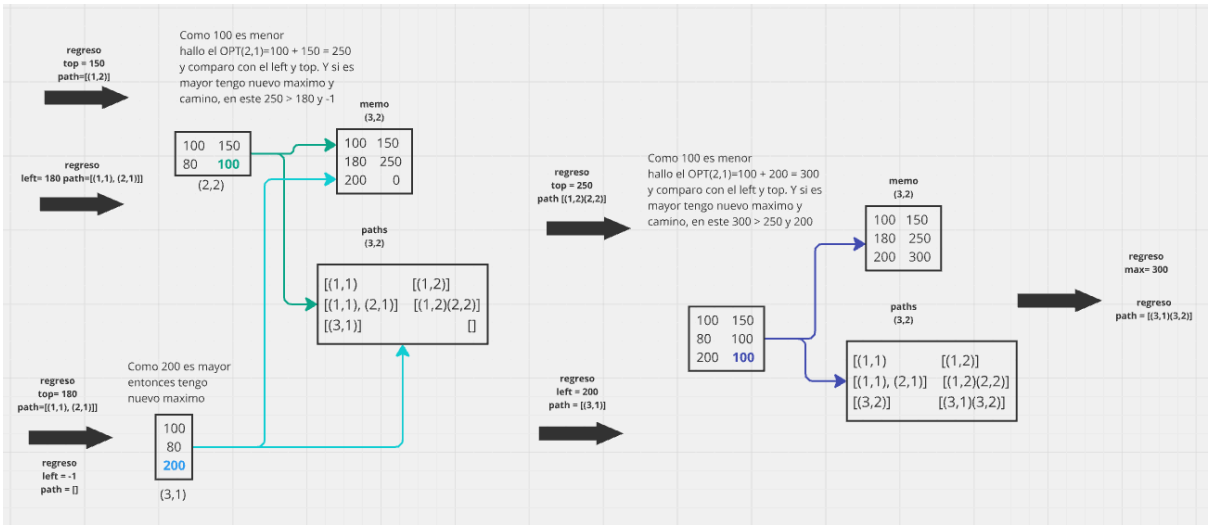
Complejidad temporal: $O(n^2 \times m^2)$.
Complejidad espacial: $O(n \times m)$.

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

Primero hacemos la descomposición de las submatrices hasta llegar al caso base, que para él el ejemplo sería 100:







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

El programa se debe ejecutar de la siguiente manera:

python3 tareas.py <cant-cuadras-norte-sur> <cant-cuadras-este-oeste> <archivo-ganancias>

A continuación se brindan dos ejemplos:

Ejemplo 1:

python3 tareas.py 4 5 manzanas1.txt	
100,150,300,100,50 80,100,80,120,100 200,100,90,120,100 140,60,80,90,50	**Manzanas: (1,3) (2,4) (2,5) (4,5)** **Ganancia: 300 + 120 + 100 + 50 = 570**

Ejemplo 2:

python3 tareas.py 4 2 manzanas2.txt	
70, 0 60, 9 100, 100 50, 0	**Manzanas: (1,1) (2,1) (4,1)** **Ganancia: 70 + 60 + 50 = 180**

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

Como lo calculado en pseudocódigo la complejidad es igual al del algoritmo.

Complejidad Temporal Total:

- **Número total de subproblemas**: Para una matriz de n filas y m columnas, hay $n \times m$ subproblemas, ya que el algoritmo calcula el valor para cada celda ($n \times m$).
- **Costo por subproblema**:
    - En el peor caso, el bucle que busca el mejor camino puede recorrer todas las celdas anteriores a ($n \times m$)., lo que implica un costo de $O((n + 1) \times (m + 1))$ para cada celda ($n \times m$)..

Por lo tanto, la **complejidad temporal total** del algoritmo es $O(n^2 \times m^2)$.

Complejidad Espacial Total:

- **Matrices memo y manzanas**: $O(n \times m)$.
- **Caminos temporales en manzanas**: En el peor caso, ocupan $O(n \times m)$.

Por lo tanto, la **complejidad espacial** total del algoritmo es $O(n \times m)$, dominada por las matrices de memo y manzanas.

A esto también podríamos incluir los cálculos de la lectura del archivo pero no afectan al cálculo de la complejidad

Parte 2: La fortaleza de la red de transporte

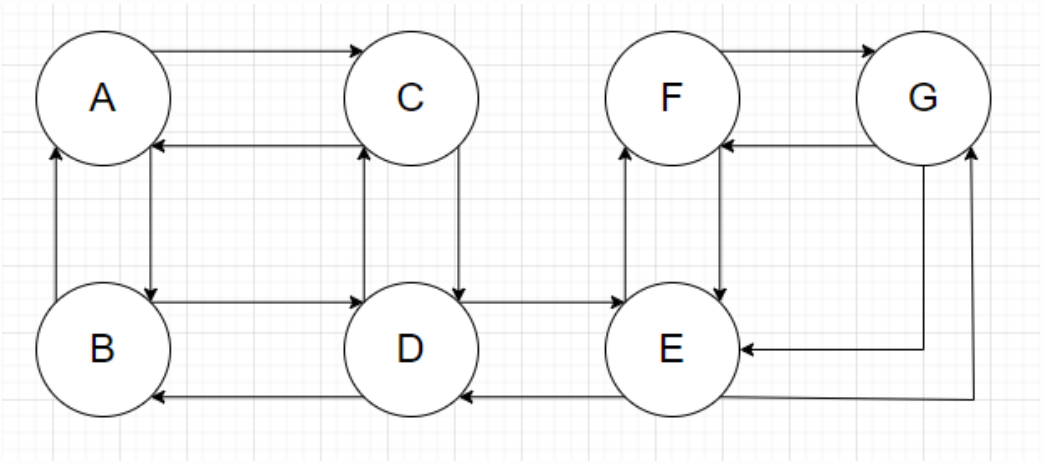
1. Considerar la siguiente propuesta: “Aquella ciudad que cuenta con menor cantidad de rutas entrantes se debe considerar como la causante de debilidad de la red de transporte. Sus rutas adyacentes corresponden al mínimo buscado”. Demostrar la optimalidad o invalidez de la afirmación.

La afirmación sugiere que la solución al problema de encontrar el corte mínimo se puede obtener simplemente observando la ciudad con menos rutas entrantes y cortando las rutas adyacentes a esa ciudad. Es decir, se plantea que el número de conexiones (o rutas) de una ciudad está directamente relacionado con su importancia en la red y que la desconexión de sus rutas adyacentes sería la solución óptima para desconectar la red.

Sin embargo, el número de rutas entrantes a una ciudad no necesariamente está directamente relacionado con el corte mínimo, ya que este depende de cómo se distribuye el flujo entre todas las conexiones de la red y no solo de la cantidad de rutas de una ciudad.

Por lo tanto, la afirmación propuesta es inválida. La ciudad con menos rutas entrantes no necesariamente representa la debilidad de la red ni sus rutas adyacentes corresponden al corte mínimo buscado. La estructura completa del grafo y la distribución del flujo a través de las rutas son los factores que determinan el corte mínimo, por lo que es necesario aplicar algoritmos de flujo máximo para identificar de manera correcta el corte mínimo.

Como contraejemplo podríamos plantear el siguiente grafo:



En este caso, la ciudad E es una de las ciudades que tiene mayor cantidad de rutas entrantes (Son 3: DE, FE y

GE). No obstante, al hacer el corte de una sola de sus rutas (DE) ya estamos desconectando un gran número de ciudades. Esta ruta corresponde a una debilidad de la red de transporte.

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

El objetivo de este problema es encontrar el mínimo número de rutas (indicando cuáles son) cuya eliminación provoque la desconexión entre alguna de las ciudades. Podemos resolver este tipo de tareas mediante la búsqueda de cortes mínimos utilizando redes de flujo.

En primer lugar, deberemos modelar el problema como un grafo, donde las ciudades son vértices y las rutas serán ejes. En pos de poder aplicar algún algoritmo que maximiza el flujo (Como el algoritmo de Ford-Fulkerson), y en consecuencia obtener el corte mínimo, deberemos crear dos ejes de sentido opuesto por cada ruta (debido a que las rutas son de doble mano). Es decir, que si hay una ruta entre las ciudades A y B, creamos dos ejes, uno de A hacia B, y otro de B hacia A. Todos estos ejes tendrán capacidad 1, ya que estamos interesados en el número de rutas que pueden cortarse.

La idea principal es aplicar un algoritmo de flujo máximo. Para ello, seleccionamos dos ciudades como "fuente" y "sumidero", y calculamos el flujo máximo entre ellas. El teorema del flujo máximo y corte mínimo nos asegura que el valor del flujo máximo es igual al tamaño del corte mínimo, es decir, la cantidad mínima de rutas cuya eliminación desconectará esas dos ciudades. Una vez que calculamos el flujo máximo y el grafo residual correspondiente, podemos identificar las rutas específicas que forman el corte mínimo. Estas rutas serán aquellas donde exista un eje entre un vértice alcanzable desde la fuente y uno no alcanzable desde el sumidero.

Dado que el problema no especifica un par de ciudades en particular para desconectar, deberemos aplicar este procedimiento entre todas las posibles combinaciones de ciudades como fuente y sumidero, y así encontrar el corte mínimo global. El corte que contenga la menor cantidad de rutas nos dará la solución al problema.

Esta solución nos asegura que, cortando el conjunto de rutas encontrado, al menos una de las ciudades quedará desconectada del resto, cumpliendo con los requerimientos del problema.

3. Presentar pseudocódigo

A continuación, mostramos el pseudocódigo del programa que resuelve el corte mínimo:

```
INICIO

Definir función CONSTRUIR_GRAFO(archivo):
```

```
  Crear grafo vacío
  Abrir el archivo
  Para cada línea en el archivo:
    Leer origen y destino de la línea
    Añadir eje bidireccional entre origen y destino con capacidad 1
  Devolver el grafo

Definir función EXISTE_CAMINO_AUMENTANTE(grafo, fuente, sumidero, padres):
  Crear conjunto de nodos visitados
  Crear cola con el nodo fuente
  Marcar fuente como visitado

  Mientras la cola no esté vacía:
    Extraer nodo de la cola
    Para cada nodo vecino con capacidad positiva:
      Si el nodo vecino no ha sido visitado:
        Añadir a la cola
        Marcar como visitado
        Registrar el nodo actual como padre del vecino (En pos de saber la ruta al sumidero)
        Si el nodo vecino es el sumidero:
          Devolver verdadero (se encontró un camino)
  Devolver falso (no se encontró camino)

Definir función FORD_FULKERSON(grafo, fuente, sumidero):
  Inicializar flujo máximo a 0
  Crear una copia del grafo que será el grafo residual

  Mientras haya un camino aumentante desde la fuente al sumidero:
    Encontrar el cuello de botella en el camino aumentante
    Actualizar las capacidades en el grafo residual
    Sumar el cuello de botella al flujo máximo

  Devolver flujo máximo y el grafo residual

Definir función ENCONTRAR_CORTE_MINIMO(grafo, fuente, sumidero, grafo_residual):
  Crear conjunto de nodos visitados
  Crear cola con el nodo fuente
  Marcar fuente como visitado

  Mientras la cola no esté vacía:
    Extraer nodo de la cola
    Para cada nodo vecino en el grafo residual con capacidad positiva:
      Si el nodo vecino no ha sido visitado:
        Añadir a la cola
```

```
            Marcar como visitado

    Crear lista de ejes para el corte mínimo
    Para cada nodo visitado:
        Para cada nodo vecino en el grafo original:
            Si el vecino no está visitado y hay capacidad en el eje:
                Añadir el eje al corte mínimo

    Devolver el corte mínimo

Definir función ENCONTRAR_CORTE_MINIMO_GLOBAL(grafo):
    Obtener la lista de ciudades a partir del grafo
    Inicializar el corte mínimo global vacío y la cantidad mínima de rutas a infinito

    Para cada combinación de ciudades como fuente y sumidero:
        Calcular flujo máximo y obtener el grafo residual usando Ford-Fulkerson
        Encontrar el corte mínimo entre fuente y sumidero
        Si el tamaño del corte es menor que la cantidad mínima de rutas:
            Actualizar la cantidad mínima y el corte mínimo global

    Devolver el corte mínimo global y la cantidad mínima de rutas

Definir función PRINCIPAL():
    Leer el archivo de rutas
    Construir el grafo a partir del archivo
    Buscar el corte mínimo global
    Mostrar la cantidad mínima de rutas y el corte mínimo

FIN
```

4. Realizar un análisis de optimalidad.

Por cada combinación de fuente y sumidero, nuestra función FORD_FULKERSON itera buscando un camino de aumento y finaliza cuando no logra encontrar uno. Luego construimos el corte mínimo utilizando el grafo residual.

Sabemos que Ford Fulkerson nos garantiza obtener el flujo máximo de manera óptima, y que este es equivalente a encontrar el corte mínimo.

Teniendo en cuenta que nuestro objetivo es encontrar los caminos mínimos para distintas combinaciones de fuentes y sumideros, podemos afirmar que utilizando Ford Fulkerson y BFS para hallar los cortes mínimos nuestra solución es óptima.

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

Complejidad Temporal

Para empezar, inicialmente debemos transformar el grafo en uno dirigido. Esto implica aumentar el número de aristas de m a 2m. Nuestra función que construye el grafo a partir de un archivo realiza esta acción de forma lineal $O(m)$, pero esta operación se realiza una sola vez y no tendrá mucho impacto en la complejidad final.

Al realizar Ford Fulkerson, tendremos que buscar caminos de aumento y actualizar su flujo. En el peor de los casos se tendrá que realizar C veces, donde C es la cantidad de ejes que salen de la fuente (ya que la capacidad siempre va a ser 1 en este caso). Y cada vez que tengamos que realizar BFS para encontrar el camino de aumento tendremos una complejidad de $O(V + E)$, siendo V el número de vértices y E el número de ejes. -> $O(C * (V + E))$

Luego de obtener el flujo máximo y el grafo residual tendremos que buscar el corte mínimo realizando BFS nuevamente -> $O(V + E)$

Estas operaciones van a ser realizadas para cada combinación posible de ciudades, es decir $\binom{V}{2}$ veces. Por lo tanto la complejidad temporal final es:

$$O(\binom{V}{2} * C * (V + E))$$

Complejidad Espacial

Tanto el grafo inicial como los residuales requieren $O(V + E)$.

La variable *parent* necesita $O(V)$ para guardar el camino de los padres al realizar BFS. Al igual que *visitados* y la cola *queue*.

La complejidad espacial final es:

$$O(V + E)$$

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

Para correr el programa, tendremos que contar con una versión de python superior a 3.0. Y simplemente, correr el siguiente comando:

```
python3 transporte.py rutas.txt
```

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

Para resolver el problema tuvimos que transformar nuestro problema en un problema de flujo máximo. Para ello tuvimos que realizar la siguiente transformación:

Tenemos un grafo no dirigido $G = (V, E)$ donde V es el conjunto de nodos que representa a las ciudades y E es el conjunto de aristas que representa a las rutas.

La transformación realizada consiste en transformar cada arista en dos aristas dirigidas con capacidad igual a 1.

Esta transformación es realizada una sola vez al comienzo de nuestra solución. Cuando se lee el archivo con las rutas se genera el grafo en tiempo polinomial $O(m)$ siendo m la cantidad de aristas.

Parte 3: Un casting para el reality show

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 como 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 lo solicitado.

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

La **clase de complejidad P** comprende aquellos problemas de decisión para los cuales existen un algoritmo A que los **resuelve** con complejidad polinomial, o sea existe constante k tal que $A = O(n^k)$

La **clase de complejidad NP** comprende aquellos problemas de decisión para los cuales existen un algoritmo B que los **certifica** (verifica que una solución dada es válida) con complejidad polinomial, o sea existe constante k tal que $B = O(n^k)$

Es fácil de ver que se puede usar un algoritmo de resolución polinomial en un algoritmo de certificación polinomial de la siguiente manera:

```
funcion certificacionPolinomial(problema, solución):
    resolucionPolinomial = resolver_polinomialmente(problema)
    if resolucionPolinomial == solucion:
        return True
    return False
```

Por lo tanto, $Q \in P \Rightarrow Q \in NP$.

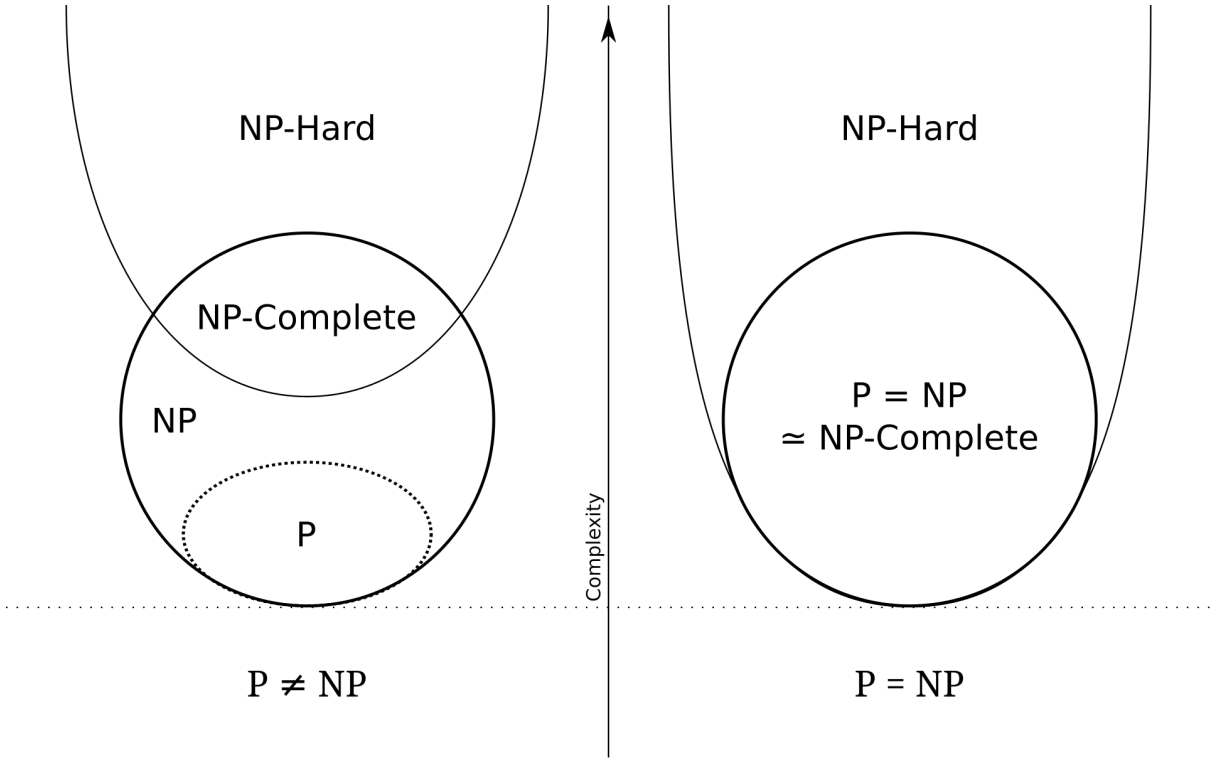
No hay al día de la fecha prueba de que P = NP.

Para presentar las otras dos clases es necesario introducir el concepto de **reducción polinomial**. Esta es una transformación de una instancia Y de un problema en una instancia de otro X en tiempo polinomial y se nota $Y \leq_p X$. Se dice en este caso que el problema X es al menos tan difícil como el problema Y.

La **clase de complejidad NP-H** comprende aquellos problemas a los cuales puede reducirse todo problema NP. Se dice en este caso que X es al menos igual de difícil que cualquier problema NP.

La **clase de complejidad NP-C** comprende aquellos problemas que son a la vez NP y NP-H. Se dice en este caso que X es de los problemas NP más difíciles.

La relación exacta entre los conjuntos depende de la relación exacta entre P y NP (aun no probada). Se muestran ambas posibilidades:



Para el caso **P=NP**

Como todo problema P es NP, y todo problema NP puede reducirse a un problema NP-H, todo problema P puede reducirse a un problema NP-H. Además, el conjunto NP-C es común a NP-H, NP (por definición) y P (por lo asumido).

Para el caso **P≠NP**

Se observa que el conjunto P está contenido en NP (explicado más arriba), que el conjunto NP-C es común a NP-H y NP (por definición) y que los problemas NP-H son más difíciles que todos excepto los NP-C (a su vez más difíciles que el resto de los NP y P).

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

Para una solución al problema, consistente en

- Las *k* personas a eliminar.
- Los dos grupos de personas resultantes de esta eliminación.

se observa que es verificable *fácilmente* (es decir, en tiempo polinomial) comprobando que:

- *k* <= *j* que se puede realizar en $O(1)$.
- Iterando cada arista e para comprobar que ambos vértices corresponden al mismo grupo o a una persona eliminada. Esto se puede realizar en $O(EV)$.

Queda así probado lo pedido.

3. Demostrar que si desconocemos la solución la misma es difícil de resolver. Utilizar para eso el problema *Minimum node deletion bipartite subgraph* (suponiendo que sabemos que este es NP-C).

Se pide probar que resolver el problema del casting es NP-C, o sea de los problemas NP más difíciles. Para esto, se muestra que:

- El problema es NP, o sea se puede certificar una solución en tiempo polinomial. Esto ya fue demostrado en el inciso 2.
- El problema es NP-H. Para esto se puede partir de un problema Y ∈ NP-C y reducirlo polinomialmente a X. Este problema Y está determinado por el enunciado en este caso (*Minimum node deletion bipartite subgraph*).

El problema *Minimum node deletion bipartite subgraph* tiene el siguiente enunciado:

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.

Es fácil de ver que el mismo se puede reducir polinomialmente al problema del casting realizando las siguientes consideraciones:

- k tiene el significado de j en el problema del casting
- Cada vértice v ∈ V representa un participante del show.
- Cada arista e (u,v) ∈ E representa una amistad entre los participantes u y v.

Es trivial que esto se puede realizar en tiempo polinomial (como mucho se requiere renombrar los vértices antes y después de encontrada la solución).

Por lo tanto, como

$Y \in NP - C$

$X \in NP \wedge Y \leq_p X \Rightarrow X \in NP - C$

4. Demostrar que el problema *Minimum node deletion bipartite subgraph* pertenece a NP-C. (Para la demostración puede ayudarse con diferentes problemas, recomendamos *Clique problem*).

Se siguen los mismos pasos del inciso 3)

En primer lugar se quiere probar que el problema es NP. Para esto, dada una solución consistente en un grupo de vértices, se debe iterar la misma y remover cada elemento del conjunto V del grafo, junto con sus aristas asociadas. Esto se puede realizar con complejidad polinomial$O[k(V-1)] = O(V)$ donde k es el número máximo de vértices que es posible remover como parte del enunciado del problema y $V-1$ una cota del máximo posible de aristas que posee cada vértice. Por lo tanto el problema es NP.

En segundo lugar se quiere verificar que el problema es NP-H. Partiendo del problema *Clique* $\in$ NP-C para un grafo $G_1\ (V,\ E)$ y un valor k_1 entero positivo, se lo puede reducir polinomialmente al problema *Minimum node deletion bipartite subgraph* de la siguiente forma:

- Creando un grafo $G_2(V, \overline{E})$ donde los vértices son los mismos y el conjunto de aristas es el complemento del conjunto de aristas E de G_1
- Tomando un $k_2 = |V| - k_1$

Para justificar esta equivalencia se hacen las siguientes observaciones:

- Un clique de tamaño k equivale a un conjunto de k nodos aislados (sin aristas) en G_2. Esto es fácil de ver pues si no hay ninguna arista entre k nodos en G_2, eso significa que tienen todas las aristas posibles (o sea es un clique de k nodos) en G_1
- Por lo tanto, una vez obtenido G_2, si es posible aislar k_1 nodos removiendo hasta $|V| - k_1$ nodos (el máximo posible) se habrá probado lo pedido. Los vértices aislados resultantes son los que conforman el clique pedido.

Esta reducción se puede hacer en $O[n(n-1)] = O[n^2]$ o sea con complejidad polinomial. Es así porque

se puede realizar obteniendo cada arista posible y eliminando aquellas presentes en E.

Por lo tanto, *Minimum node deletion bipartite subgraph* es NP-H.

Con estas dos deducciones, se puede concluir entonces que *Minimum node deletion bipartite subgraph* es NP-C.

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

El problema casting del reality es:

- NP (probado en el inciso 2)
- NP-H (probado en el inciso 2)
- NP-C (probado en el inciso 3)

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 la afirmación es verdadera, resulta que el problema del casting es P, o sea se puede encontrar una solución en tiempo polinomial.

Por otro lado, supongamos otro problema $Y \in NP - C$ cualquiera. Se observa que se ambos se pueden reducir entre sí con igual complejidad.

$Y \leq_p CASTING$

$CASTING \leq_p Y$

Esto prueba que $Y \in P$ y por lo tanto $NP = P$

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?

Como $X \leq$ y p $CASTING$ $CASTING \in NP - C \Rightarrow X \in NP - H$

Referencias

- Algoritmos y Complejidad - Algoritmos sobre Grafos
- Diferencia entre Branch & Bound y Backtracking

Correcciones:

Parte 1:

Corrección 1: Ecuación de Recurrencia

$$\mathrm{OPT}(n,m) = \begin{cases} \mathrm{memo}[n,m] & \text{si } \mathrm{memo}[n,m] > \mathrm{OPT}(n-1,m) \text{ y } \mathrm{memo}[n,m] > \mathrm{OPT}(n,m-1) \\ \mathrm{OPT}(n-1,m) & \text{si } \mathrm{OPT}(n-1,m) > \mathrm{memo}[n,m] \text{ y } \mathrm{OPT}(n-1,m) \geq \mathrm{OPT}(n,m-1) \\ \mathrm{OPT}(n,m-1) & \text{si } \mathrm{OPT}(n,m-1) > \mathrm{memo}[n,m] \text{ y } \mathrm{OPT}(n,m-1) \geq \mathrm{OPT}(n-1,m) \\ \mathrm{matriz}[n][m] & \text{si } \mathrm{matriz}[n][m] > \mathrm{OPT}(n-1,m) \text{ y } \mathrm{matriz}[n][m] > \mathrm{OPT}(n,m-1) \end{cases}$$

Siendo *memo* la matriz donde almacenamos las ganancias dada una determinada posición (m,n).

Corrección 2: Pseudocódigo

ACTUALIZAR_MEMO_OBTENER_GANANCIA_CAMINO_MAXIMO(n, m, ganancia_max_actual, max_ganancia_este_oeste, max_ganancia_norte_sur, camino_este_oeste, camino_norte_sur, camino_actual, memo, manzanas):

```
Si ganancia_max_actual > max_ganancia_norte_sur y ganancia_max_actual > max_ganancia_este_oeste:
    memo[n][m] = ganancia_max_actual
    manzanas[n][m] = camino_actual
    Retornar ganancia_max_actual, manzanas[n][m]
Sino si ganancia_max_actual < max_ganancia_norte_sur y max_ganancia_este_oeste <= max_ganancia_norte_sur:
    memo[n][m] = max_ganancia_norte_sur
    manzanas[n][m] = camino_norte_sur
    Retornar max_ganancia_norte_sur, camino_norte_sur
Sino:
    memo[n][m] = max_ganancia_este_oeste
    manzanas[n][m] = camino_este_oeste
    Retornar max_ganancia_este_oeste, camino_este_oeste
```

OBTENER_MAXIMO_GANANCIA_Y_CAMINO(matriz, n, m, max_ganancia_este_oeste, max_ganancia_norte_sur, camino_este_oeste, camino_norte_sur, ganancia_actual, memo, manzanas):

```
  camino_actual = []
  ganancia_max_actual = ganancia_actual
  Para cada celda (i, j) desde (0, 0) hasta (n, m):
    Si el valor en matriz[i][j] > ganancia_actual:
      ganacia_acumulada = memo[i][j] + ganancia_actual
      Si ganacia_acumulada > ganancia_max_actual:
        Actualizar ganancia_max_actual = ganacia_acumulada
        camino_actual = manzanas[i][j]
  Añadir (n + 1, m + 1) a camino_actual
  Retornar ACTUALIZAR_MEMO_OBTENER_GANANCIA_CAMINO_MAXIMO(n, m,
ganancia_max_actual, max_ganancia_este_oeste, max_ganancia_norte_sur,  camino_este_oeste,
camino_norte_sur, camino_actual, memo, manzanas)
```

```
OBTENER_GANANCIA_Y_CAMINO_MAXIMO_POSICION_ACTUAL(ganancia, n, m,
memo, manzanas):
  memo[n][m] = ganancia
  manzanas[n][m] = [(n + 1, m + 1)]
  Retornar ganancia, manzanas[n][m]
```

```
MAX_GANANCIA(matriz, n, m, memo, manzanas):
  Si GANANCIA_ESTA_EN_MEMO(memo, n, m):
    Retornar memo[n][m], manzanas[n][m]

  Si ES_POSICION_INICIAL(n, m):
    Retornar OBTENER_GANANCIA_CASO_BASE(matriz, memo, manzanas)

  ganancia_actual = matriz[n][m]
  max_ganancia_este_oeste = max_ganancia_norte_sur  = -1
  camino_este_oeste = camino_norte_sur  = []

  Si TENEMOS_MANZANAS_ESTE_OESTE:
    max_ganancia_este_oeste, camino_este_oeste = MAX_GANANCIA(matriz, n, m - 1, memo,
manzanas)

  Si TENEMOS_MANZANAS_NORTE_SUR:
    max_ganancia_norte_sur, camino_norte_sur = MAX_GANANCIA(matriz, n - 1, m, memo,
manzanas)
```

Si max_ganancia_este_oeste < ganancia_actual y max_ganancia_norte_sur <= ganancia_actual:
Retornar **OBTENER_GANANCIA_Y_CAMINO_MAXIMO_POSICION_ACTUAL**(ganancia_actual, n, m, memo, manzanas)
Sino:
Retornar **OBTENER_MAXIMO_GANANCIA_Y_CAMINO**(matriz, n, m, max_ganancia_este_oeste, max_ganancia_norte_sur, camino_este_oeste, camino_norte_sur, ganancia_actual, memo, manzanas)

Corrección 3: Análisis de complejidad pseudocódigo

OBTENER_MAXIMO_GANANCIA_Y_CAMINO:

- Esta función contiene dos bucles anidados que recorren desde (0, 0) hasta (n, m), resultando en una complejidad de O(n * m)

ACTUALIZAR_MEMO_OBTENER_GANANCIA_CAMINO_MAXIMO:

- Esta función tiene una complejidad de O(1) porque solo realiza comparaciones y actualizaciones en las matrices memo y manzanas.

La complejidad general no cambiará pero se agregó, la forma como se calcula la ganancia de forma más clara.

Corrección 4: Análisis de complejidad en código

Si bien se planteó teóricamente una complejidad de $O(n * m)$, en el desarrollo del código se usó una función que cambió la complejidad y es el método `copy.deepcopy()` que agrega un extra de complejidad de n+m pues este recorre todos los elementos del array para realizar la copia. El método max_ganancia() que es la función con mayor coste pues es acá donde recorremos la matriz y hacemos la copia de los elementos resultando en una complejidad temporal de $O(n^2 * m^2 * (n + m))$ y a la espacial seguimos teniendo la matriz pero debemos agregar el coste la copia resultando en una complejidad temporal de $O(n * m * (n + m))$.

Corrección 5: Corrección en código fuente

- Forma de ejecución:
  python tareas.py <cant-cuadras-norte-sur> <cant-cuadras-este-oeste> <archivo-ganancias>
- Se modularizar las funciones del programa
- Se corrigió la ejecución en el caso indicado

- Se cambió nombres de variables y funciones para mayor claridad

Parte 3:

Corrección 1: Definición de NP-H

La clase de complejidad NP-H comprende aquellos problemas a los cuales puede reducirse **polinomialmente** todo problema NP

Corrección 2: Prueba de certificador polinomial (inciso 2)

Para una solución al problema, consistente en

- Las *k* personas a eliminar.
- Los dos grupos de personas resultantes de esta eliminación.

se observa que es verificable *fácilmente* (es decir, en tiempo polinomial) comprobando que:

- *k* <= *j* que se puede realizar en $O(1)$.
- Ambos grupos tienen al menos una persona cada uno. Esto se puede realizar en O(1)
- Ninguna persona de un grupo tiene relación de amistad con alguien de otro grupo. Esto se puede realizar en $O(|V||E|)$.
- Ninguna persona está repetida en los dos grupos. Esto se puede realizar como parte del inciso anterior, está comprendida en su complejidad.

En las expresiones de arriba V son las personas y E las relaciones de amistad.

Queda así probado lo pedido.

Corrección 3: Prueba de que problema del casting es NP-H (inciso 3).

Se pide probar que resolver el problema del casting es NP-H, o sea de los problemas NP más difíciles. Para esto se puede partir de un problema Y $\in$ NP-C y reducirlo polinomialmente a X. Este problema Y está determinado por el enunciado en este caso (*Minimum node deletion bipartite subgraph*).

El problema *Minimum node deletion bipartite subgraph* tiene el siguiente enunciado:

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.

Es fácil de ver que el mismo se puede reducir polinomialmente al problema del casting realizando las siguientes consideraciones:

- k tiene el significado de j en el problema del casting
- Cada vértice v ∈ V representa un participante del show.
- Cada arista e (u, v) ∈ E representa una amistad entre los participantes u y v.

Es trivial que esto se puede realizar en tiempo polinomial (como mucho se requiere renombrar los vértices antes y después de encontrada la solución).

Corrección 4: Prueba de que *Minimum node deletion bipartite subgraph* es NP (inciso 4)

Dada una solución consistente en un grupo de vértices a remover para convertir al grafo en bipartito, se debe:

- Iterar la misma y remover cada elemento del conjunto V del grafo, junto con sus aristas asociadas. Esto se puede realizar con complejidad polinomial $O[k(|V| - 1)] = O(|V|)$ donde k es el número máximo de vértices que es posible remover como parte del enunciado del problema y $|V| - 1$ una cota del máximo posible de aristas que posee cada vértice.
- Para el grafo restante, aplicamos el siguiente algoritmo:

Marcamos todos los vértices como no visitados
Sea vertice_actual un vértice cualquiera que etiquetamos con *Grupo 1*.
Se realiza BFS desde vertice_actual:
    Para cada vértice que no está visitado, lo marcamos como visitado y lo etiquetamos con el grupo opuesto al propio.
    Para cada vértice ya visitado, verificamos que la etiqueta a asignar no difiere de la ya presente. Si difieren, el grafo no es bipartito.
    Se asigna nuevo vertice_actual entre los adyacentes al actual.

Este algoritmo se puede realizar en tiempo O (|V|+|E|). Por lo tanto la complejidad total es O(|V|+|E|), o O($|V|^2$) en el peor de los casos. Como la complejidad es polinomial, el problema es NP.

Corrección 5: Prueba de que *Minimum node deletion bipartite subgraph* es NP-H (inciso 4)

En segundo lugar se quiere verificar que el problema es NP-H. Partiendo del problema *Clique* ∈ NP-C para un grafo G_1 (V, E) y un valor k_1 entero positivo, se lo puede reducir polinomialmente al problema *Minimum node deletion bipartite subgraph* de la siguiente forma:

- Creando un grafo $G_2(V, \overline{E})$ donde los vértices son los mismos y el conjunto de aristas es el complemento del conjunto de aristas E de G_1
- Tomando un $k_2 = |V| - k_1$

La primera parte de esta reducción se puede hacer en $O[|V|(|V| - 1)] = O(|V|^2)$ o sea con complejidad polinomial. Es así porque se puede realizar obteniendo cada arista posible y eliminando aquellas presentes en E.

Para justificar esta equivalencia se realizan las siguientes observaciones:

- Un clique de tamaño k_1 equivale a un conjunto de k_1 nodos sin aristas **entre sí** en G_2 . Esto es fácil de ver pues si no hay ninguna arista entre k_1 nodos en G_2 , eso significa que tienen todas las aristas posibles (o sea es un clique de k_1 nodos) en G_1
- Esta falta de aristas entre nodos implica que al dividir los nodos en dos conjuntos sin aristas entre sí (como solución del problema del grafo bipartito) todos los nodos que pertenecen a un clique en G_1 estarán en el mismo conjunto de G_2. Luego sería necesario recorrer ambas partes para verificar que al menos una tiene la cantidad de elementos k_1 requeridos.
- Al permitir eliminar hasta $|V| - k_1$ nodos se contempla el caso extremo, esto es, todo un conjunto con exactamente k_1 nodos y otro vacío.

Luego de resolver el problema se dispone de un conjunto de nodos N a eliminar o bien se sabe que no hay solución posible. Si no hay solución posible para *Minimum node deletion bipartite subgraph*, tampoco la hay para *Clique*. Si se encontró una solución, hay que transformarla mediante el siguiente algoritmo:

1. Recorrer G_2 eliminando los vértices pertenecientes a N.
2. Recorrer los $|V| - |N|$ vértices restantes e ir dividiéndolos en los conjuntos según las aristas de cada uno. Se puede realizar fácilmente mediante BFS (ver **Corrección 4**).
3. Si uno de los conjuntos tiene al menos tamaño k_1, *Clique* tiene solución. En caso contrario no la tiene.

La segunda parte de esta reducción se puede realizar en $O(|V|^2)$ o bien $O(|V| + |E|)$ dependiendo de la representación de grafo utilizada (la parte más compleja es el borrado de vértices).

Por lo tanto, *Minimum node deletion bipartite subgraph* es NP-H.

Con estas dos deducciones (**Corrección 4 y 5**), se puede concluir entonces que *Minimum node deletion bipartite subgraph* es NP-C.

Corrección 6: Complejidad de X (inciso 7)

Por enunciado $X \leq_p CASTING$ y sabemos por los incisos anteriores que $CASTING \in NP - C$.
Únicamente se puede afirmar que X es al menos tan complicado como CASTING.

Observamos que

1. X no necesariamente es NP. Para afirmar esto sería necesario que toda solución de X sea reducible polinomialmente a una solución de CASTING, lo cual no está indicado en el enunciado (solo se sabe que se puede reducir toda instancia de X a una instancia de CASTING).
2. X no necesariamente es NP-H. Para afirmar esto sería necesario probar que todo problema NP puede reducirse a X. Mientras que todo problema $Y \in NP$ es reducible polinomialmente a CASTING por ser este NP-H, esto no implica relación alguna entre las complejidades concretas de X e Y.

Correcciones 2:

Parte 1:

Corrección 1: Ecuación de Recurrencia

$$\mathrm{OPT}[i,j] = \begin{cases} V[i,j] & \text{si } i = 0 \text{ y } j = 0 \\ \max\limits_{k \le i, l \le j, (k,l) \neq (i,j)} \{\mathrm{OPT}[k,l] + V[i,j]\} & \text{si existe } V[k,l] > V[i,j] \\ V[i,j] & \text{en otro caso} \end{cases}$$

Para que se entienda mejor modificamos los nombres de las variables en la ecuación de recurrencia. En nuestro código y en el pseudocódigo, OPT es memo y V[i,j] es matriz[i][j].

OPT[i,j] es la máxima ganancia posible que se puede obtener saliendo desde [i,j] hasta [0,0].

Si estamos en la posición base [0,0] nos quedamos con el valor de esa posición.

Si no estamos en la posición base tenemos que buscar el máximo OPT de todas las posiciones anteriores [k,l] donde el valor en la matriz es mayor al actual, sumándole el valor de la posición actual.

Si no encontramos ninguna posición anterior donde el valor es mayor al de posición actual, entonces nos quedamos con el valor actual V[i,j].

Parte 3:

Corrección 4: Prueba de que *Minimum node deletion bipartite subgraph* es NP (inciso 4)

Dada una solución consistente en un grupo de vértices a remover para convertir al grafo en bipartito, se debe:

- Iterar la misma y remover cada elemento del conjunto V del grafo, junto con sus aristas asociadas. Esto se puede realizar con complejidad polinomial$O[k(|V| - 1)] = O(|V|)$ donde k es el número máximo de vértices que es posible remover como parte del enunciado del problema y $|V| - 1$ una cota del máximo posible de aristas que posee cada vértice.
- Para el grafo restante, aplicamos el siguiente algoritmo:

    Marcamos todos los vértices como no visitados
    Mientras queden vértices sin visitar:
        Sea vertice_actual un vértice cualquiera que etiquetamos con *Grupo 1*.
        Se realiza BFS desde vertice_actual:
            Para cada vértice que no está visitado, lo marcamos como visitado y lo etiquetamos con el grupo opuesto al propio.
            Para cada vértice ya visitado, verificamos que la etiqueta a asignar no difiere de la ya presente. Si difieren, el grafo no es bipartito.
            Se asigna nuevo vertice_actual entre los adyacentes al actual.
    Una vez terminada la iteración de todos los vértices sin detectar etiquetas de distinto tipo en un mismo nodo, se puede afirmar que el grafo es bipartito.

Este algoritmo se puede realizar en tiempo O (|V|+|E|). Por lo tanto la complejidad total es O(|V|+|E|), o O($|V|^2$) en el peor de los casos. Como la complejidad es polinomial, el problema es NP.

Corrección 5: Prueba de que *Minimum node deletion bipartite subgraph* es NP-H (inciso 4)

En segundo lugar se quiere verificar que el problema es NP-H. Partiendo del problema *Clique* $\in$ NP-C para un grafo G_1 $(V,\ E)$ y un valor k_1 entero positivo, se lo puede reducir polinomialmente al problema *Minimum node deletion bipartite subgraph* de la siguiente forma:

- Creando un grafo $G_2(V, \overline{E})$ donde los vértices son los mismos y el conjunto de aristas es el complemento del conjunto de aristas E de G_1
- Adicionalmente, para cada nodo aislado de G_2 luego del primer paso, se agrega un nodo adyacente.
- Tomando un $k_2 = |V| - k_1$

La primera parte de esta reducción se puede hacer en $O[|V|(|V| - 1)] = O(|V|^2)$ o sea con complejidad polinomial. Es así porque se puede realizar obteniendo cada arista posible y eliminando aquellas presentes en E. Esta complejidad incluye además, la creación de nodos adyacentes a los aislados y sus correspondientes aristas.

Para justificar esta equivalencia se realizan las siguientes observaciones:

- Un clique de tamaño k_1 equivale a un conjunto de k_1 nodos sin aristas **entre sí** en G_2 . Esto es fácil de ver pues si no hay ninguna arista entre k_1 nodos en G_2 , eso significa que tienen todas las aristas posibles (o sea es un clique de k_1 nodos) en G_1
- Esta falta de aristas entre nodos implica que al dividir los nodos en dos conjuntos sin aristas entre sí (como solución del problema del grafo bipartito) todos los nodos que pertenecen a un clique en G_1 estarán en el mismo conjunto en la división bipartita de G_2.
- El agregado de nodos conectado cada uno únicamente a cada nodo aislado no afecta la resolución pero sí permite evitar resoluciones incorrectas en caso de que un nodo en G_2 no tenga ejes, y por lo tanto pueda pertenecer a cualquiera de las partes.
- Al permitir eliminar hasta $|V| - k_1$ nodos se contempla el caso extremo, esto es, todo un conjunto con exactamente k_1 nodos y otro vacío.

Luego de resolver el problema se dispone de un conjunto de nodos N a eliminar o bien se sabe que no hay solución posible. Si no hay solución posible para *Minimum node deletion bipartite subgraph*, tampoco la hay para *Clique*. Si se encontró una solución, hay que transformarla mediante el siguiente algoritmo:

4. Recorrer G_2 eliminando los vértices pertenecientes a N.
5. Recorrer los $|V| - |N|$ vértices restantes e ir dividiéndolos en los conjuntos según las aristas de cada uno. Se puede realizar fácilmente mediante BFS (ver **Corrección 4**).
6. Si uno de los conjuntos tiene al menos tamaño k_1, *Clique* tiene solución. En caso contrario no la tiene.

La segunda parte de esta reducción se puede realizar en $O(|V|^2)$ o bien $O(|V| + |E|)$ dependiendo de la representación de grafo utilizada (la parte más compleja es el borrado de vértices).

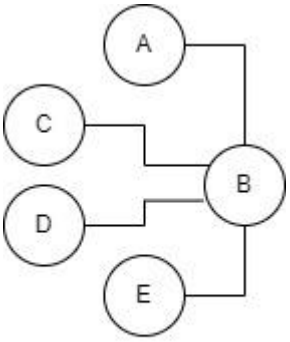
Por lo tanto, *Minimum node deletion bipartite subgraph* es NP-H.

Con estas dos deducciones (**Corrección 4 y 5**), se puede concluir entonces que *Minimum node deletion bipartite subgraph* es NP-C.

A continuación se ejemplifican algunos casos de uso:

Caso 1

Supongamos que se desea encontrar un *clique* de $k_1 = 3$ elementos en el siguiente grafo G_1



G_1

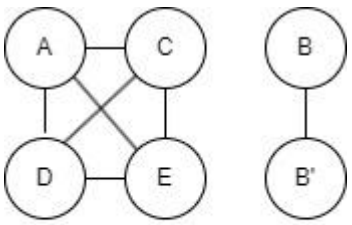
Partiendo del grafo G_1 mostrado arriba, se crea G_2 con los mismos vértices de G_1 pero ejes complementarios:

$\overline{E} = \{A - C, A - D, A - E, C - D, C - E, D - E\}$

Además, como B queda aislado (sin ejes) se agrega un nodo B' adyacente a este. Los ejes son

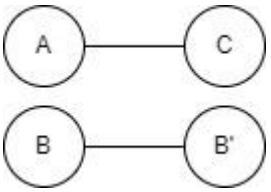
$E_{G2} = \{A - C, A - D, A - E, C - D, C - E, D - E, B - B'\}$

Resulta la siguiente instancia de *Minimum node deletion bipartite graph*, donde se permite eliminar hasta $k_2 = 5 - 3 = 2$ nodos



G_2

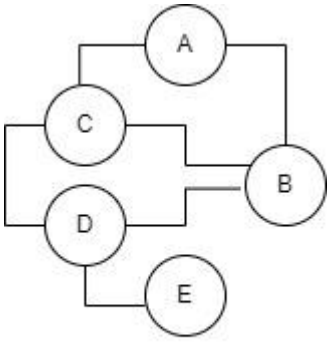
Se observa que debido al conjunto {A, C, D, E} no es posible que el grafo sea bipartito (si por ejemplo, A pertenece al grupo 1, necesariamente C debe pertenecer al grupo 2, pero luego ni D ni E pueden pertenecer a ninguno de los dos grupos. Entonces es necesario eliminar al menos 2 nodos. Eliminando los mismos se puede efectuar la siguiente bipartición del grafo:



Sin embargo, como ninguna posee 3 elementos, no existe la solución pedida.

Caso 2

Supongamos que se desea encontrar un *clique* de $k_1 = 4$ elementos en el siguiente grafo G_1



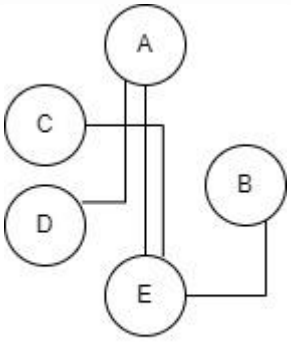
G_1

Partiendo del grafo G_1 mostrado arriba, se crea G_2 con los mismos vértices de G_1 pero ejes complementarios:

$\overline{E} = \{A - D, A - E, C - E, B - E\}$

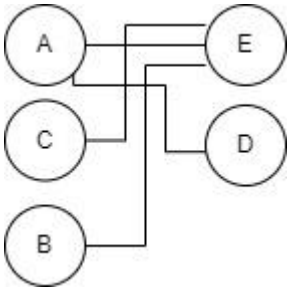
No resultan vértices aislados así que esos son todos los ejes requeridos por el algoritmo.

Resulta la siguiente instancia de *Minimum node deletion bipartite graph*, donde se permite eliminar hasta $k_2 = 5 - 4 = 1$ nodo



G_2

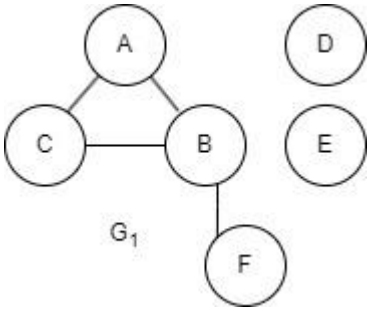
En este caso es posible encontrar una solución sin eliminar nodo alguno:



Sin embargo, como ninguna de las partes tiene al menos 4 elementos, tampoco hay solución.

Caso 3

Supongamos que se desea encontrar un *clique* de $k_1 = 3$ elementos en el siguiente grafo G_1

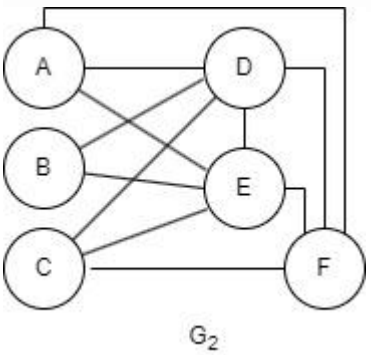


Partiendo del grafo G_1 mostrado arriba, se crea G_2 con los mismos vértices de G_1 pero ejes complementarios:

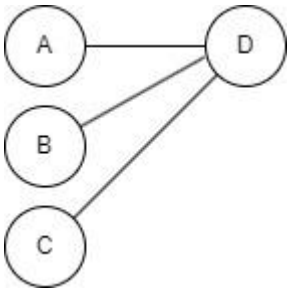
$\overline{E} = \{A - D, A - E, A - F, B - D, B - E, C - D, C - E, C - F, D - F, D - E, E - F\}$

No resultan vértices aislados así que esos son todos los ejes requeridos por el algoritmo.

Resulta la siguiente instancia de *Minimum node deletion bipartite graph*, donde se permite eliminar hasta $k_2 = 6 - 3 = 3$ nodos



Se puede obtener un grafo bipartito eliminando E y F (obsérvese que eliminando solo 1 nodo no es posible pues para cualquier elección sigue habiendo cliques de 3).



Como una posee 3 elementos, existe la solución pedida para el problema clique.