FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 3 — Redes de Flujo

[75.29/95.06] Teoria de Algoritmos I
Primer cuatrimestre de 2022

Alumno	Padron	Email
BENITO, Agustin	108100	abenito@fi.uba.ar
BLÁZQUEZ, Sebastián	99673	sblazquez@fi.uba.ar
DEALBERA, Pablo Andres	106585	pdealbera@fi.uba.ar
DUARTE, Luciano	105604	lduarte@fi.uba.ar
PICCO, Martín	99289	mpicco@fi.uba.ar

Entrega:	Primera
Fecha:	Miercoles 18 de Mayo del 2022

Índice

1. Parte 1: El viaje a Qatar

1.1. Problema enunciado

Se nos presenta con el siguiente problema: Dado un grupo de ciudades, que incluye una ciudad de origen O y una ciudad destino D, y una serie de caminos que conectan dichas ciudades que tienen de atributo una capacidad máxima de personas que pueden circular y además un costo por persona, encontrar cual es la máxima cantidad de personas que pueden ir desde O hasta D al menor costo posible. Podemos asumir que la capacidad de personas es siempre positiva y entera, al igual que los costos por persona.

1.2. Transformacion al problema de flujo máximo con costo mínimo

El problema enunciado anteriormente se puede representar mediante una red de flujo donde la ciudad de origen es la fuente, el destino es el sumidero, y donde ademas, las ciudades serían los vértices, los caminos las aristas y las personas que van a viajar, el flujo. A diferencia de las redes de flujo con las que veníamos trabajando, las aristas en este caso van a tener otro atributo ademas de la capacidad. Tambien, por cada arista vamos a tener un costo por unidad de flujo. En el modelo presentado, la capacidad representa a la cantidad maxima de personas que pueden ir de una ciudad a otra. Luego, el costo por unidad de flujo es el costo por persona que utilice la ruta que conecte 2 ciudades. Donde el costo de un flujo cualquiera resulta la sumatoria del producto entre el flujo resultante en cada arista y el costo por unidad de de cada arista. Entonces, el problema presentado en el cual debemos encontrar las rutas para transportar la máxima cantidad de personas al costo mínimo se traduce en un problema de encontrar el flujo máximo con costo mínimo.

1.3. Resolucion del Flujo maximo con costo mínimo

Para resolver este problema vamos a utilizar un algoritmo conocido como el "Cycle cancelling algorithm" que fue originalmente planteado por Klein en 1967. [5] Lo que se propone en este algoritmo se divide en dos grandes partes, encontrar el flujo maximo y luego minimizar el costo.

1.3.1. Ford-Fulkerson

Primero hay que encontrar el flujo máximo. Esto podemos hacerlo mediante el metodo de Ford-Fulkerson. Para explicar este método debemos primero explicar los conceptos de camino de aumento y grafo residual. Dado una red de flujo $G = (V, E)$ y un flujo f, el grafo residual G_f es un grafo dirigido que indica como podemos cambiar el flujo de la red G. Para crear el grafo residual, primero obtenemos las capacidades residuales siguiendo[2]:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si}(u,v) \in E \\ f(v,u) & \text{si}(v,u) \in E \\ 0 & \text{si no} \end{cases}$$

Es decir que la capacidad residual de una arista (u, v) va a ser su capacidad menos el flujo que pasa por esa arista y la de una arista (v, u) va a ser directamente el flujo. Entonces, utilizando los mismos vértices de la red solo que por cada arista con flujo en la red, el grafo residual va a tener dos aristas.

Luego, un camino de aumento es simplemente un camino que vaya de la fuente al sumidero mediante el grafo residual.[2]

Entonces, sabiendo estas dos cosas podemos plantear el algoritmo. Primero, inicializamos el flujo total en 0, es decir ponemos en 0 el flujo por cada arista. Luego, mientras haya un camino de aumento, esto puede hallarse por DFS o BFS (luego vamos a ver que es conveniente utilizar BFS) por ejemplo, sabemos que el cuello de botella va a ser la mínima capacidad en este camino, pues no va a poder pasar más flujo que el cuello de botella. Luego, aumentamos el flujo en este camino en el grafo residual, saturando la arista del camino cuya capacidad era la mínima. Esto se

hace con que, por cada arista en el camino, si esta arista pertenece a la red de flujo entonces el nuevo flujo va a ser el actual más el cuello de botella. Si no, significa que la arista es una arista que vuelve en el grafo residual y debemos restarle el cuello de botella.[2]

1. Pseudocódigo

```
Ford_fulkerson(G,s,t)
    por cada arista (u,v) en G
        inicializar flujo de (u,v) en 0

    mientras haya un camino de aumento P
        Sea c_m la menor capacidad de P
        por cada arista (u,v) en P
            si (u,v) pertenece a G
                (u,v).flujo = (u,v).flujo + c_m
            si no
                (v,u).flujo = (v,u).flujo - c_m
```

2. Complejidad En primer lugar, se asigna un flujo 0 a cada arista. Esta operación cuesta $O(E)$. Luego, tenemos un ciclo que itera mientras haya un camino de aumento. Analizar este ciclo es un tanto más complejo.

   Primero, como todas las capacidades son enteras positivas, todas las capacidades residuales del grafo G_f van a ser enteras positivas. Luego, sea G una red de flujo con un flujo f y sea \$P % un camino de aumento. Entonces, por lo mencionado en la explicación del algoritmo, el flujo nuevo es el actual más el cuello de botella, que es positivo y entero. Esto implica que el flujo nuevo es siempre mayor al anterior por al menos una unidad. Por otro lado, para buscar una cota máxima de flujo, suponemos que todas las aristas salientes de la fuente van a estar saturadas. Con esta suposición, el flujo máximo es la sumatoria de la capacidad de este subconjunto de aristas. Llamando a este número C, sabemos que c es la una cota superior del flujo máximo[**Tardos**]. Esto implica que como mucho, el C iteraciones. En cada iteración, se visita cada arista del camino de aumento. Por lo tanto, la complejidad final de ese ciclo es $O(C.E)$

   Entonces, la complejidad final de Ford-Fulkerson resulta $O(C.E)$. Es decir, seria pseudo-polinomial

   Sin embargo, esto es porque no sabemos como se elige el camino de aumento. Si nosotros elegimos el camino de aumento como el mas corto entre la fuente y el sumidero, es decir utilizamos BFS, la complejidad resulta polinomial. Esto es porque si elegimos el camino mas corto se va a saturar como minimo una arista. Entonces, el proximo camino minimo va a ser de igual longitud o mas largo, con una longitud maxima posible de V. De esta forma, en lugar de hacer C iteraciones vamos a terminar haciendo VE interaciones.

   A este algoritmo se lo conoce como Edmonds-Kalp y tiene una complejidad de $O(VE^2)$

3. Optimalidad Como se mencionó en la explicación del método, si hay un camino de aumento, podemos aumentar el flujo donde el flujo nuevo va a ser mayor al flujo anterior. Entonces, cuando no haya más caminos de aumento, el flujo va a ser máximo. Por lo presentado en la sección de complejidad, sabemos que eventualmente no va a haber más caminos de aumento. Entonces, la solución debe ser optima.

1.3.2. Ciclos negativos

Luego, la segunda parte del algoritmo se trata de minimizar el costo del flujo máximo calculado. Para ello, es importante entender como funciona el costo y como este puede ser representado en el grafo residual. Sea f el flujo sobre una red, E el conjunto de las aristas y c(x) una función costo, tenemos que el costo del flujo es:

$$c(f) = \sum_{e \in E} f(e)\ c(e)$$

El hecho de que tengamos un costo en las aristas va a impactar en el grafo residual de la forma que el par del grafo residual de una arista en la red de flujo va a tener el mismo costo pero negativo**[Kelin]** . Es decir, notando a una arista e como los vértices u,v que conecta, en orden de la dirección, esto lo representamos como:

$$c(u, v) = -c(v, u)$$

Luego, un teorema [1] para redes de grafos indica que un flujo f es el de costo mínimo si y solo si no hay ciclos negativos en los costos del grafo residual. Esto es porque [3]:

Sea *gamma* un ciclo de costo negativo en el grafo residual G_f y llamando C_m a la capacidad residual mínima presente en *gamma*. Podemos aumentar el flujo sobre este ciclo de forma que las aristas que no pertenezcan al ciclo se vean inafectadas y las que si pertenezcan al ciclo se modifiquen así:

$$f_{nuevo}(u, v) = \begin{cases} f(u, v) + C_m & \text{cuando}(u, v) \in \gamma \\ f(v, u) - C_m & \text{cuando}(u, v) \in \gamma \end{cases}$$

Entonces, el costo del nuevo flujo se obtiene de:

$$c(f_{nuevo}) = c(f) + c_m \ c(\gamma)$$

De acá, es evidente que mientras $c(\gamma)$ sea negativo, es decir que existan ciclos de costos negativos va a existir un flujo más barato que el actual. [3]

Entonces, ahora entendemos que es necesario que el grafo residual final no tenga ciclos negativos de costo. Para encontrar los ciclos negativos podemos utilizar Bellman-Ford.

Teniendo esto en consideración, el algoritmo para calcular el flujo máximo de costo mínimo, que fue originalmente propuesto por Morton Klein en 1967, propone 5 pasos para resolver el problema [5]:

1. Obtener el flujo máximo de la red sin considerar el costo.
2. Actualizar el grafo residual G_f con el costo por unidad de flujo negativo, como se mencionó previamente.
3. Probar si hay ciclos negativos dirigidos en el grafo G_f. En caso de no haber, se terminó el problema.
4. Se redistribuye el flujo de manera que se satura una de las aristas del ciclo negativo.
5. Repetir desde el punto 2.

Entonces, la idea es primero obtener el flujo máximo da la red dada, lo hacemos por Edmonds-Karp. Luego, por lo presentado anteriormente, mientras existan ciclos de costo negativo sabemos que se puede reducir el costo saturando una de las aristas del ciclo negativo. Como sucede en Ford-Fulkerson, cada vez que aumentamos el flujo reducimos el costo total del flujo por al menos 1. Esto último se cumple porque los costos de cada arista son enteros positivos, pues el precio de los pasajes son números enteros. De esta manera, tenemos que si comenzamos con un costo inicial $C_i(f)$ vamos a ir reduciendo el costo de a por lo menos 1 hasta llegar al costo final $C_f(f)$. Entonces, el algoritmo eventualmente va a finalizar en $C_f(f) - C_i(f)$ iteraciones y obtendremos la solución óptima.

1.3.3. Pseudocodigo

```
Cycle-canceling():
    Sea f el flujo maximo
    Sea G(f) el grafo residual
    Obtener f y G(f) mediante Edmonds-Karp
    costo_min = 0
    Mientras G(f) tenga un ciclo negativo C:
        Obtener el C mediante Bellman-Ford
        Sea c_m la menor capacidad residual de C
        Por cada arista (u,v) en C
            si (u,v) pertenece a C
                (u,v).flujo = (u,v).flujo + c_m
            si no
                (v,u).flujo = (v,u).flujo - c_m

    Por cada arista (u,v) en G
        costo_min += (u,v).flujo * (u,v).costo


    Retornar f y costo_min
```

1.3.4. Análisis temporal y espacial

Primero aplicamos Edmonds-Karp para obtener el flujo máximo cuya complejidad temporal es $O(V\ E^2)$, donde V es la cantidad de vertices y E la cantidad de aristas que tiene la red. Luego, se aplica el algoritmo de Bellman-Ford, cuya complejidad es $O(V\ E)$ hasta que no haya mas ciclos negativos.

Siendo que, como mencionamos antes, cada iteración reduce como mínimo en 1 el costo, se va a llamar a Bellman-Ford un máximo de $C_i(f) - C_f(f)$ veces. Ahora, el costo inicial depende del algoritmo por lo tanto es mejor establecer una cota superior al costo inicial. Llevándolo a un extremo, sea C_{max} la capacidad más alta de una arista, a K_{max} como el costo más alto de una arista, entonces, el costo inicial tiene una cota superior tal que $C_i(f) \leq K_{max}\ C_{max}\ E$. Pensando que C_f tiene que ser positivo, podríamos maximizar a la resta como $C_i(f) - C_f(f) \leq K_{max}\ C_{max}\ E$.

Por lo tanto, el algoritmo de Bellman-Ford va a ser llamado un máximo de $K_{max}\ C_{max}\ E$ veces. Entonces, la complejidad temporal del algoritmo resulta $O(V\ E^2\ K_{max}\ C_{max})$.

1.4. Detalles de implementación

El algoritmo fue implementado en Python y probado con la versión `3.10.4`.

Para la ejecución del algoritmo normal no hay dependencia, para exportar el grafo a imagen, se necesita como dependencia `graphviz` que se puede instalar con:

```
pip install graphviz
```

1.4.1. Ejecución del programa

El programa contiene un `shebang` para ser ejecutado en una terminal de la siguiente forma:

```
./src/parte_1.py <filename>
```

El comprimido entregado incluye un carpeta en `assets/` con grafos ejemplos, por ejemplo:

```
./src/parte_1.py ./assets/grafo-qatar.csv
```

```
La cantidad maxima de personas que pueden viajar es:  6
El costo de todos los viajes es:  14
```

1. Exportador de Grafo a Imagen

   Aparte de esto, esta incluido un exportador que genera un imagen en formato *SVG* de los grafos y se puede generar con el siguiente comando:

```
./src/export.py ./assets/grafo-qatar.csv
```

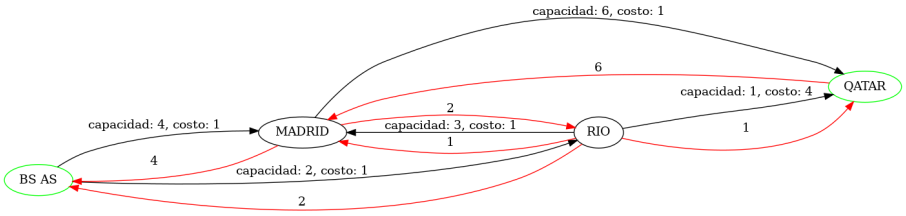

Figura 1: Hospital con un entrenador cargado.

2. Parte 2: Un reality único

2.1. Definición de $EXACT - COVER$

Dado un conjunto de elementos $U = u_1, u_2, \ldots, u_n$, y un conjunto $S = S_1, S_2, \ldots, S_m, S_j \subseteq U$; hallar un conjunto $T \subseteq S$ tal que $\bigcap_{i=1}^{|T|} T_i = U$, y que $T_i \cap T_j = \varnothing, \forall T_i, T_j \in T$ (los elementos son disjuntos entre si). Decimos entonces, que T es una cobertura exacta, o partición, de U. [4]

2.2. Demostración $CASTING \in NP - C$

Buscamos demostrar que $CASTING$ pertenece a la clase NP-C. Para ello, requerimos demostrar que:

1. $CASTING \in NP$
2. $CASTING \in NP - H$

2.2.1. $CASTING \in NP$

Para demostrar (i), requerimos hallar un algoritmo certificador que verifique una solución del problema en tiempo polinomial. Sugerimos el siguiente:

```
# P: conjunto de participantes propuestos, con las caract. que cumplen (pi.caract)
# K: conjunto de características deseadas
CERT-CASTING(P, K)
    # (1) máximo un participante por característica, pero un participante puede tener varias
    Verificar |P| ≤ |K|

    # (2) no hay características repetidas entre participantes
    Verificar que p_i.caract ∩ p_j.caract = ∅, ∀p_i, p_j ∈ P, i ≠ j

    # (3) se está cumpliendo con todas las características deseadas
    Siendo C = ⋂_{i=1}^{|P|} p_i.caract el conjunto de todas las
    características presentes en el conjunto P, verificar que:
        # (a)
        c_i ∈ K, ∀c_i ∈ C
        # (b)
        |C| = |K|
```

Análisis de complejidad:

1. Se realiza en tiempo constante ($O(1)$)
2. Involucra una comparación de todos contra todos ($O(|P|^2)$); la comparación involucra la intersección de dos listas, que en el peor de los casos pueden contener todas las características, pero si se hace uso de un hashmap puede realizarse en tiempo lineal, al ser la iteración de dos listas ($O(|K|)$). En consecuencia, resulta $O(|P|^2|K|)$.

3. Para construir el conjunto de características cubiertas se iteran todas las personas seleccionadas y se unen las listas de características ($O(|P||K|)$). A continuación se verifica que el conjunto de características cubiertas tenga el mismo tamaño que el de características deseadas ($O(1)$). Por ende, resulta $O(|P||K|)$.

Por el análisis realizado, el algoritmo certificador tiene complejidad $O(1 + |P|^2|K| + |P||K|) = O(|P|^2|K|)$, que resulta ser polinomial, por lo que $CASTING \in NP$.

2.2.2. $CASTING \in NP - H$

Para demostrarlo, con reducir un problema $X \in NP - C$ a $CASTING$, podemos demostrar que pertenece a $NP-H$, dado que al ser $X \in NP-C$, vale que $X \in NP-H$, por lo que podemos reducir cualquier problema $Y \in NP$ a X y, si X se puede reducir a $CASTING$, por transitividad Y también es reducible al mismo.

Asumiendo que $EXACT - COVER \in NP - C$, si se logra encontrar una reducción tal que $EXACT - COVER \leq_P CASTING$, podemos afirmar que $CASTING \in NP - C$, y que por lo tanto $CASTING \in NP - H$.

Sugerimos el siguiente algoritmo:

```
EXACT-COVER-TO-CASTING
    # (1) las características se corresponden con los elementos del conjunto a cubrir
    # (no hay dos personas que vayan a cumplir una característica)
    K = U

    # (2) las personas se corresponden con cada conjunto a elegir
    P = {}
    para cada S_j ∈ S:
        P = P ∪ {id : j, caract : S_j}
```

Análisis de complejidad:

1. Tiempo constante, dado que es el mismo conjunto ($O(1)$)

2. Para generar el conjunto de "candidatos", iteramos los conjuntos a elegir para la cobertura, y representamos a cada uno como una "persona" que participa del casting, y que cumple determinadas características, que son finalmente los elementos del conjunto para el que se busca hallar la partición. Esto se realiza en tiempo lineal. ($O(|S|)$)

Resulta entonces que el algoritmo de reducción es polinomial, por lo que hallamos una reducción polinomial tal que $EXACT - COVER \leq_P CASTING$, por lo que, asumiendo que $EXACT - COVER \in NP - C$, podemos asegurar que $CASTING \in NP - H$.

Quedan demostradas las dos condiciones para que $CASTING \in NP - C$.

2.3. Demostración $EXACT - COVER \in NP - C$

Las condiciones a demostrar son las mismas que en la sección anterior.

2.3.1. $EXACT - COVER \in NP$

Se propone el algoritmo:

```
# X: conjunto del que se busca verificar la partición
# S: colección de conjuntos que potencialmente forman una partición de X

CERT-EXACT-COVER(X, S)
    # (1) máxima cantidad de conjuntos
    Verificar |S| ≤ |X|
    # (2) todo conjunto en S está incluido en X
    Verificar que Si ⊆ X, ∀S_i ∈ S
    # (3) los conjuntos en S son disjuntos
    Verificar que S_i ∩ S_j = ∅, ∀S_i, S_j ∈ S, coni ≠ j
    # (4) la unión de los conjuntos en S forman a X
    Verificar que ⋃_{i=1}^{|S|} S_i = X
```

Análisis de complejidad:

1. Se puede realizar en tiempo constante ($O(1)$)
2. Se puede verificar que un conjunto $S_i \subseteq X$ tomando cada elemento del conjunto S_i y verificando que este se encuentra en X. Si nuestro algoritmo utiliza un set o hashmap para guardar al conjunto X entonces esta operación se puede realizar en $O(|S_i|)$. Luego dicha operación será necesaria realizarla para todos los conjuntos S_i, consecuentemente este paso tomará $O(|S||max\ S_i|)$.
3. Se puede verificar realizando la misma validación que en el paso (2) que toma $O(|S_i|)$ pero entre cada uno de los conjuntos que son parte de S entre sí. Es decir, que es necesario realizar dicha operación $O(|S|^2)$ veces, lo cual termina teniendo una complejidad $O(|max\ S_i||S|^2)$.
4. Se puede verificar tomando cada elemento de X y validando que este pertenezca a alguno de los conjuntos S_i. Nuevamente, si nuestro algoritmo utiliza un set o hashmap para guardar a los conjuntos S_i entonces esta operación se puede realizar en $O(|X||S|)$.

Finalmente nuestro algoritmo verificador toma $O(1 + |S||max\ S_i| + |max\ S_i||S|^2 + |X||S|) = O(|max\ S_i||S|^2 + |X||S|)$, es decir, un tiempo polinomial en función de los parámetros de entrada.

2.3.2. $EXACT - COVER \in NP - H$

Para demostrar que $EXACT - COVER \in NP - H$ realizaremos una reducción polinomial del problema $3SAT$ a este. Para ello partiremos del problema $3SAT$ en el cual tenemos varias cláusulas con máximo 3 literales cada uno, por ejemplo $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$. A partir de esta expresión booleana construiremos el conjunto

$$X = \{x_1, x_2, \ldots, x_{|X|}, C_1, C_2, \ldots, C_{|C|}, n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{23}, ..., n_{c1}, n_{c2}, n_{c3}\}.$$

Los valores x_i representan a las variables x_i en el problema $3SAT$, C_j representa a las cláusula j de dicho problema y n_{km} representa un "nexo" entre la variable x_i que se encuentra en el número m de la claúsula k con dicha cláusula.

Luego podemos construir el conjunto S que contiene los posibles subconjuntos de X con los que se busca armar la partición de X. Para ello incluiremos en S a los nexos creados previamente $(\{n_{11}\}, \{n_{12}\}, \{n_{13}\}, \{n_{21}\}, \{n_{22}\}, \{n_{23}\}, ..., \{n_{c1}\}, \{n_{c2}\}, \{n_{c3}\})$ y a estos nexos con sus cláusulas $(\{C_1, n_{11}\}, \{C_1, n_{12}\}, \{C_1, n_{13}\}, \{C_2, n_{21}\}, \{C_2, n_{22}\}, \{C_2, n_{23}\}, \ldots, \{C_{|C|}, n_{|C|1}\}, \{C_{|C|}, n_{|C|2}\}, \{C_{|C|}, n_{|C|3}\})$. Finalmente también deberemos agregar a S dos conjuntos por cada variable x_i del problema $3SAT$ que representarán la posibilidad de que x_i tome un valor verdadero o falso. En el caso del valor verdadero de x_i el conjunto que llamaremos V_i incluirá a x_i y a los n_{km} en los que x_i tiene un literal negado. En el caso del valor falso de x_i el conjunto que llamaremos F_i incluirá a x_i y a los n_{km} en los que x_i tiene un literal no negado.

Finalmente se resuelve el problema $EXACT-COVER$ con los X y S previamente construidos y se transforma el resultado para obtener el de $3SAT$. Para ello se buscan cuáles conjuntos V_i o F_i se utilizaron para realizar la partición. Si se utilizó el conjunto V_i quiere decir que la variable x_i toma un valor verdadero y si se utilizó el conjunto F_i quiere decir que la variable x_i toma un valor falso.

Para ejemplificar este procedimiento veamos un ejemplo. Si el problema $3SAT$ a resolver es $(x_1 \vee x_2) \wedge (\ x_1 \vee x_4 \vee\ x_3)$ entonces en la primera transformación construimos los conjuntos:

- $X = \{x_1, x_2, x_3, x_4, C_1, C_2, n_{11}, n_{12}, n_{21}, n_{22}, n_{23}\}$
- $V_1 = \{x_1, n_{21}\}, V_2 = \{x_2\}, V_3 = \{x_3, n_{23}\}, V_4 = \{x_4\}$
- $F_1 = \{x_1, n_{11}\}, F_2 = \{x_2, n_{12}\}, F_3 = \{x_3\}, F_4 = \{x_4, n_{22}\}$

- $S = \{\{n_{11}\}, \{n_{12}\}, \{n_{21}\}, \{n_{22}\}, \{n_{23}\}, V_1, V_2, V_3, V_4, F_1, F_2, F_3, F_4,$
  $\{C1, n_{11}\}, \{C1, n_{12}\}, \{C2, n_{21}\}, \{C2, n_{22}\}, \{C2, n_{23}\}\}$

Luego de esto una posible solución al problema $EXACT - COVER$ puede ser la partición $P = \{\{n_{12}\}, \{n_{21}\}, V_1 = \{x_1, n_{21}\}, V_2 = \{x_2\}, V_3 = \{x_3, n_{23}\}, V_4 = \{x_4\}, \{C1, n_{11}\}, \{C2, n_{22}\}\}$ y la transformación para la solución es:

- $V_1 \rightarrow x_1$ verdadero
- $V_2 \rightarrow x_2$ verdadero
- $V_3 \rightarrow x_3$ verdadero
- $V_4 \rightarrow x_4$ verdadero

Análisis de complejidad:

1. La construcción del conjunto X toma $O(|X|)$ para agregar a los elementos x_i, $O(|C|)$ para agregar a los elementos C_j que representan a las claúsulas y $O(|X||C|)$ para agregar a los elementos que representan sus nexos n_{km}, siendo $|X|$ la cantidad de variables y $|C|$ la cantidad de cláusulas en el problema $3SAT$.

2. Por otro lado la construcción del conjunto S toma $O(|X||C|)$ para agregar a los elementos que representan los nexos n_{km}, $O(|X||C|)$ para agregar a los elementos V_i y F_i y $O(|C|)$ para agregar a los elementos C_j.

3. Finalmente la transformación del resultado de $EXACT - COVER$ en el resultado de $3SAT$ se puede realizar en un tiempo lineal recorriendo la solución de $EXACT - COVER$. Consecuentemente podemos justificar que las transformaciones son polinomiales y que por lo tanto la reducción es polinomial demostrando que $EXACT - COVER \in NP - H$ y por ello $EXACT - COVER \in NP - C$.

2.4. Utilizando el concepto de transitividad y la definición de NP-C explique qué ocurriría si se demuestra que el problema EXACT-COVER pertenece a la clase P

Si $EXACT - COVER \in P$, quiere decir que existe un algoritmo que resuelve el problema en tiempo polinomial. Como se demostró, $EXACT - COVER \in NP - C$, que equivale a decir que $EXACT - COVER \in NP, NP - H$, por lo que, para cualquier problema $X \in NP$, podemos hallar una reducción polinomial para llevarlo a $EXACT - COVER$, de manera que $X \leq_P EXACT - COVER$. Dado que, bajo la hipótesis dada, $EXACT - COVER$ se puede resolver en tiempo polinomial, y X puede reducirse a $EXACT - COVER$ en tiempo también polinomial; resolver X también se vuelve polinomial, utilizando primero la reducción a $EXACT - COVER$, obtener su resolución, y luego transformar a la solución de X, todas operaciones polinomiales. Como se encontró un algoritmo que resuelve X en tiempo polinomial, se puede afirmar que $X \in P, \forall X \in NP$. Luego, $P = NP$.

2.5. Un tercer problema al que llamaremos X se puede reducir polinomialmente a EXACT-COVER, qué podemos decir acerca de su complejidad?

Si $X \leq_P EXACT - COVER$, se puede afirmar que la complejidad de $EXACT - COVER$ es igual o mayor a la de X, y que si podemos resolver cualquier instancia de $EXACT - COVER$, también podemos resolver cualquiera de X.

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

Los problemas de tipo $NP-C$ se encuentran incluídos en las clases NP y $NP-H$, lo cual quiere decir que se puede construir un verificador polinomial de posibles soluciones a dichos problemas (son NP) y que son al menos tan difíciles de resolver como cualquier otro problema en NP (son $NP - H$).

Una característica interesante de los problemas $NP - C$ es que se pueden reducir polinomialmente entre sí, con lo cual si se resuelve uno de ellos en tiempo polinomial entonces también resulta posible hacerlo para los demás.

A su vez, una de las preguntas más importantes a resolver es si $NP = P$. Si se demuestra que un problema en $NP - C$ se puede resolver en tiempo polinomial entonces se demuestra que cualquier problema en $NP - C$ se puede resolver con la misma complejidad y por lo tanto que todos los problemas en NP se pueden resolver polinomialmente y que $NP = P$. Por otro lado si se demuestra que un problema en NP no se puede resolver en tiempo polinomial entonces se demuestra que $NP! = P$, pero lo complicado de esto último es que es difícil demostrar que no existe un algoritmo que resuelva un problema en tiempo polinomial, sino que tal vez no lo hemos encontrado hasta el momento.

3. Fuentes

[1] R. Busacker y T. L. Saaty. *Finite Graph Networks*. 1.ª ed. McGraw-Hill, 1965.

[2] Thomas Cormen y col. *Introduction to Algorithms*. 3.ª ed. The MIT Press, 2009.

[3] Jeff Erickson. *Algorithms*. 1.ª ed. University of Illinois, 2017.

[4] Richard M. Karp. *Complexity of Computer Computations*. 1972. Cap. Reducibility Among Combinatorial Problems, págs. 85-103.

[5] Morton Klein. "A PRIMAL METHOD FOR MINIMAL COST FLOWS WITH APPLICATIONS TO THE ASSIGNMENT AND TRANSPORTATION PROBLEMS". En: *Management Science* 14.3 (1967), págs. 205-266.