Universidad de Buenos Aires
Facultad de Ingeniería
1er cuatrimestre 2023

Teoría de Algoritmos I (75.29)

TRABAJO PRÁCTICO Nº3
TEMA: Redes de flujo y problemas NP completos
FECHA:10/05/2023
GRUPO: Dijkstra fan club
INTEGRANTES:

Ménager, Jeanne - #2255240
<jeannemenager3107@gmail.com>
Rico, Mateo Julián - #108127
<mricofi.uba.ar>
Ramirez Scarfiello, Nicolás Alberto - #102090
<nramirez@fi.uba.ar>
Civini, Armando Tomás - #104350
<acivini@fi.uba.ar>
Puglisi, Agustín - #104245
<apuglisi@fi.uba.ar>
Zuretti, Agustin Santiago - #95605
<azuretti@fi.uba.ar>

Índice

Parte I
Primera entrega

1. Parte 1: Flujo máximo con demandas

1.1. Resolución del problema

Para solucionar el problema de flujo máximo con demandas nos basamos en el algoritmo presentado en un apunte[1] de otra universidad.

Dado un grafo $G = (V, E)$ que representa nuestra red de flujo, en el cual cada eje $e \in E$ tiene una capacidad $c(e)$ y una demanda $l(e)$, el algoritmo consiste en aplicarle una transformación a G para hallar un flujo válido (satisfacer las demandas) y luego aplicarle el algoritmo de Ford-Fulkerson con una pequeña modificación para maximizar el flujo.

Para transformar a G lo que hacemos es a cada eje e le asignamos una capacidad $c(e) - l(e)$. Luego, agregamos el eje (s, t) con capacidad $c(s, t) = \infty$. El siguiente paso es agregar los vértices s' y t' que actúan como una "super fuente" y "super sumidero" respectivamente. Agregamos ejes (s', v) para todo v tal que

$$L = \sum_{w \in V} l(v, w) - \sum_{u \in V} l(u, v) > 0$$

con capacidad $c(s', v) = L$ es decir, conectamos la super fuente con aquellos vértices cuya "demanda entrante" es menor a la "demanda saliente". Análogamente agregamos ejes (v, t') para todo v tal que

$$L = \sum_{w \in V} l(v, w) - \sum_{u \in V} l(u, v) < 0$$

con capacidad $c(v, t') = L$, es decir, a aquellos vértices cuya demanda entrante sea mayor a la saliente los conectamos con el super sumidero t'. El siguiente paso es aplicarle Ford-Fulkerson a este grafo transformado, y quedarnos con el grafo residual $G' = (V, E')$. Para cada eje en retroceso $e' = (v, u)$ en el grafo residual, a su capacidad $c(e')$ le sumamos la demanda $l(e)$ del eje en adelanto del grafo original. Ahora en los ejes en retroceso de G' tenemos el flujo válido. El siguiente paso es modificar las capacidades de los ejes de G. Para cada $e = (u, v)$ en el grafo original G, le asignamos una capacidad $c(e) = c(e) - c(e')$, donde e' es el eje en retroceso (v, u) de G', y agregamos un eje en retroceso (v, u) en G, cuya capacidad es $c(e') - l(e)$. Luego de estas modificaciones aplicamos Ford-Fulkerson a este grafo para maximizar el flujo total. Por último, a la capacidad cada eje del grafo residual resultante de Ford-Fulkerson le sumamos la demanda del eje en retroceso del grafo original G y devolvemos el grafo residual, el cual contiene el flujo total.

1.2. Pseudo-Código

1.2.1. Ford-Fulkenson

Para este algoritmo, se recibe como entrada una matriz G de adyacencia del grafo, donde los valores son las capacidades de los ejes. También se recibe cual nodo es la fuente s, y cual nodo es el sumidero t. (ignorar el =0 de los returns).

Algorithm 1 Ford-Fulkenson G, s, t

while Existe un camino p:$s \to t$ en G **do**
  $maxCap \leftarrow min\{G(u,v) : (u,v) \in p\}$
  for $(u,v) \in p$ **do**
    $G(u,v) \leftarrow G(u,v) - maxCap$
    $G(v,u) \leftarrow G(v,u) + maxCap$
  end for
end while
return G =0

1.2.2. Camino valido con capacidad mínima

Para este algoritmo asumimos que la matriz de adyacencia ahora cuenta con dos valores, la capacida máxima y la capacidad mínima del eje correspondiente.

Algorithm 2 ValidFlow G, s, t, *nodos*

```
# calculo lo que produce o demanda cada nodo y
# le resto a la capacidad maxima la capacidad minima
for u ∈ nodos do
  for v ∈ nodos do
    produce(u)+ = G(u, v).minCap
    produce(v)− = G(u, v).minCap
    G(u, v).maxCap− = G(u, v).minCap
  end for
end for
G(s, t).maxCap ← Inf
# agrego la super fuente y el super sumidero
# y los conecto con los nodos demandantes y productores
agregarnodo(G, s′)
agregarnodo(G, t′)
for u, valor ∈ produce do
  if valor < 0 then
    G(s′, u).maxCap ← −valor
  end if
  if valor > 0 then
    G(u, t′).maxCap ← valor
  end if
end for
G_valido ← Ford − Fulkenson(G, s′, t′)
# volvemos a agregar las capacidades minimas
# que le habiamos restado, se agregan en el eje inverso
for u ∈ nodos do
  for v ∈ nodos do
    G_valido(v, u).maxCap+ = G(u, v).minCap
  end for
end for
return G_valido =0
```

1.2.3. Flujo máximo con capacidad mínima

Algorithm 3 MaxFlow G, s, t, *nodos*

$G_{valido} \leftarrow ValidFlow(G, s, t, nodos)$
for $(u, v) \in G$ **do**
  # recordar que los ejes de flujo en el G_{valido} estan invertidos por
  # Ford-Fulkenson
  $G(u, v).maxCap- = G_{valido}(v, u).maxCap$
  $G(v, u).maxCap \leftarrow G_{valido}(v, u).maxCap - G(u, v).minCap$
end for
# resuelvo Ford-Fulkenson con la nueva matriz
$G_{FF} \leftarrow FordFulkenson(G, s, k)$
# vuelvo a agregar la capacidad minima que sacamos
for $(u, v) \in G$ **do**
  $G_{FF}(u, v).maxCap+ = G(v, u).minCap$
end for
return G_{FF} =0

Finalmente se puede calcular el flujo maximo a partir de todos los caminos que llegan al sumidero.

Recordar que el algoritmo de Ford-Fulkenson retorna el flujo máximo en los ejes opuestos al eje original.

1.3. Ejemplo

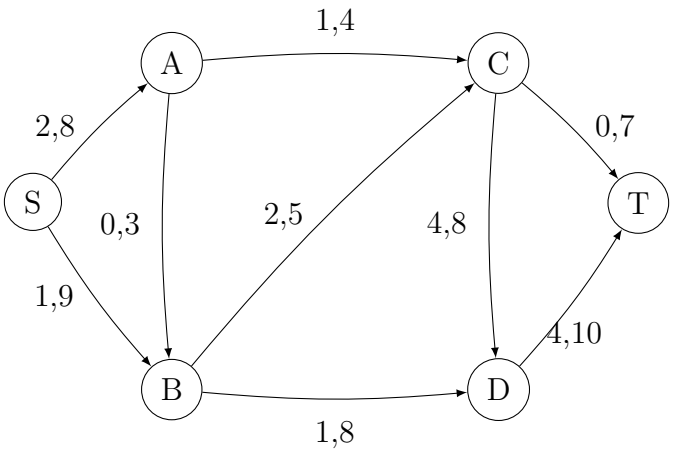
1.3.1. Enunciado

El gran comandante Stalin produce medallas para su codedorado ejercito, producir las medallas tiene, un punto de donde viene la materia prima, dos etapas de producción y todas las medallas tienen que ser enviadas al mismo lugar. Para cada una de las dos etapas hay dos fabricas que pueden realizar este trabajo, A y B para la primera etapa, y C y D para la segunda etapa. También puede ocurrir que se envien medallas de A a B o de C a D y para A las medallas solo pueden viajar a C.

Además los maravillosos caminos sovieticos solo soportan una cantidad máxima de toneladas de medallas circulando por ellos para no colapsar. Como la USSR es la tierra de la igualdad, Stalin quiere que por lo menos haya cierta cantidad de trabajo en cada fabrica y la mejor manera de regular esto es obligando a que circule una cierta cantidad mínima de medallas obligatoriamente por cada camino. Valores que orgullosamente establecio él.

1.3.2. Brinde un ejemplo paso a paso donde se aprecie el funcionamiento de toda su propuesta

Sea un grafo G con limites inferior.



Agregamos dos vertices S' y T'.

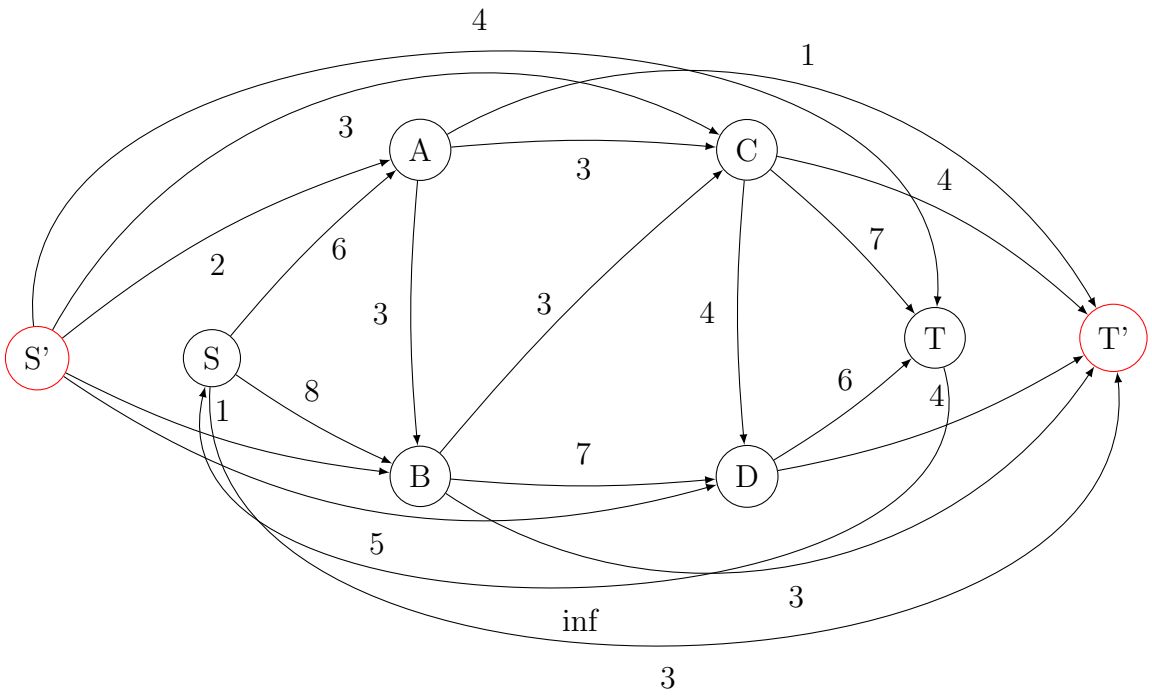
Por cada vertice v, agregamos un eje de S' a v con capacidad igual a la suma de los demandes de los ejes que llegan a v.

Por cada vertice v, agregamos un eje de v a T' con capacidad igual a la suma de los demandes de los ejes que salen de v.

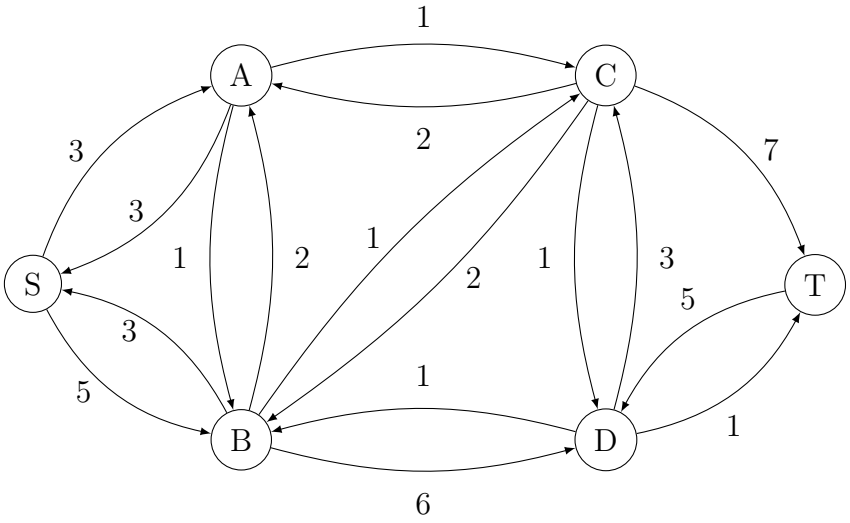
Por cada eje u-¿v, cambiamos su capacidad en su capacidad menos su demanda.

Por fin, agrefamos un eje de S a T con capacidad infinito.

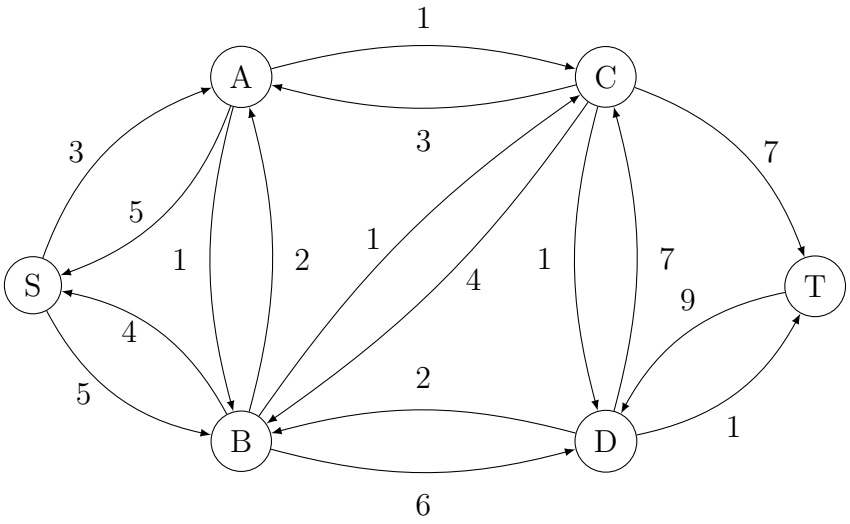
El resultado es este gráfo :



Usamos el teorema de Ford-Fulkerson para encontrar un flujo válido, encontramos ese grafo :

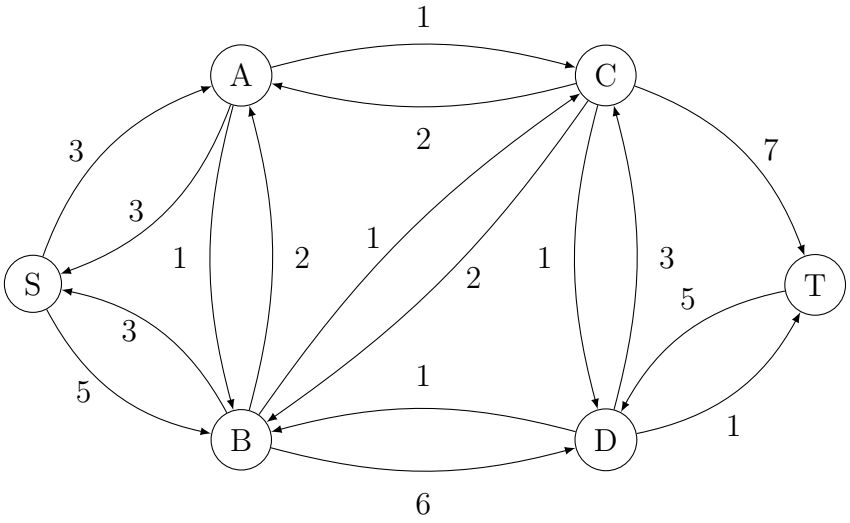


Agregar la capacidad minima que sacamos antes a cada eje del nuevo grafo. Así obtenamos el grafo valido :

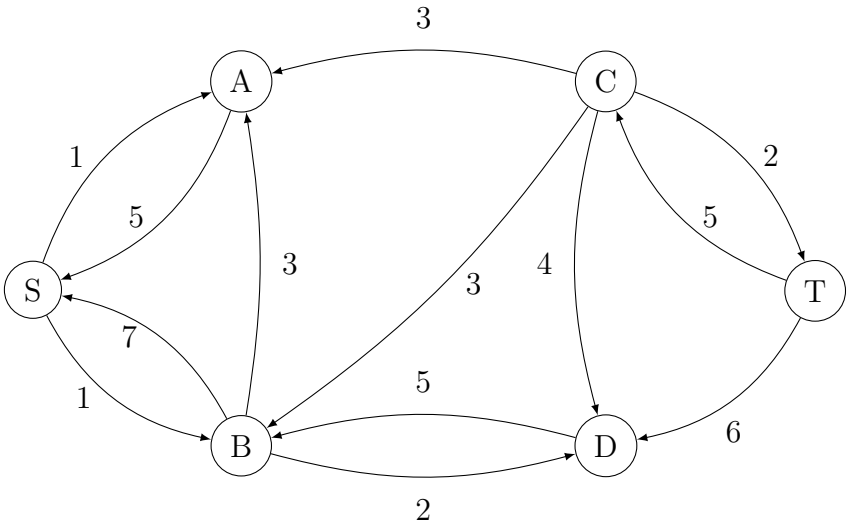


Cambiamos los ejes de G gracias al grafo valido y agregamos ejes tambien.

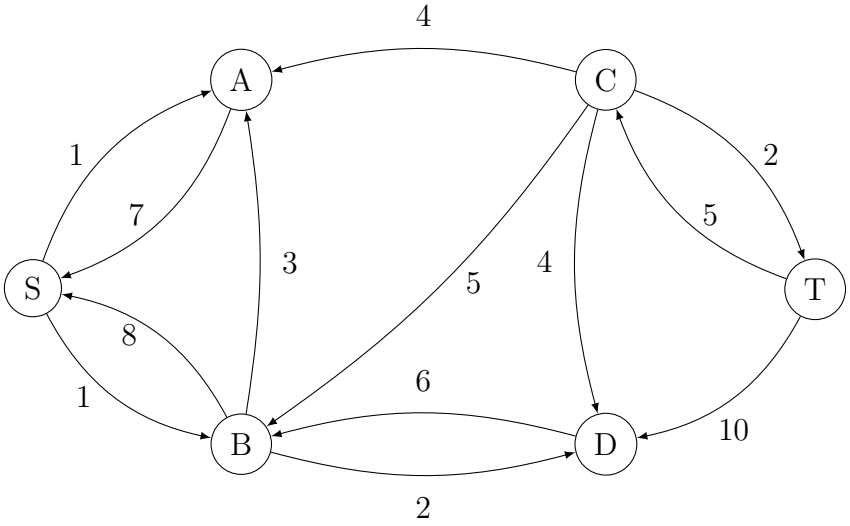
A los ejes (u,v) que ya existen en el grafo, les restamos las capacidades máximas de los aretes (v,u) del grafo válido. Añadimos ejes (v,u) en G que tengan como valores las capacidades máximas de los aretes (v,u) del grafo válido menos las demandas de los aretes (u,v) en G.



Usamos el teorema de Ford-Fulkerson para encontrar un flujo maximo, encontramos ese grafo :



Vuelvemos a agregar la capacidad minima que sacamos :



Entonces, el flujo maximal es de 15.

1.4. Complejidad

1.4.1. Pseudocódigo

Este algoritmo se compone en tres partes: Ford-Fulkenson, Aplicar la transformada para obtener un flujo válido y aplicar la transformada para poder calcular el flujo máximo. La complejidad del algoritmo de Ford-Fulkenson es conocida y es $O(E * c)$ en cuanto a lo espacial, siendo c la suma de las capacidades máximas y E la cantidad de ejes. Además la complejidad espacial en nuestro caso sería $O(V^2)$ ya que implementamos la estructura de datos con una matriz de adyacencia.

La complejidad de la transformación para obtener un flujo válido es de $O(V^2)$ ya que para calcular la producción o demanda de cada uno de los nodos, tenemos que recorrer todas sus conexiones. Para obtener el flujo válido aplicamos Ford-Fulkenson por lo cual la complejidad del algoritmo resulta $O(E * c)$ ya que concideramos que concideramos que esta es de un mayor orden. En cuanto a complejidad espacial, en este algoritmo se hace uso de un vector con tamaño V pero sigue siendo de mayor orden la complejidad espacial usada para representar el grafo, $O(V^2)$.

Finalmente la complejidad de la transformada para obtener el flujo máximo es de $O(E)$ ya que tenemos que recorrer todos los ejes. De todas maneras al tener que aplicar Ford-Fulkenson luego, este va a seguir siendo la mayor complejidad temporal.

En conclusión la complejidad temporal del pseudocódigo es $O(E * c)$ y la complejidad espacial de $O(V^2)$.

1.4.2. Código

Para el código mantuvimos la misma complejidad que la analizada para el pseudocódigo. Esto es gracias a que usamos exclusivamente arreglos como nuestras estructuras de datos por lo que mantienen la complejidad teórica de las operaciones. Además para encontrar los caminos en Ford-Fulkenson hicimos uso de BFS lo que mantiene la complejidad del algoritmo.

Para ejecutar el código ejecute el archivo *main.py* y pasele como argumento el *path* del archivo donde se encuentra la red de flujo.

2. Parte 2: La separación en grupos compatibles

2.1. Análisis Teorico

Se conoce como P, conjunto de problemas de decisión para los que existe un algoritmo que lo resuelve de forma eficiente. Decimos que un algoritmo resuelve eficientemente un problema, si para toda instancia del problema, encuentra la solución en tiempo polinomial. Por otra parte se conoce como NP al conjunto de problemas de decisión para los que existe un algoritmo que los certifique en forma eficiente. Entonces podemos preguntarnos Si P = NP. Esto significaria que para cualquier problema que podamos verificar una solución en tiempo polinómico, también podriamos resolverlo en tiempo polinómico. Esto implicaría que resolver

un problema tiene la misma complejidad que verificarlo y viceversa. Esto ultimo se conoce como el problema P vs NP, y al dia de hoy es un problema sin resolver en ciencias de la computacion.

Para definir los problemas NP-Har, observemos lo siguiente. Si tomamos un problema X, para el cual existe un problema Y, pertenciente a NP y que certifica X, entonces X, pertenece a NP-Hard.Podemos decir entonces que llamamos NP-Hard a cualquier problema X que es al menos igual de dificil que cualquier problema en NP. Sin embargo hay que destacar que los problemas NP-Hard, no necesariamente tienen que pertenecer a NP. Luego si un problema pertenece a NP-Hard Y NP al mismo tiempo decimos que es NP-C. Estos problemas son los mas dificiles de resolver dentro de NP.

2.2. Certificación del problema

Para que la certificación de un problema sea eficiente, debe ocurrir que para cada instancia del problema, este se debe poder verificar en tiempo polinomial. El algoritmo que cumpla con esto debe recibir una instancia del problema y un certificado del mismo, y debe devolver si el certificado cumple o no.

En el caso del problema, la instancia seria por ejemplo, una lista de todos los empleados que son compatibles entre si. Dado el certificado con la solución propuesta, la empresa solo debe buscar si para cada grupo propuesto, los empleados son compatibles dentro de este. Es decir, si se encuentran juntos en una fila de la lista que recibió como parámetro. Es sencillo observar que la solucion a este problema es polinómica y por lo tanto su verificacion eficiente.

2.3. Resolución del Problema

El problema de "clique cover" consiste en encontrar la menor cantidad de cliques (subconjunto de vértices que estén todos conectados entre sí), que cubran todos los vértices de un grafo no dirigido.

Sabemos que el problema de clique cover es NP-C. Esto significa que si encontramos una solución eficiente para resolver un problema NP-C, podríamos resolver eficientemente todos los problemas de la clase NP, además estos problemas pueden ser verificados en tiempo polinómicos.

Para demostrar que el problema presente en este TP de grupos cumpatibles no es fácil de resolver, podríamos intentar reducirlo al problema de clique cover.

Para ello, podríamos construir un grafo, en donde los nodos representen a los empleados, y las aristas refieran a las compatibilidades presentes en la planilla.

De esta forma consideramos que si dos empleados son compatibles, habrá una arista entre ellos. Por lo tanto el problema original se reduce a buscar la menor cantidad de cliques, en donde cada subconjunto de vértices que estén completamente conectados representará un grupo de trabajo.

En conclusión, este problema es al menos tan difícil como el problema de clique cover.

2.4. Demostración de que clique cover pertenece NP-C

El "k coloreo de grafos" es un problema donde teniendo un máximo de k colores, se busca asignar un color a cada vértice de un grafo, de manera tal que dos vértices adyacentes no tengan el mismo color.

Sabiendo que este problema es NP-C, de manera similar a lo hecho anteriormente si pudiéramos reducir desde este problema al problema clique cover, este último también pertenecería a NP-C.

Comencemos demostrando que el problema clique cover es NP. Esto significa que existe un algoritmo de verificación que verifica si una instancia de solución candidata es correcta en tiempo polinómico.

Sea G un grafo, y C un conjunto de cliques dado.

Podemos verificar si el conjunto de cliques cubren todos los nodos de G, comprobando que cada uno de los nodos esté incluido en al menos una de las cliques.

Por lo tanto clique cover es NP.

Ahora demostraremos que cualquier problema en NP se puede reducir a clique cover en tiempo polinómico.

- Dado un grafo G y un número k.
- Construimos un grafo G' que tenga el mismo conjunto de nodos que G, pero con las aristas invertidas. Es decir que si en G hay una arista entre dos vértices u y v, en G' no habrá una arista en u y v.
- Cada conjunto independiente en G' se vuelve un clique.
- Si encontramos una solución en G' podemos utilizar esa solución para obtener una asignación de colores que sea válida en el grafo G. Porque cada clique en G' corresponde a un conjunto de vértices en G que pueden ser coloreados por un mismo color.
- Por lo que si encontramos una clique cover en G', podríamos colorear el grafo G con k colores, ya que si el grafo es coloreable, lo podemos dividir en k conjuntos independientes.
- En caso contrario, no podemos colorear el grafo G original, porque si pudiéramos, cada conjunto de vértices coloreados con un mismo color formaría una clique en G'.
- De forma inversa, si G' puede ser cubierto por k cliques, entonces G tiene un partición en k conjuntos independientes, y por lo tanto es k-coloreable.

Al reducir un problema NP-C al problema de clique cover, queda demostrado entonces que este último también pertenece a NP-C.

2.5. Transitividad y NP-C: Caso del empleado de recursos humanos

La transitividad es una propiedad derivada de las reducciones polinomiales que nos sirve para establecer complejidades y/o acotar problemas. Según la misma,

dado un problema A que se puede reducir polinomialmente a un problema B, y dado un problema B que se puede reducir polinomialmente a un problema C, podemos afirmar que A se puede reducir polinomialmente a C.

Por otro lado, se define la clase de complejidad NP-Complete (o NP-C o NP-Completo) a aquellos problemas que pertenecen a NP y también forman parte de NP-Hard. Es decir a aquellos problemas de decisión que perteneciendo al conjunto de NP corresponden a los más difíciles entre ellos.

Suponiendo según dice el enunciado, que tenemos un método eficiente para responder el pedido cualquiera sean los empleados, compatibilidades y grupos de trabajo quiere decir que, al menos un problema de los más difíciles (en NP-C) se puede resolver en tiempo polinomial. Entonces podemos afirmar que todos los problemas en NP se podrían resolver en tiempo polinomial. La consecuencia resultante de esto sería que P = NP.

Por último, si un tercer problema al que llamaremos X se puede reducir polinomialmente al problema de grupos compatibles, entonces podemos aplicar la transitividad para así reducir polinomialmente X a NP-C. Esto ya que X se reduce polinomialmente a Grupos compatibles, y luego Grupos compatibles se reduce polinomialmente a NP-C. Podemos concluir finalmente que X se reduce polinomialmente a NP-C y calcular así su complejidad.

Parte II
Segunda entrega

1. Comentarios sobre las correcciones

1.0.1. Parte 1

La carátula no incluye los datos de los alumnos.

Presentan una propuesta correcta para resolver el problema. La explicación es clara. Incluyen pseudocódigo de la solución propuesta.

Desarrollan correctamente el paso a paso de un ejemplo de funcionamiento del algortimo.

No incluyen varios puntos pedidos: proponer un enunciado que se pueda resolver reduciéndolo al problema dado, como realizaría la reducción, análisis de la complejidad algorítmica (teórica y real).

Programa Parte 1: No incluyen instrucciones de ejecución, incluyen un ejemplo de prueba. Tuve que comentar una línea del código para que no corra un ejemplo por default. Más allá de eso, el programa funciona correctamente. Devuelve lo pedido de forma clara. Pasa las pruebas de cátedra.

1.0.2. Parte 2

La explicación de las clases de complejidad y sus relaciones es bastante acertada. La definición de NP-H no es correcta y no se termina de entender que quiere decir.

Explican correctamente la certificación en tiempo polinomial.

Realizan incorrectamente la reducción polinómica. Si reducen el problema de "grupos compatibles.ª çlique cover.estan afirmando que çlique cover.es al menos tan difícil que "grupos compatibles". Tomaron un problema cuya complejidad no conozco y lo transformaron en uno NP-C, esto no significa que no tenga una manera eficiente de resolver "grupos compatibles".

La demostración de por qué çlique cover.es NP-C es correcta. Aquí correctamente reducen el problema de çoloreo de grafos.ª çlique cover". Recuerden que esta reducción se hace para demostrar que nuestro problema es NP-H (NP-C viene por ser NP también).

No terminan la oración en el punto 2.5.

Conlcuyen correctamente que demostrar que un problema NP-C pertence a NP produciría que P=NP.

Nuevamente confunden el orden de la reducción polinomial y que implica. Consideren que yo perfectamente puedo reducir un problema X que pertence a P a un problema NP-H. Me estaría haciendo la tarea más difícil, pero no tengo ningun impedimento en hacerlo. No significa que ahora X pase a ser NP-H.

2. Parte 1: Flujo máximo con demandas

2.1. Enunciado

El gran comandante Stalin produce medallas para su codedorado ejercito, producir las medallas tiene, un punto de donde viene la materia prima, dos etapas de producción y todas las medallas tienen que ser enviadas al mismo lugar. Para cada una de las dos etapas hay dos fabricas que pueden realizar este trabajo, A y B para la primera etapa, y C y D para la segunda etapa. También puede ocurrir que se envien medallas de A a B o de C a D y para A las medallas solo pueden viajar a C.

2.2. Instrucciones de ejecución

Para ejecutar el código ejecute el archivo *main.py* y pasele como argumento el *path* del archivo donde se encuentra la red de flujo.

2.3. Complejidad

2.3.1. Pseudocódigo

Este algoritmo se compone en tres partes: Ford-Fulkenson, Aplicar la transformada para obtener un flujo válido y aplicar la transformada para poder calcular el flujo máximo. La complejidad del algoritmo de Ford-Fulkenson es conocida y es $O(E * c)$ en cuanto a lo espacial, siendo c la suma de las capacidades máximas y E la cantidad de ejes. Además la complejidad espacial en nuestro caso sería $O(V^2)$ ya que implementamos la estructura de datos con una matriz de adyacencia.

La complejidad de la transformación para obtener un flujo válido es de $O(V^2)$ ya que para calcular la producción o demanda de cada uno de los nodos, tenemos que recorrer todas sus conexiones. Para obtener el flujo válido aplicamos Ford-Fulkenson por lo cual la complejidad del algoritmo resulta $O(E * c)$ ya que concideramos que concideramos que esta es de un mayor orden. En cuanto a complejidad espacial, en este algoritmo se hace uso de un vector con tamaño V pero sigue siendo de mayor orden la complejidad espacial usada para representar el grafo, $O(V^2)$.

Finalmente la complejidad de la transformada para obtener el flujo máximo es de $O(E)$ ya que tenemos que recorrer todos los ejes. De todas maneras al tener que aplicar Ford-Fulkenson luego, este va a seguir siendo la mayor complejidad temporal.

En conclusión la complejidad temporal del pseudocódigo es $O(E * c)$ y la complejidad espacial de $O(V^2)$.

2.3.2. Código

Para el código mantuvimos la misma complejidad que la analizada para el pseudocódigo. Esto es gracias a que usamos exclusivamente arreglos como nuestras estructuras de datos por lo que mantienen la complejidad teórica de las operaciones.

Además para encontrar los caminos en Ford-Fulkenson hicimos uso de BFS lo que mantiene la complejidad del algoritmo.

2.4. Parte 2: La separación en grupos compatibles

2.4.1. Corrección: Análisis Teorico

Se conoce como P, conjunto de problemas de decisión para los que existe un algoritmo que lo resuelve de forma eficiente. Decimos que un algoritmo resuelve eficientemente un problema, si para toda instancia del problema, encuentra la solución en tiempo polinomial. Por otra parte se conoce como NP al conjunto de problemas de decisión para los que existe un algoritmo que los certifique en forma eficiente. Entonces podemos preguntarnos Si P = NP. Esto significaria que para cualquier problema que podamos verificar una solución en tiempo polinómico, también podriamos resolverlo en tiempo polinómico. Esto implicaría que resolver un problema tiene la misma complejidad que verificarlo y viceversa. Esto ultimo se conoce como el problema P vs NP, y al dia de hoy es un problema sin resolver en ciencias de la computacion.

Ahora analizaremos los problemas de tipo NP-C.

Sea X un problema de decisión, X es NP-C si:

- X es NP
- Todo problema NP se puede reducir a X en tiempo polinomial, y por lo tanto es también NP-H.

Por lo que, en resumen, P es un subconjunto de NP ya que P está incluido en NP.

Los problemas de tipo NP-C son la intersección entre los problemas de tipo NP y NP-H.

Y un problema NP-H no puede ser P.

2.4.2. Corrección: Resolución del Problema

El problema de "clique cover" consiste en encontrar la menor cantidad de cliques (subconjunto de vértices que estén todos conectados entre sí), que cubran todos los vértices de un grafo no dirigido.

El problema presentado en el presente trabajo llamado "grupos compatibles"pertenece a la clase NP-C.

Para demostrar esto, primero justificaremos que es NP. Para verificar una instancia de solución, por cada empleado podemos verificar si se encuentra en exactamente un grupo y además no hay conflictos en el grupo en el que está. Esto se puede realizar en tiempo polinómico.

Para justificar que pertenece a NP-C de manera similar a lo hecho anteriormente, podemos reducir el problema de clique cover a grupos compatibles de la siguiente forma:

- Sea G un grafo y k un número de entrada para el problema de clique cover.

- Creamos una instancia I del problema de grupos compatibles.
- Cada nodo en G será un empleado, y cada par de aristas (u, v) de G, establecemos una compatibilidad entre los empleados correspondientes en el problema grupos comaptibles.
- Establecemos k como el número máximo de grupos de trabajo.
- Si consideramos cada grupo como una clique en el grafo G, cubrimos todos los nodos en G con cliques, por lo que resolvimos el problema de clique cover.
- De la misma forma si encontramos el menor número de cliques para cubrir todos los nodos de G, encontramos una partición de los empleados en k grupos donde cada grupo cumple con la restricción de compatibildiad.

Por lo que al reducir el problema de clique cover al presentado en el TP, y demostrar que una solución válida para uno implica una solución valida para el otro concluimos que el problema de grupos compatibles es NP-C.

2.4.3. Un tercer problema al que llamaremos X se puede reducir polinomialmente al problema de "grupos compatibles", qué podemos decir acerca de su complejidad?

Dado que se ha demostrado que el problema de "grupos compatibles" es NP-C, si tenemos otro problema X tal que $X \leq_p$ "grupos compatibles", es decir, X se puede reducir polinomialmente a "grupos compatibles", entonces se puede concluir que X es a lo sumo tan difícil de resolver como "grupos compatibles". Esto es así porque dada una instancia I del problema X, puedo resolverla reduciéndola a una instancia de "grupos compatibles". Sin embargo esto no implica que no exista un algoritmo que resuelva X de forma más eficiente.

Referencias

[1] Erickson, J. (2015). Algorithms - Maximum flow extensions
https://courses.engr.illinois.edu/cs498dl1/sp2015/notes/25-maxflowext.pdf