FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico Nº3

Un proyecto distribuido en dos y *Más allá de Bellman-Ford*

Children of Dijkstra

Fecha entrega: 5/12/2022 (Tercera entrega)

[75.29] Teoria de Algoritmos I

Segundo cuatrimestre 2022

Alumno

Bauni	Chiara	102981
Leloutre	Daniela	96783
Guglielmone	Lionel	96963
Leon	Agustina	102208
Buceta	Belen	102121

Índice

1. Un proyecto distribuido en dos

Contamos con un proyecto que tiene un conjunto de tareas a realizar. Entre algunas tareas existe cierta interdependencia, es decir que para realizar una se requiere los resultados que otorgan una o más tareas previas. Se cuenta con dos equipos de trabajo con sus propios recursos. Cualquier tarea la puede realizar cualquiera de los equipos con un costo asociado. Llamaremos $c_{i,e}$ al costo que el equipo "e" realice la tarea i. Existe un costo de transferencia de resultados que se aplica si el resultado de una tarea elaborada por un equipo se la debe trasladar al otro para resolver otra tarea. Llamaremos $t_{i,j}$ al costo de la transferencia del resultado de la tarea i a la tarea j. Si dos tareas son realizadas por el mismo equipo no hay costo asociado al intercambio de resultados. Consideraremos que todos los costos son enteros positivos. Se desea minimizar el costo total del desarrollo del problema.

Se pide:

1. Construya una estrategia utilizando redes de flujo que solucione el problema. Utilice gráficos para representar la red de flujo diseñada. Explíquela con sus palabras.
2. Brinde pseudocódigo que explique cada uno de los pasos de su solución.
3. Analice la complejidad temporal y espacial de su propuesta.
4. Brinde al menos un ejemplo paso a paso donde se aprecie el funcionamiento del mismo.
5. Programe el algoritmo.
6. Analice: su programa mantiene la complejidad temporal y espacial teórica?

Formato de los archivos:

El programa debe recibir por parámetro el path del archivo con los procesos a realizar. El archivo de los procesos es un archivo de texto donde cada línea corresponde a un proceso, sus costos y dependencias.

El formato de la línea es:

```
CÓDIGO_PROCESO,COSTO_EQUIPO1,COSTO_EQUIPO2,[LISTA DE DEPENDENCIAS CON SUS COSTOS]
```

La lista de dependencias corresponde a los procesos que siguen al mismo y el costo asociado de que esta la realice el otro equipo.

Ejemplo: "procesos.txt"

```
A,1,2,B,2
B,5,10,C,3,D,1
C,4,4,D,2
D,1,3
...
```

En el ejemplo la tarea "A" cuesta 1 si la realiza el equipo 1 y 2 si la realiza el equipo 2. El resultado de la tarea "A" es necesario para realizar la tarea "B". Si la tarea "A" y "B" son realizadas por diferentes equipos entonces la transferencia de resultado cuesta 2

Debe resolver el problema y retornar por pantalla la solución. Debe mostrar que tareas debe resolver cada equipo y el costo total que se incurrirá dada esta división.

1.1. Estrategia de redes de flujo

La solución al problema de la selección de tareas con dos equipos considera las siguientes hipótesis:

- Ambos equipos son igualmente eficientes tanto en tiempo como en desempeño de la tarea que se le asigna
- Las tareas son completadas por el equipo asignado hasta el final
- Los costos de tiempo del traspaso de las actividades están contemplados en el costo del traslado
- Sea P el conjunto de tareas necesarias para completar el proyecto P, las mismas se realizan todas, con independencia del equipo que las realiza
- Cada proyecto aporta una ganancia G y concretar dicho proyecto involucra una serie de tareas, cada una con un costo asociado $c_{i,e}$. Independientemente de que equipos realicen qué tareas, el valor de G es constante. Por lo tanto, la asignación de tareas busca minimizar el costo de toda la operación o, lo que es lo mismo, busca maximizar la diferencia entre la ganancia G y el costo C que se incurre al realizar las tareas.
- La constante G es un valor positivo y su valor absoluto es mayor al valor absoluto del costo mas alto.
- Los costos asociados a cada tarea $c_{i,e}$ son negativos.
- Dada una serie de tareas $t_1, t_2, .., t_k$ con una secuencia definida, la ultima tarea t_k se vincula con el nodo G que representa la ganancia del proyecto.
- C es la sumatoria de todos los costos de todas las tareas realizadas por los equipos mas las transferencias, esto es:
$$C = \sum c_{i,e} + t_{i,j} \tag{1}$$
- Sean t_1, t_2 y t_3 tres tareas tales que t_3 depende de t_1 y t_2 para realizarse. Entonces, no hay transitividad entre las tareas, es decir, es suficiente que t_2 se realice antes que t_3 y t_1 antes de t_2 (no hay vinculo directo entre t_3 y t_1).

Estrategia: Dado un conjunto de tareas $\{t_1, t_2, .., t_n\}$ y costos asociados a cada tarea según el equipo que la realice y un costo de transferencia por traspaso de tareas, se procede a la construcción de la red de flujos:

1. Agregar un nodo s (fuente) y t (sumidero) y un nodo por cada tarea condicionado por el equipo que la realiza.
2. Adicionalmente, agregar un nodo auxiliar G que representa la ganancia del proyecto
3. Para cada nodo i con. $p_i > 0$, agregar un eje $s - i$ con capacidad p_i. En este caso, el único nodo con valor positivo sera el que representa la ganancia del proyecto, es decir G.

4. Para cada nodo i con $p_i < 0$, agregar un eje $i - t$ con capacidad $-p_i$. En este caso, los nodos de valor negativo son las tareas que conlleva realizar el proyecto.

5. Por cada relación de precedencia (j precede a i) $\rightarrow$ agregar un eje $i - j$ con capacidad R (i.e. la capacidad del eje) sin transitividad.

Finalizada la construcción de la red, se aplica el algoritmo de Ford-Fulkerson para encontrar la ganancia máxima. Además, se obtienen los subconjuntos A' y B' y el corte mínimo que maximiza esta ganancia, es decir:

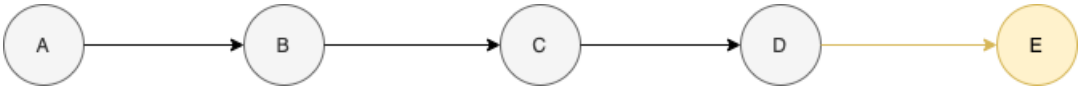
$$Ganancia = G - C \tag{2}$$

Con el resultado que arroja este algoritmo, se obtiene el flujo máximo que para este problema particular representa la $Ganancia$ máxima. Además, dado que se conoce el valor de G, con un simple despeje se puede obtener el costo C mínimo:

$$C = G - Ganancia \tag{3}$$

Ejemplo de funcionamiento: A continuación se muestra con un ejemplo simple el funcionamiento del algoritmo planteado.

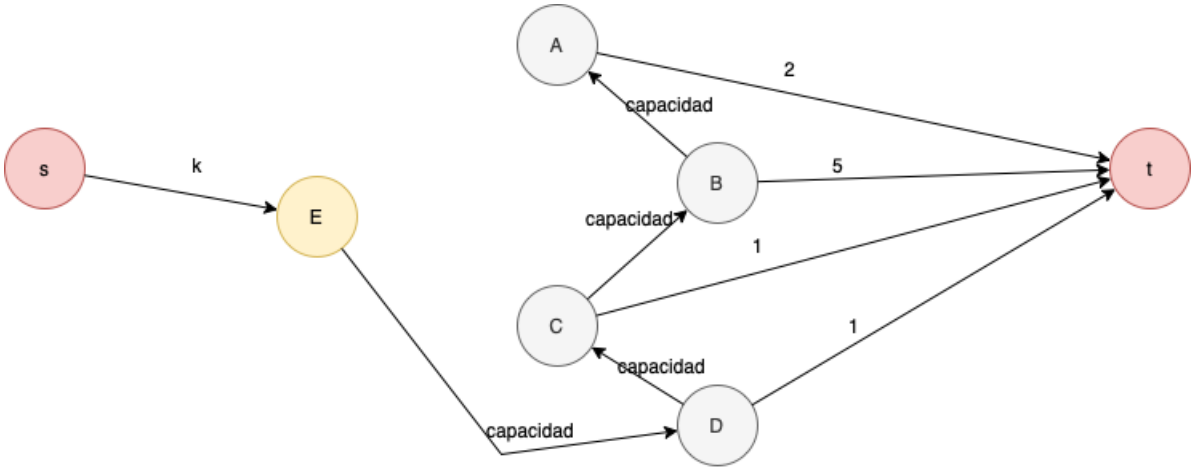
Sean A, B, CyD un conjunto de tareas a realizarse de forma secuencial tal como indica el diagrama debajo:



Además, sea E el nodo que representa el beneficio de entregar el proyecto terminado ($E > Costo\ maximo$). Los costos de cada tarea quedan representados en la siguiente tabla:

i	A	B	C	D	E
p	-2	-5	-1	-1	k>MaxCost

Entonces, la construcción de la red de flujos resulta en la siguiente estructura:

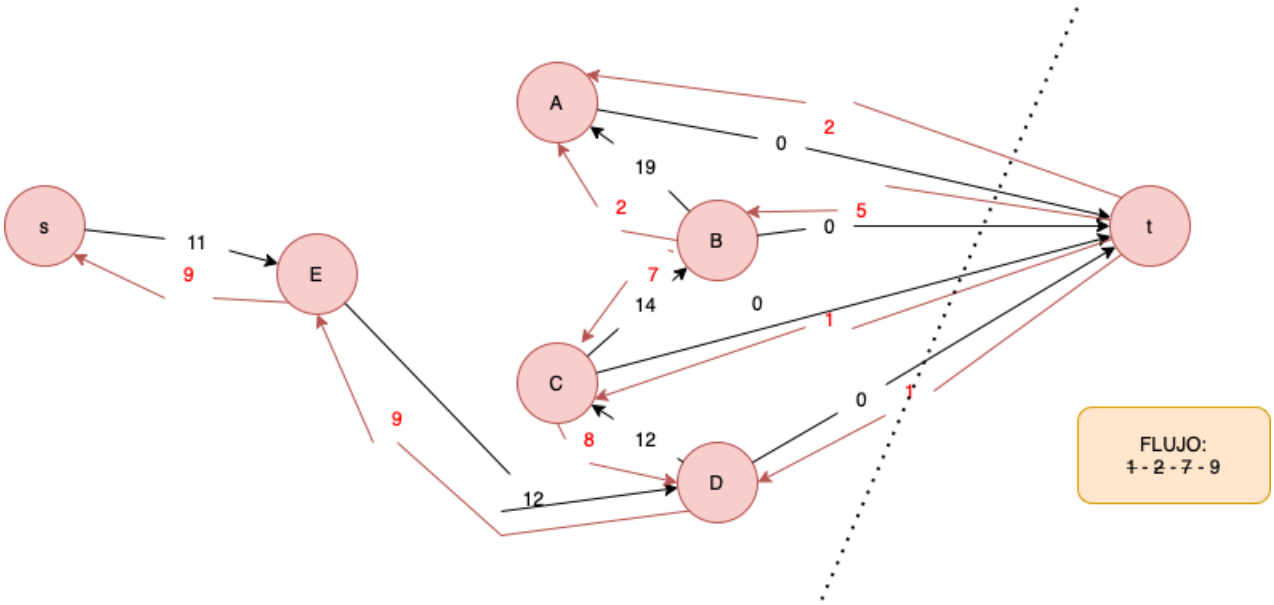


Donde

- k es un valor positivo que representa el valor bruto del proyecto (lo que paga el cliente que solicita el proyecto)
- *capacidad* es un valor a definir asociado al costo máximo en que se incurre si se elijen los procesos mas caros
- cada eje que une a A, B, C y D con t es el costo (expresado en su valor absoluto) de cada tarea en el desarrollo del proyecto

Una vez construido esta red de flujos, se procede a aplicar el algoritmo de Ford-Fulkerson, lo que arroja el flujo máximo. Los subconjuntos que se obtienen son $M = \{s, E, D, C, B, A\}$ y $N = \{t\}$.

Si suponemos que $k = 20$ (mayor que el máximo costo posible) y que $capacidad = 21$, entonces la $Ganancia = 9$

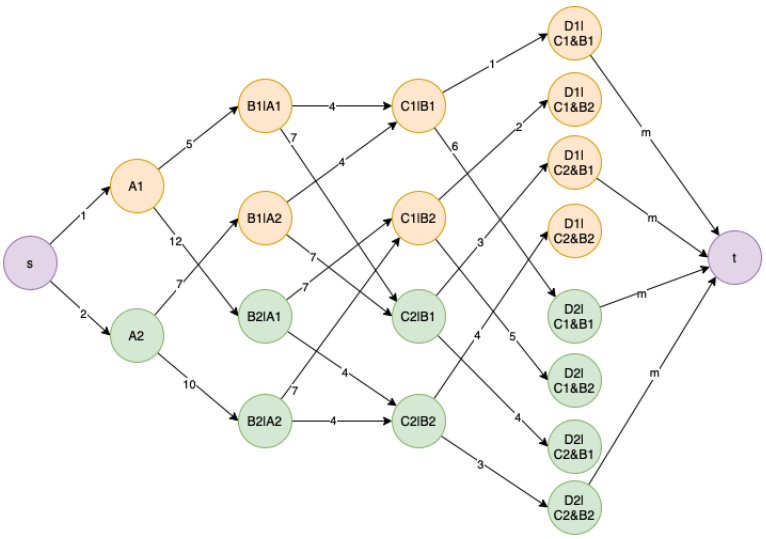


Sabiendo que $G = k = 20$ podemos aplicarlo a la siguiente diferencia:

$$Ganancia = G - C \rightarrow 9 = 20 - C \rightarrow C = 20 - 9 = 11 \tag{4}$$

Por lo que el costo mínimo es 11.

Aclaración particular del caso en el que hay interdependencia de tareas y no transitividad: En el ejemplo que se da en el enunciado, los costos de transferencia entre tareas depende de donde provengan las mismas. Por tal motivo, para una tarea t_j que depende de otras tareas $t_i, .., t_h$ se plantea una relación de condicionalidad, representada por un nodo. Esto quiere decir que cada tarea realizada por el equipo 1 y que proviene del equipo 2, es un nodo distinto a la misma tareas realizada por el equipo 1 que proviene del mismo equipo 1 (sin transferencia). El diagrama debajo ejemplifica este caso:



i	A1	A2	B1|A1	B1|A2	B2|A1	B2|A2	C1|B1	C1|B2	C2|B1	C2|B2	D1|C1&B1	D1|C1&B2	D1|C2&B1	D1|C2&B2	D2|C1&B1	D2|C1&B2	D2|C2&B1	D2|C2&B2
p	1	2	5	5 + 2	10 + 2	10	4	4 + 3	4 + 3	4	1	1 + 1	1 + 2	1 + 2 + 1	3 + 2 + 1	3 + 2	3 + 1	3

NOTA: No encontramos una forma correcta de definir los valores que damos a la ganancia G (es la constante k) y la capacidad de las aristas que van entre los nodos mas internos (el valor que llamamos *capacidad*). Esto hace imposible obtener la solución óptima al problema presentado.

Nos gustaría recibir un *feedback* especifico para determinar si el algoritmo planteado conduce a una solución óptima y así poder calcular las complejidades y poder programarlo.

1.2. Pseudo código de la estrategia

1.3. Análisis de complejidad

1.4. Ejecución del algoritmo

1.5. Complejidad del código del algoritmo

2. Más allá de Bellman-Ford

Para el cálculo del costo mínimo con costos negativos se utiliza el algoritmo Bellman-Ford. El mismo tiene como precondición para llegar a la solución óptima la ausencia de ciclos negativos en el grafo. Ante su presencia, en la búsqueda del camino, podemos quedar atrapados dentro de estos. En ese caso la longitud del camino será infinita y el costo termina infinito negativo.

Deseamos un algoritmo que ante esta misma situación nos brinde un camino que NO ingrese en los ciclos. Llamaremos a este problema como "camino simple mínimo en un grafo con ciclos negativos" (shortest simple-path in a graph with negative-cycles: "SSPGNC"). Podemos expresarlo como problema de decisión de la siguiente manera: "Dado un grafo con peso en sus ejes y un par de nodos al que llamaremos origen y destino. Queremos saber si existe un camino simple de costo menor a C que los una".

No se conoce un algoritmo eficiente que resuelva este problema.

Se pide:

1. Reducir polinomialmente el problema "Problema de camino más largo" (Longest Path problem) a "SSPGNC".

2. Demuestre que posteriormente que "Problema de camino más largo" es NP-C (puede ayudarse con diferentes problemas, recomendamos "el problema del camino hamiltoneano")

3. En base a los puntos 1 y 2, puede afirmar que el problema "SSPGNC" es NP-Completo? Si la respuesta es "no" resuelva qué faltaría para que lo sea. Si la respuesta es "si" justifíquelo utilizando los conceptos de clases de complejidad.

4. Utilizando el concepto de transitividad y la definición de NP-C explique qué ocurriría si se demuestra que existe un algoritmo eficiente que resuelve cualquier instancia de "SSPGNC".

5. Un tercer problema al que llamaremos X se puede reducir polinomialmente a "SSPGNC", qué podemos decir acerca de su complejidad?

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

2.1. Respuesta 2.1

Lo que buscamos es reducir polinomialmente el problema de camino más largo (Longest Path problem) a SSPGNC.

Para realizar el análisis: "Dado un grafo con peso en sus ejes y un par de nodos al que llamaremos origen y destino. Queremos saber si existe un camino simple de costo menor a C que los una".

Haremos antes de comenzar una observación importante en cuanto a si el grafo del problema planteado (el problema de camino más largo) es con pesos o no. En primer lugar, explicaremos la reducción considerando que es un grafo con pesos y luego observaremos qué cambiaríamos en caso de que no lo sea.

Si consideramos que el camino más largo entre dos vértices dados s y t de un grafo ponderado G = (V,E) donde V son los vértices y E las aristas. Es lo mismo que un camino más corto en un grafo ¬ G derivado de G cambiando cada peso a su negación. Por lo tanto, si los caminos más cortos se pueden encontrar en ¬ G , entonces los caminos más largos también se pueden encontrar en G .
Luego, podemos ver que para llevar a cabo la reducción necesitamos poder transformar cualquier instancia del grafo G en una instancia del grafo ¬G en tiempo O(E) . A continuación un gráfico de lo que explicamos.

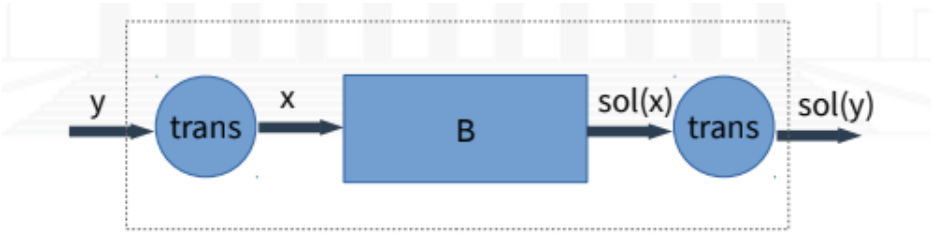


Figura 1: Reducciones
]Diapositivas de las clases de TDA 1

Lo que estamos haciendo es aplicar para cada pi peso del grafo original G, transformarlo a ¬pi. De esta manera, conformamos el grafo ¬G. Luego, resolvemos para el problema del grafo -G y obtenemos de esa forma la sol(-G). Esa solución sol(-G) la transformamos en sol(G) del problema del grafo G.
En el caso de que el grafo del problema del camino más grande no sea ponderado entonces lo que hacemos es agregarle el peso de 1 a cada arista y aplicar la misma transformación anteriormente detallada.

2.2. Respuesta 2.2

Para comenzar tendremos en cuenta que el problema del camino más largo es un problema de decisión que puede ser formulado de la siguiente manera: "Dado un grafo G(V,E) donde V son los vértices V = v1..vn y E las aristas E = e1..en, y un K entero positivo, queremos saber si existe un camino simple que contenga al menos K aristas en G". De esta manera se nos retornará por un sí o por un no como respuesta al problema.

Para demostrar que es NP-C Sea C un problema de decisión Un problema C es NP-C cuando:

- Existe un algoritmo de certificación de C puede verificar una posible solución de cualquier instancia del problema en tiempo eficiente (polinomial). Esto quiere decir que C está incluido en NP.
- Todo problema de perteneciente a NP puede ser reducido polinomialmente a C. Y por lo tanto es NP-Hard.

Comencemos por probar que el Problema de camino más largo es NP. Llamaremos PCL al Problema de camino más largo.
Pensemos en una posible solución al problema, dada una instancia de un grafo G, supongamos una lista

de aristas E de tamaño de al menos K y máximo |G| y examinamos si ésta consiste en un camino simple en el grafo G. Consideremos que se quiere encontrar la solución al grafo G(V1,Vn). Esto puede ser realizado en tiempo polinomial.
Siguiendo con la demostración ahora buscaremos probar que PCL es NP-Hard. En este caso utilizaremos el sugerido en el enunciado: “el problema del camino hamiltoneano” al que llamaremos HAM.
Procedemos a mostrar que PCL es NP-Hard reduciendo el problema del Camino Hamiltoneano a el camino más largo: HAM $< p$ PCL.

Sea una instancia del grafo G de un camino hamiltoneano, vamos a crear una instancia (G’,K) del problema del camino más largo con las siguientes características.

- G = G’: Vamos a suponer que el grado de HAM es el mismo que el de PCL.
- K = |V| - 1.

De manera tal que existirá un camino simple de longitud K en G’ si y solo si G contiene un camino Hamiltoniano.

2.3. Respuesta 2.3

No se puede afirmar que el problema “SSPGNC” es NP-Completo. Se puede decir que es NP-Hard pero falta probar que es NP, es decir, que existe un algoritmo que certifique (o verifique) a “SSPGNC” de forma eficiente.

2.4. Respuesta 2.4

Para dar la explicación utilizaremos el siguiente gráfico para ser más claros en el desarrollo de la respuesta. Además usaremos la notación PCL = .El problema del camino más largo".
Llamamos Z al conjunto de todos los problemas de tipo NP.

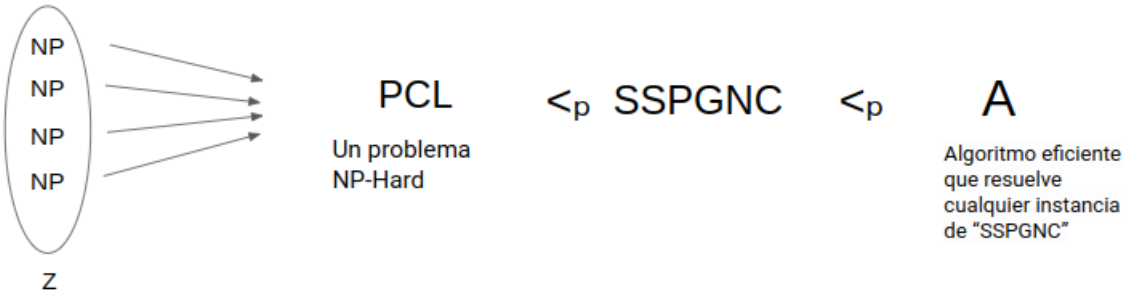


Figura 2: Diagrama

Como vemos en la imagen, tenemos muchos problemas del tipo NP que pueden reducirse a PCL $\in$ NP-Hard. De los puntos anteriores resueltos del enunciado, sabemos que PCL $\in$ NP-C, por lo tanto, PCL

$\in$ NP-Hard. Más aún esto implica que todo problema NP puede ser reducido a PCL en tiempo polinomial y de allí esa parte del gráfico.
Ahora bien, en el **inciso 1.** reducimos PCL $<_p$ SSPGNC, por lo que por transitividad,

- Z $<_p$ PCL $<_p$ SSPGNC

Por lo tanto,

- Z $<_p$ SSPGNC

De manera tal que SSPGNC resulta NP-Hard.
Con esto en mente, avanzamos y vemos que el enunciado nos propone que existe un algoritmo eficiente que resuelve cualquier instancia de "SSPGNC". Llamamos A a ese algoritmo. Esto significa que, nuevamente, por transitividad,

- Z $<_p$ PCL $<_p$ SSPGNC $<_p$ A

Por lo tanto,

- Z $<_p$ A

Observemos que esta relación indica que un problema NP-Hard (SSPGNC)puede ser reducido a un problema A que se resuelve en tiempo polinomial. Más aun, todo problema NP puede ser resuelto por un algoritmo de orden polinomial. Es decir, se están reduciendo en tiempo polinomial a todos los problemas NP. Se sigue entonces que todo problema de tipo NP sería un problema de tipo P. Finalmente, bajo esta existencia de ün algoritmo eficiente que resuelve cualquier instancia de SSPGNC.[es]taríamos probando que P = NP. Es decir, se demostraría que todo problema NP puede ser resuelto en tiempo polinomial.

2.5. Respuesta 2.5

- Si X $<_p$ SSPGNC

Entonces podemos decir que el problema SSPGNC es al menos tan dificil que el problema X.

2.6. Respuesta 2.6

Análisis entre las clases de complejidad.
Comenzamos por la clase P. Cuando hablamos de problemas de tipo P estamos haciendo referencia a aquellos problemas cuya solución cuenta con algún algoritmo de orden polinomial.
Ahora bien, en el caso de los problemas de tipo NP, estamos hablando de problemas que podemos verificar (o certificar) en tiempos de orden polinomial pero que para su solución podríamos requerir un algoritmo de orden superior.
A su vez, todo problema de tipo P es un problema de tipo NP. Es decir, P está incluido en NP: P C NP.

A continuación detallamos una explicación sobre esta afirmación.

P tiene un algoritmo de orden polinómico que lo resuelve. Mientras que un problema de tipo NP tiene un algoritmo de orden polinómico que lo verifica. De manera que dado si corremos (o procesamos) el algoritmo que soluciona a P y comprobando la solución.

Ahora bien, la relación NP C P, es una cuestión no resuelta hasta la fecha. No existe una demostración por cierta o falsa. Lo que sí podemos distinguir es que en caso de que esto sea cierto, implicaría que cualquier problema que pueda ser verificado (o certificado) en tiempo polinomial puede ser resuelto con un algoritmo de orden polinomial (se observa que esto no implica que sepamos específicamente cuál sea ese algoritmo, pero sabremos que existe).

Si NP C P no es cierto, entonces existirán problemas de los cuales no existirán algoritmos de orden polinomial que los solucionen.

Seguimos ahora con el análisis de los problemas de tipo NP-C.

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

- C es NP
- Todo problema NP se puede reducir a C en tiempo polinomial.Y por lo tanto es NP-Hard.

Para analizar ahora la relación con los otros tipos de complejidad nos valdremos del siguiente gráfico.

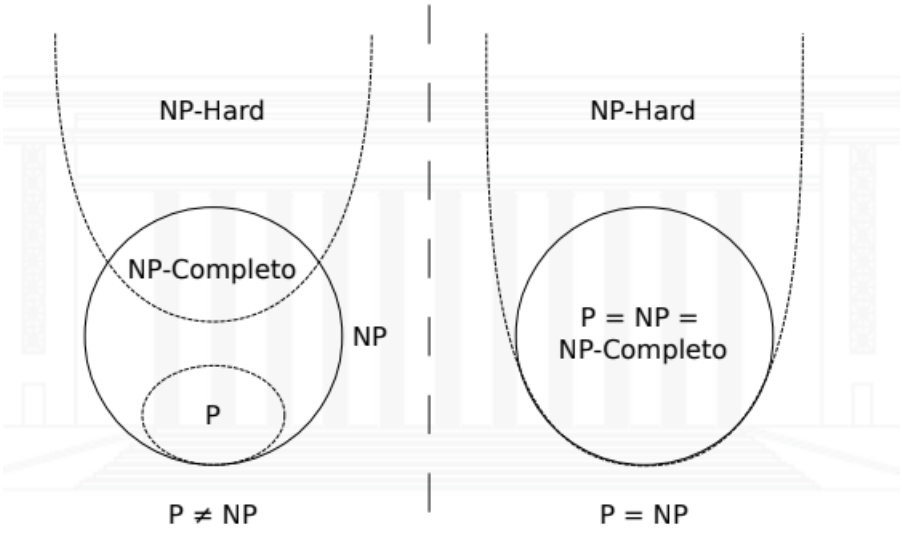


Figura 3: Complejidades

Del lado izquierdo podemos visualizar las relaciones si $P \neq NP$.

Para este caso se distinguen las siguientes relaciones.

- P es un subconjunto de NP. Como vimos antes P está incluido en NP.
- Ningún problema NP-Hard es P.
- Los problemas de tipo NP-C constituye la intersección entre los problemas de tipo NP (pero no del tipo P) y NP-Hard.

Del lado derecho podemos visualizar las relaciones si $P \equiv NP$. Para este caso se distinguen las siguientes relaciones.

- Existen dos conjuntos: NP-Hard y NP=P=NP-C.

3. Segunda Entrega

3.1. Comentarios: Un proyecto distribuido en dos

Incluyen una seria de presupuestos para su propuesta. entre ellas "Sean t1, t2 y t3 tres tareas tales que t3 depende de t1 y t2 para realizarse. Entonces, no hay transitividad entre las tareas, es decir, es suficiente que t2 se realice antes que t3 y t1 antes de t2 (no hay vinculo directo entre t3 y t1).". Esto no es necesariamente así. Mas allá de eso indican que no pueden llegar a una solución factible.

Consideren evaluar el problema como uno de corte mínimo

3.2. Correcciones: Un proyecto distribuido en dos

3.2.1. Estrategia redes de flujo para problema de asignación de tareas

La estrategia que se presenta como solución a problemas de la familia de problemas del enunciado se basa fuertemente en 2 conceptos:

- redes de flujo para representar los datos de los equipos, esto es, los costos de desarrollo de cada equipo, los costos de traspasos entre equipos y la dependencia entre tareas.
- el algoritmo de *Ford-Fulkerson* para determinar cómo se deben repartir las N tareas entre los 2 equipos, qué traspasos se van a realizar y el costo total de la operación.

Por lo tanto, la estrategia presentada para solucionar los problemas de este tipo se dividen en dos etapas:

1. la modelización del problema en una estructura de grafo
2. el procesamiento de dicho grafo con una adaptación del algoritmo de *Ford-Fulkerson*

ETAPA 1: Construcción del grafo

La etapa de construcción del grafo no es trivial y esta marcada por la necesidad de poder distinguir las tareas que se asignarán a cada equipo. Además, esta construcción debe considerar las dependencias entre las tareas, esto es, si la tarea t_i se vincula con otra tarea t_j o inclusive mas de una. Las hipótesis que se deben cumplir para representar los datos recibidos en un modelo de grafos satisfactoriamente son las siguientes:

- Existen 2 equipos exclusivamente (la cantidad de equipos no es mayor ni menor).
- Una tarea realizada por un equipo u otro es indistinta. Esto quiere decir que ambos equipos son igual de eficientes y toman la misma cantidad de tiempo, resultando en un producto igual. Si hubiese alguna diferencia entre las tareas realizadas por el equipo 1 o 2, dicha diferencia solo se refleja en el costo por equipo que se brinda como dato.
- Las tareas se completan sin errores ni fallas por cualquiera de los dos equipos.
- Los costos de producción y traspasos son fijos e inmutables.
- Todas las tareas pueden ser realizadas por ambos equipos. Esto quiere decir que si existe una tarea t_i la misma puede ser asignada tanto al equipo 1 como al 2. La decisión de asignar la tarea a alguno de los dos equipos depende exclusivamente de los costos de cada equipo y el costo de traspaso (si lo hubiera) y no por falta de conocimiento y/o competencias por parte de alguno de los equipos.

A continuación se indican los pasos a seguir en la construcción del grafo y la explicación de cada paso:

1. Dado que cada tarea puede ser realizada por cualquiera de los dos equipos, se construye un par de nodos N1 y N2 para cada una de las tareas, donde N es el nombre la tarea y N1 indica que la tarea N la realiza el equipo 1 y N2 indica que la tarea N la realiza el equipo 2.

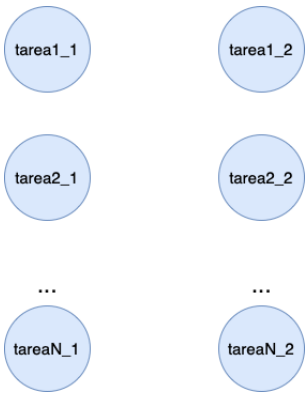


Figura 4: Representación de tareas por equipo en forma de nodos

2. Se crean 2 nodos S y T. El nodo S representa las tareas que alcanzan al equipo 1 y el nodo T las tareas que alcanzan al equipo 2.

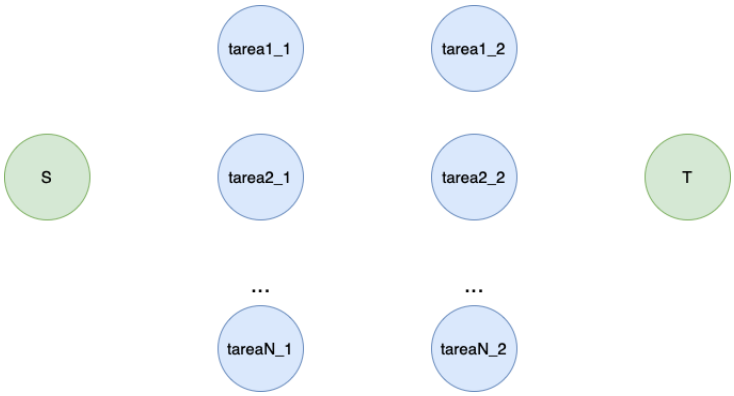


Figura 5: El nodo fuente se asocia a las tareas que realiza el equipo 1 el sumidero a las tareas que realiza el equipo 2

3. Se vincula cada nodo S y T con los nodos N1 y N2 respectivamente (N={tarea1, tarea2,.., tarean}). El peso de cada eje se define de la siguiente manera:

   - en el caso de los ejes S-N1, el peso del eje es el costo de asignar la tarea N al equipo 1.
   - para los ejes N2-T, el peso de cada eje es el costo de asignar la tarea N al equipo 2.

   El sentido de los ejes va desde S-T.

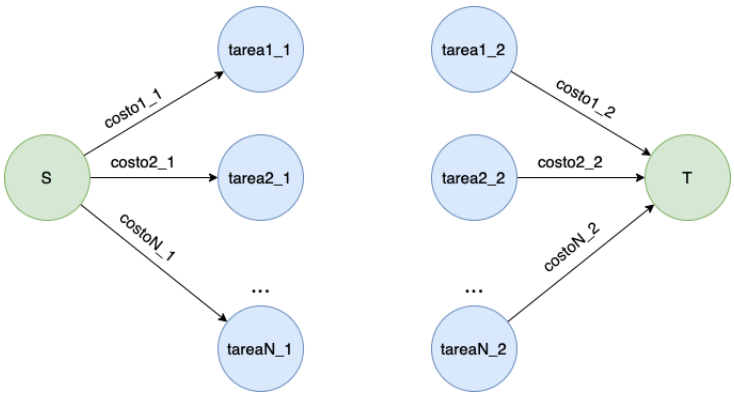


Figura 6: Costos de cada tarea, por equipo

4. Se traza un eje entre cada tarea N1 y N2 (en el sentido del flujo), con peso muy grande (que en el contexto del problema, sera un valor inalcanzable). Con este artilugio, se impide que se corte dicho eje, lo cual no tendría sentido pues ese corte indicaría que la tarea N fue asignada a ambos equipos (absurdo). La idea de construir este eje auxiliar es vincular las tareas N, independientemente de quien la realice (i.e. la tarea N1 y N2 sigue siendo la tarea N).

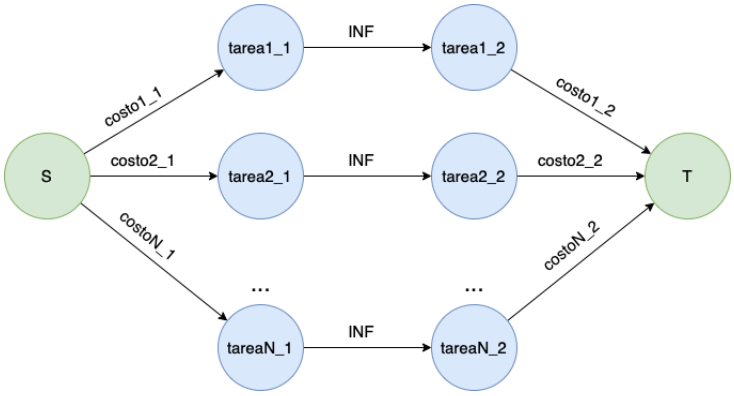


Figura 7: Vinculación entre cada tarea N

5. Finalmente, se construyen los ejes de dependencias entre tareas, las cuales están direccionadas en el sentido del flujo (i.e. de S a T). El peso de estos ejes es el costo del traspaso entre equipos.

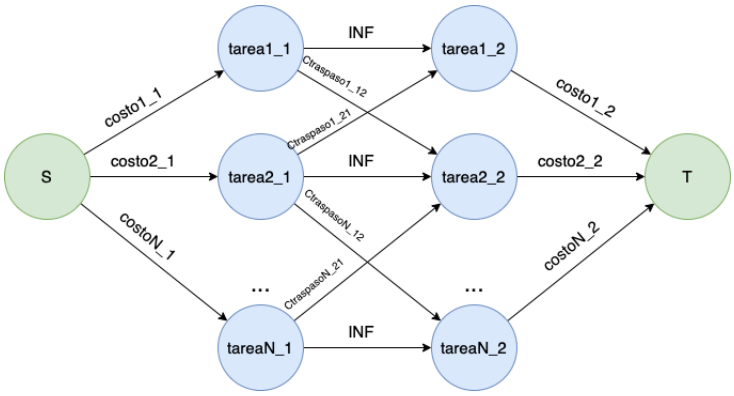


Figura 8: Representación final del problema en un grafo

ETAPA 2: Procesamiento del grafo por algoritmo adaptado de *Ford-Fulkerson*

Una vez que se construye el grafo, se aplica el algoritmo de *Ford-Fulkerson* con algunas modificaciones, de modo tal que al finalizar todas las iteraciones, se obtiene la siguiente información:

- el costo total de todas las asignaciones (con traspasos incluidos)
- cómo asignar las tareas entre los 2 equipos
- qué traspasos se realizan

Para saber como se realiza la asignación de tareas a cada equipo, cuando ya no hay caminos disponibles S-T, se obtienen todos los nodos a los cuales se puede acceder desde S y se los guarda en una lista (aplicando una transformación para no obtener tareas repetidas). Estas tareas serán las asignadas al equipo 2. Su complemento son asignadas al equipo 1.

Finalmente, se devuelven:

- el array de tareas a realizar por el equipo 1
- el array de tareas a realizar por el equipo 2
- el valor del flujo (el costo)

A fin de visualizar la explicación presentada, se describe el siguiente ejemplo: supongamos que para un determinado problema, se construye el grafo (como se detallo en la **ETAPA 1**) y se aplica FF sobre el mismo, llegando al siguiente resultado:

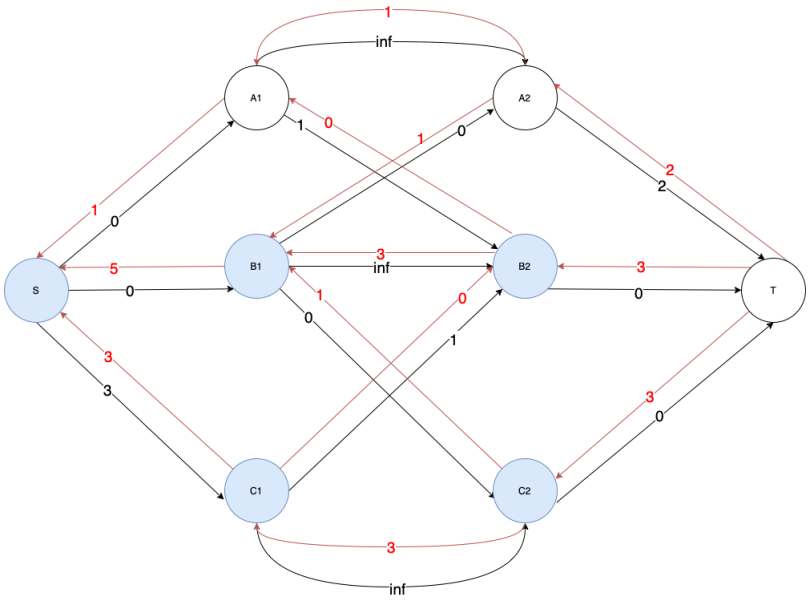


Figura 9: Ejemplo de problema cuando ya no hay mas caminos S-T disponibles

Tal como se puede ver, no hay mas camino S-T, por lo que se cortan las iteraciones. Acá es donde surgen las modificaciones que proponemos. Primero, cuando no hay mas caminos S-T, se itera sobre todos los nodos que son alcanzables desde S y se los guarda en una estructura de almacenamiento llamada **tareasEquipo2**. Del gráfico, se puede ver que esta lista seria la siguiente:

```
tareasEquipo2 = [B1, B2, C1, C2]
```

Luego, se aplica una transformación (del tipo *map_ reduce*) de modo tal que si el array guarda al par de nodos tarea_i1 y tarea_i2, se combinan ambas en una sola tarea, la tarea_i:

```
tareasEquipo2 = [B, C]
```

Cuando se tienen todas las tareas i alcanzables, se corta esta iteración. Luego, se construye otra lista (**tareasEquipo1**) que es el complemento de la lista **tareasEquipo2**:

```
tareasEquipo2 = [A]
```

Entonces, el algoritmo propuesto devuelve los siguientes resultados:

```
tareasEquipo2 = [B, C]
tareasEquipo2 = [A]
flujo = f
```

Si se desea ver un análisis detallado del funcionamiento de todo el algoritmo adaptado, dirigirse a la sección **Ejemplo paso a paso de la solución**.

3.2.2. Pseudo código de la solución

Pseudo código de la construcción del grafo

```
Sean N = {tarea_1, tarea_2, .., tarea_n} el conjunto de tareas a realizar
Sean dos equipos 1 y 2, indistinguibles (en eficiencia y velocidad) con costos
distintos (o no) para realizar las tareas del conjunto N.
Para cada tarea i en N
    crear 2 nodos tarea_i1 y tarea_i2
Crear 2 nodos S y T.
Para cada nodo tarea_i1
    Crear un eje entre S y la tarea_i1 y asignarle el costo de que
    la tarea_i la realice el equipo 1
Para cada nodo tarea_i2
    Crear un eje entre tarea_i2 y T y asignarle el costo de que
    la tarea_i la realice el equipo 2
Para cada nodo tarea_i1
    Si hay una dependencia de la tarea_i1 con la tarea_j2 (i!=j):
        crear un eje del nodo tarea_i1 hacia el nodo tarea_j2
        crear un eje del nodo tarea_j1 hacia el nodo tarea_i2
        setearle el mismo costo de trasferencia a ambos ejes
Para cada par tarea_i1-tarea_i2:
    crear un eje con costo infinito
```

Pseudo código del algoritmo de resolución sobre el grafo (Ford-Fulkerson) El siguiente pseudo código es una adaptación del conocido algoritmo de Ford-Fulkerson. El cambio esta motivado por la necesidad de devolver, no solo el flujo total (el cual es un indicador del costo del proyecto) sino también la información de cómo se reparten las tareas entre los dos grupos.

```
FF_Adaptado(G):
flujo_total = 0
Para cada eje (u, v) en G:
    flujo(u, v) = 0
Mientras haya un camino, P, de s -> t en el grafo residual G_f:
    b = bottleneck(P)
    flujo_total += b
Para cada eje (u, v) en P:
    Si (u, v) es un eje hacia adelante:
        flujo(u, v) += b
    Sino:
        flujo(v, u) -= b
tareas_equipo2 = [ ]
Mientras haya un nodo i sin visitar:
    tareas_equipo2.agregar(i)
Reducir (tareas_equipo2) (deja 1 sola tarea i de cada par de tarea i1 e i2)
tareas_equipo1 = complemento(tareas_equipo2) (guarda las tareas que no estan
en tareas_equipo2)
Devolver flujo_total, tareas_equipo1, tareas_equipo2
```

3.2.3. Complejidad

La complejidad del algoritmo propuesto puede separarse en dos partes: la complejidad en la construcción de la instancia del problema (el grafo) y la complejidad del posterior procesamiento con el algoritmo de *Ford-Fulkerson* adaptado.

Complejidad de la construcción del grafo Sea n la cantidad de tareas a repartir entre los dos equipos y k la cantidad de traspasos posibles.

La complejidad de la construcción del grafo se desglosa de la siguiente forma:

- 2 nodos por tarea ->$O(2n)$
- 2 nodos s y t ->$O(2)$
- n ejes para vincular S con los nodos tarea del equipo 1 y n ejes para vincular los nodos tarea del equipo 2 con T ->$O(2n)$
- un eje por cada par de tareas ->$O(n)$
- 2 ejes por cada dependencia entre tareas ->$O(n^2)$

Por lo tanto, la complejidad espacial en la construcción del grafo es $O(n^2)$ y la complejidad temporal es $O(n^3)$.

Complejidad del procesamiento del grafo La complejidad del procesamiento del grafo se desglosa de la siguiente forma:

- la cantidad de vértices $|V|$ es menor a la de ejes $|E|$
- el grafo residual tiene, a lo sumo, $2|E|$ ejes
- buscar los caminos de aumento $\rightarrow O(|V| + |E|) \sim O(|E|)$
- la función $augment(f, P)$ toma a lo sumo $O(|V|)$
- construir un grafo residual toma $O(|E|)$
- el proceso itera a lo sumo C veces, donde $C = \sum C_e$ (C :capacidad total) $\rightarrow$ obs: va a iterar C veces si las capacidades de los demás ejes transportan de a 1 unidad
- agregar cada nodo alcanzado desde S tiene costo $O(|E|)$

La complejidad del algoritmo es $O(|E|C)$

3.2.4. Ejemplo paso a paso de la solución

A continuación, se aplica la solución presentada sobre el caso del enunciado y se muestra su funcionamiento paso a paso.

Sean las tareas a realizarse: A, B, C y D. Sean, además, los equipos 1 y 2 los encargados de realizar cualquiera de las tareas, con un costo asociado para cada tarea. Sea el archivo que guarda esta información el siguiente:

```
A,1,2,B,2
B,5,10,C,3,D,1
C,4,4,D,2
D,1,3
```

La solución presentada tienen 2 etapas: la construcción del grafo y la aplicación del algoritmo de *Ford-Fulkerson* sobre dicho grafo.

Construcción del grafo : A continuación se detallan los pasos y los criterios que se toman en la construcción del grafo:

1. Se construye un par de nodos para cada una de las tarea A, B, C y D. En este caso, A1 y A2, B1 y B2, C1 y C2, D1 y D2.
2. Se crea un nodo fuente S y otro sumidero T

3. Se vincula el nodo S con los nodos N1 y el nodo T con los nodos N2, con N = {A,B,C,D}

   El resultado parcial de estos 3 primeros pasos es el siguiente:

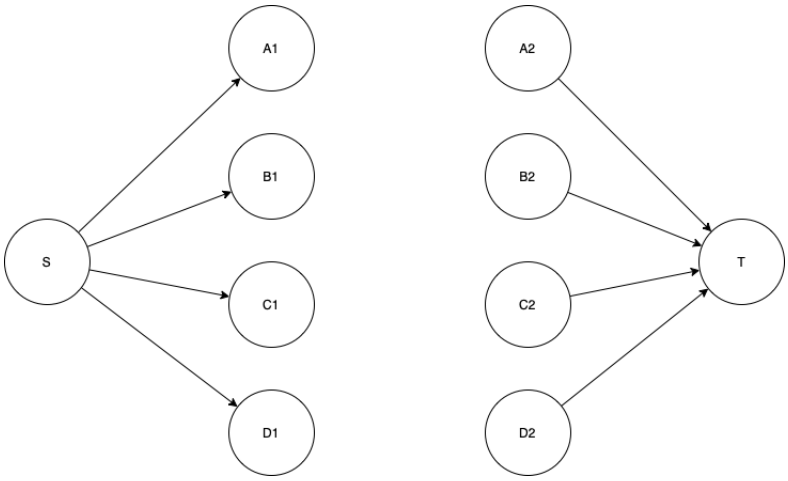


Figura 10: Pares de nodos construidos por cada tarea por equipo y vinculados a los nodos fuente y sumidero correspondientemente

4. Se le asigna a cada eje S-N1 un peso equivalente al costo de asignarle la tarea N-ésima al equipo 1. Análogamente, el valor de cada eje N2-T es el costo de asignarle la tarea N-ésima al equipo 2.

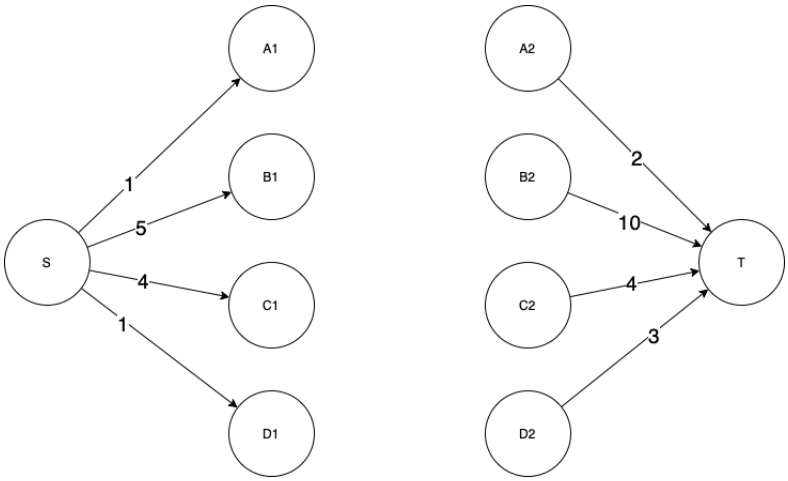


Figura 11: El valor de los ejes S-N1 y N2-T, con N={A,B,C,D} es el costo de asignarle la tarea N-ésima al equipo 1 y 2 respectivamente

5. Se construyen ejes para vincular la tarea N1 con N2, con N={A,B,C,D} y se les asigna un peso infinito (dentro del dominio del problema), de modo tal que por ese eje no se genere el efecto *bottleneck*.

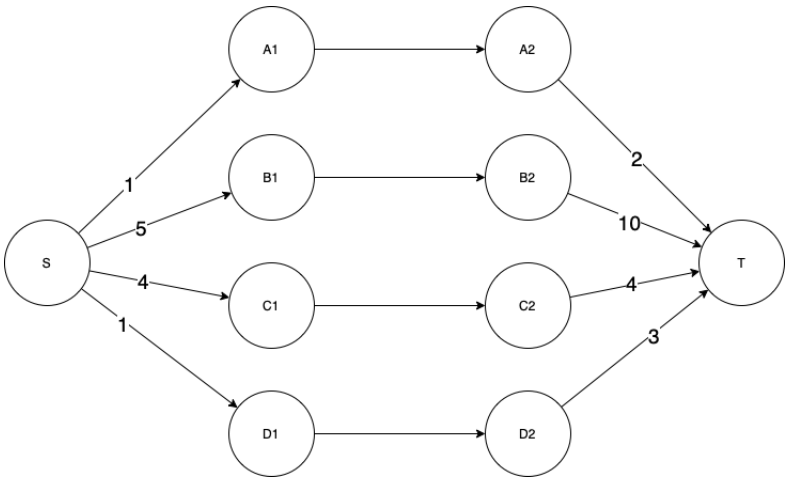


Figura 12: Se vincula N1 con N2 (con N={A,B,C,D}) a través de un eje con un peso que se puede leer como infinito en el dominio del problema

6. Finalmente, se crean los ejes que representan las dependencias dentro del dominio del problema. Para cada dependencia, se crea un eje que va desde el nodo N1 al nodo M2 (N != M), con peso igual al costo de cambiar de manos durante la ejecución del proyecto.

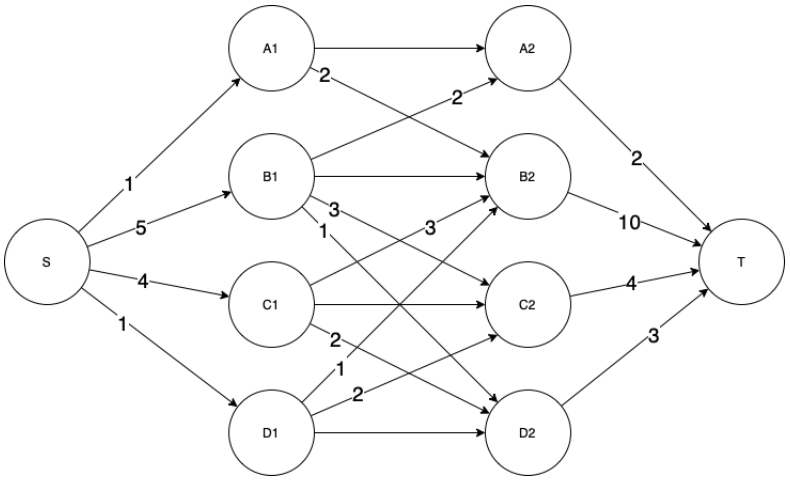


Figura 13: Se representan las dependencias como ejes que van desde N1 hacia M2 (N!=M) y el peso de dicho eje es el costo de cambiar de equipo. Observar que el sentido de todos los ejes indica el flujo del grafo.

Aplicación del algoritmo de *Ford-Fulkerson* : Ahora, se muestra un paso-a-paso de la aplicación de FF y la interpretación de los resultados:

1. Se construye el grafo residual, el cual resulta ser el siguiente:

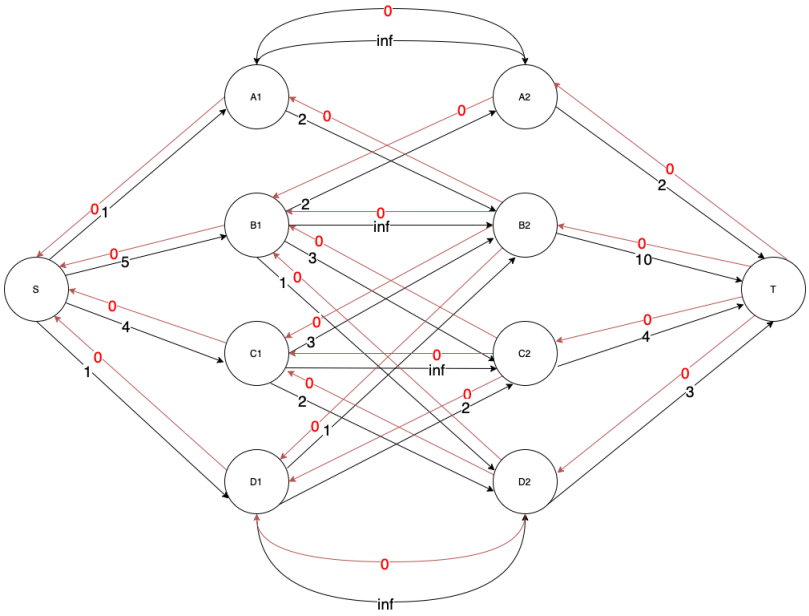


Figura 14: Grafo residual del grafo original.

Tal como puede verse, los ejes inversos (en colorado) en cada caso se han inicializado con valor 0.

2. Se procede a elegir un camino S-T, en este caso, el flujo de aumento es P1 = {S,A1,A2,T}. En este caso, el *bottleneck* es 1, por lo que se suma al flujo acumulado (que inicialmente es nulo) y se procede a actualizar el flujo de todos los ejes que conforman P1:

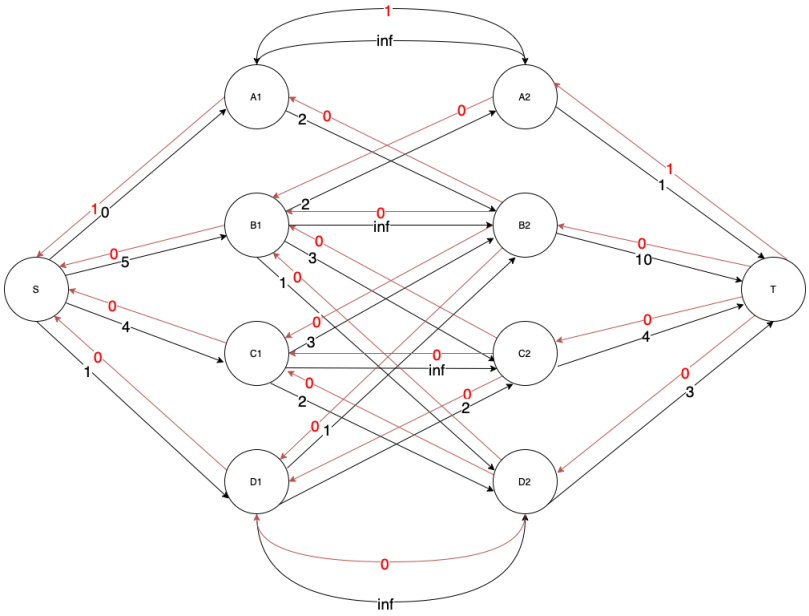


Figura 15: Actualización del grafo residual para P1. Flujo parcial = 1

3. Se elige un nuevo camino S-T, P2 = {S,B1,B2,T}. Ahora, el valor de *bottleneck* es 5 y se suma al flujo acumulado, resultando en un nuevo flujo acumulado parcial de valor 6. Se actualiza el grafo según P2:

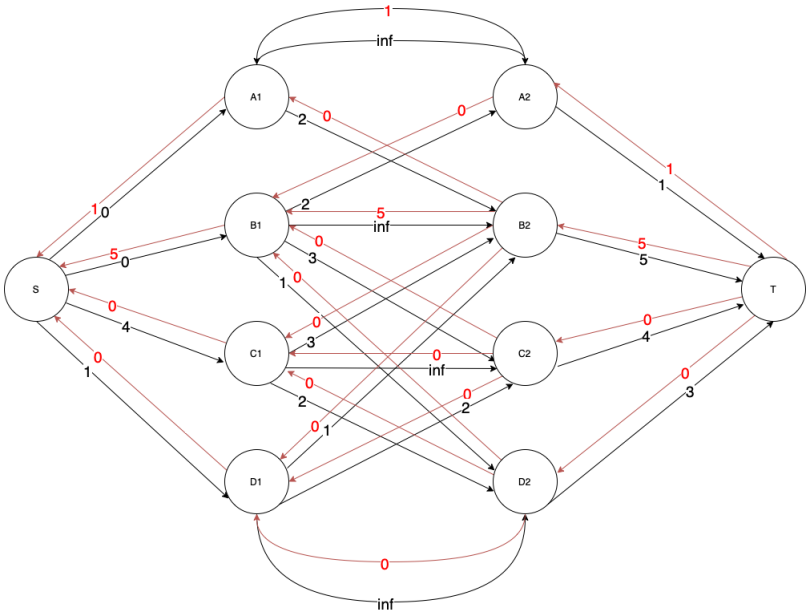


Figura 16: Actualización del grafo residual para P2. Flujo parcial = 6

4. Como aun quedan caminos disponibles que vayan de S-T, se elije uno mas, P3={S,C1,C2,T}. En este caso, *bottleneck(P3) = 4*: se suma dicho valor al flujo acumulado (flujo parcial = 10) y se actualizan los ejes de P3:

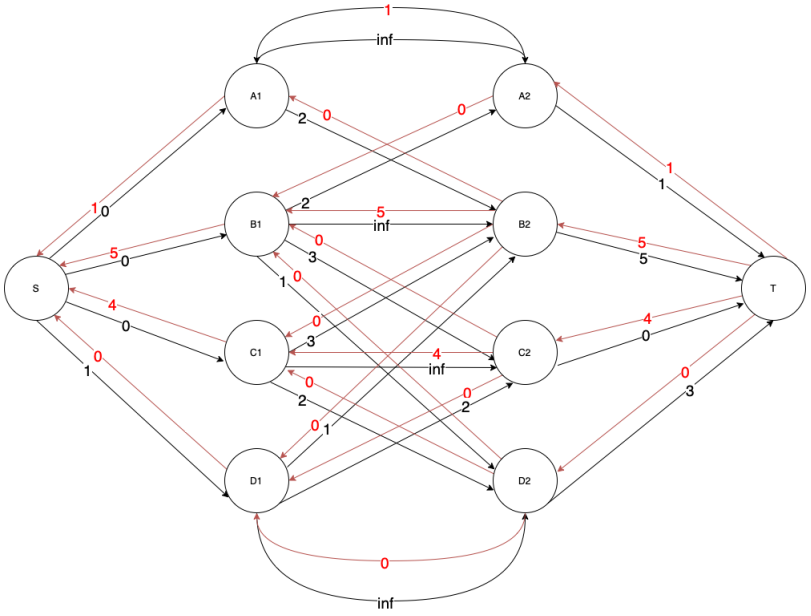


Figura 17: Actualización del grafo residual para P3. Flujo parcial = 10

5. Se busca un nuevo camino que lleve de S a T, ahora P4 = {S,D1,D2,T}. En este caso, *bottleneck(P4) = 1* y el flujo acumulado es 11. Se actualiza el grafo residual:

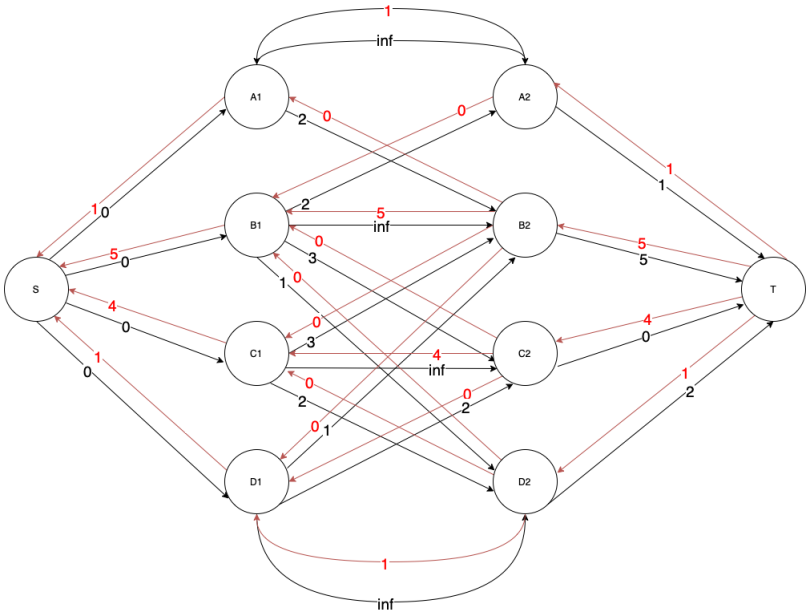


Figura 18: Actualización del grafo residual para P4

Como se puede notar, ya no hay mas caminos de aumento de flujo por lo que se corta la interacción, con un flujo total de 11. Además, se logra el siguiente corte:

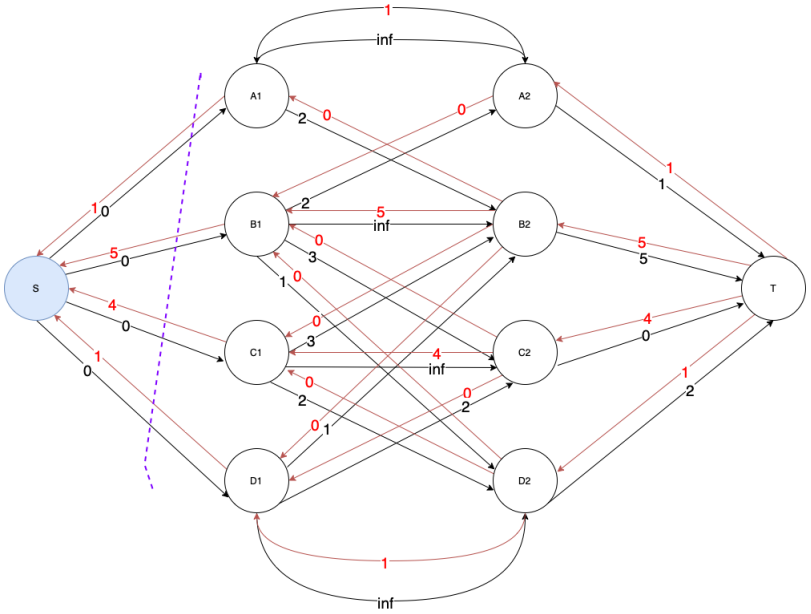


Figura 19: Grafo residual final y corte del mismo. Flujo total = 11

Como ya no hay mas caminos S-T, se itera buscando nodos alcanzables desde S. Tal como se puede ver, no hay nodos alcanzables desde S por lo que el array tareasEquipo2 esta vacia y tareasEquipo1 = [A, B, C, D]. Se devuelve el valor del flujo y los dos vectores, es decir:

```
tareasEquipo1 = [A,B,C,D]
tareasEquipo2 = [ ]
flujoTotal = 11
```

finalizando el algortimo.

3.2.5. Programación del algoritmo

Para ejecutar el programa es necesario instalar las librerías igraph y matplotlib para la visualización del grafo. Esto se hace a través del los siguientes comandos:

```
pip install igraph
pip install matplotlib
```

El programa se ejecuta de la siguiente manera:

```
python3 ./tp3tda.py file_path
```

Para una ejecución online sin ninguna instalación se puede correr desde Google Collab: Clickeando aqui

3.2.6. Complejidad del programa

3.3. Comentarios: Mas alla de Bellman-Ford

La reduccion polinomial es correcta. aunque falta indica que pasa con el parametro Ç"(En la definicion del problema "Queremos saber si existe un camino simple de costo menor a C que los una"). Es el mismo C? cambia?

No es muy claro el certificador polinomial de "PCL". Çonsideremos que se quiere encontrar la solución al grafo G(V1,Vn). Esto puede ser realizado en tiempo polinomial."< − como? esto es lo que tienen que explicar. que se verifica y como lo verifica? La reduccion polinomial es correcta

Bien que afirma que no es NP-C con los probado hasta ahora. Pero el enunciado pide que si la respuesta es negativa agregue lo que falta para demostrarlo. Falta ese punto

Es correcto utilizar un algoritmo "A" como parte de una reduccion. las reducciones polinomiales vinculan problemas. no algoritmos

Las explicaciones de clases de complejidad y su relacion son correctas.

Referencia de donde obtine las imagenes que no son de su autoria

3.4. Correcciones: Mas alla de Bellman-Ford

3.4.1. Correcciones Respuesta 2.2

Procedemos a probar que el Problema de camino más largo es NP. Llamaremos PCL al Problema de camino más largo. Pensemos en una posible solución al problema.
Dada una instancia de un grafo G.
Sea K un entero positivo igual al K del enunciado del problema de decisión dado.
Sea C certificador: una lista de aristas E de tamaño de al menos K y máximo |G|.
Se puede examinar en tiempo polinomial si esta lista constituye un camino simple en el grafo G. Y además que el subconjunto en total es de tamaño al menos K y como máximo |G|.
De esta manera podemos concluir que PCL es NP.

3.4.2. Correciones Respuesta 2.3

Agregamos lo que resta para probar que es NP-Completo.
Para ello es necesario probar que el problema es NP es decir, que existe un algoritmo que certifique (o verifique) a “SSPGNC”.
Recordamos el problema de decisión: "Dado un grafo con peso en sus ejes y un par de nodos al que llamaremos origen y destino. Queremos saber si existe un camino simple de costo menor a C que los una".
Ahora procedemos a probar que el problema es NP:
Sea K un entero = C.
Sea S un certificador: subconjunto de aristas con peso.
Sean s origen y t destino, dos nodos que pertenecen al grafo G.
Se puede verificar en tiempo polinomial que S constituye un camino simple que une de s a t, de costo menor a K.
Con esto se prueba que el problema es NP.

3.4.3. Correcciones Respuesta 2.4

Para dar la explicación utilizaremos el siguiente gráfico para ser más claros en el desarrollo de la respuesta. Además usaremos la notación PCL = El problema del camino más largo.
Llamamos Z al conjunto de todos los problemas de tipo NP.

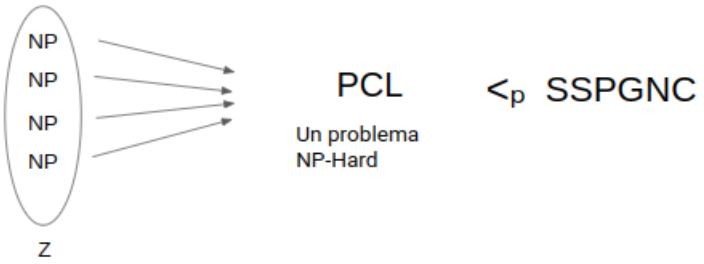


Figura 20: Diagrama

Como vemos en la imagen, tenemos muchos problemas del tipo NP que pueden reducirse a PCL $\in$ NP-Hard. De los puntos anteriores resueltos del enunciado, sabemos que PCL $\in$ NP-C, por lo tanto, PCL $\in$ NP-Hard. Más aún esto implica que todo problema NP puede ser reducido a PCL en tiempo polinomial y de allí esa parte del gráfico.
Ahora bien, en el **inciso 1.** reducimos PCL $<p$ SSPGNC, por lo que por transitividad,

- Z $<p$ PCL $<p$ SSPGNC

Por lo tanto,

- Z $<p$ SSPGNC

De manera tal que SSPGNC resulta NP-Hard.
Con esto en mente, avanzamos y vemos que el enunciado nos propone que existe un algoritmo eficiente que resuelve cualquier instancia de "SSPGNC". Llamamos A a ese algoritmo.

Observemos que esto indica que un problema NP-Hard (SSPGNC)puede ser resuelto por un algoritmo A en tiempo polinomial. Mas aun, todo problema NP puede ser resuelto por un algoritmo de orden polinomial. Es decir, se están resolviendo en tiempo polinomial a todos los problemas NP. Se sigue entonces que todo problema de tipo NP sería un problema de tipo P. Finalmente, bajo esta existencia de ün algoritmo eficiente que resuelve cualquier instancia de SSPGNC.[es]taríamos probando que P = NP. Es decir, se demostraría que todo problema NP puede ser resuelto en tiempo polinomial.

4. Tercera Entrega

4.1. Estrategia redes de flujo para problema de asignación de tareas

Para esta entrega, se planteo el problema de la asignación de tareas a través de una modelización distinta a la de la entrega anterior. Aunque se toman muchos de los supuestos realizados anteriormente, las tareas realizadas por ambos equipos, el traspaso de tareas y el calculo de costo se realiza utilizando otro grafo como se indica a continuación.

ETAPA I: Construcción del grafo

En primer lugar, a diferencia de la propuesta anterior, es este modelo las tareas a realizarse se modelan con un único nodo. Es decir que para las k tareas a realizarse por cualquiera de los dos equipos existen k nodos:

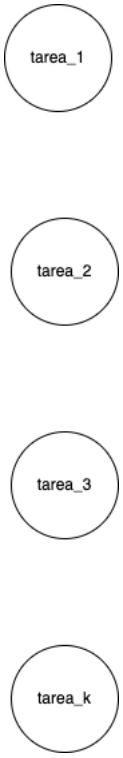


Figura 21: Tareas a realizarse

Luego, se agregan los nodos **fuente** y **sumidero** los cuales son asociados a los equipos 1 y 2 respectivamente:

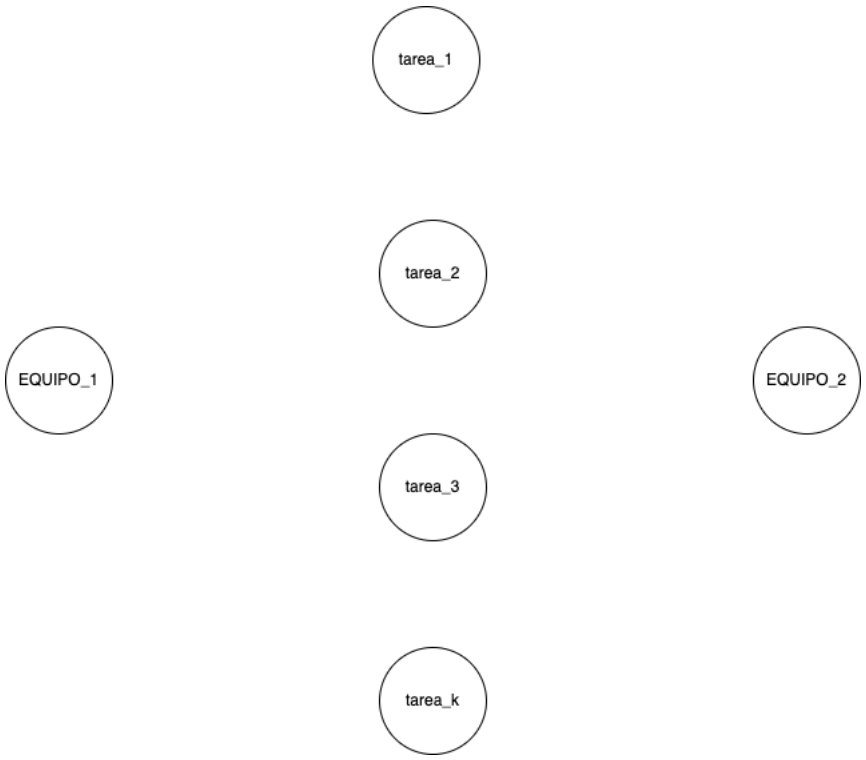


Figura 22: Equipos responsables de las tareas

Además, se vincula a cada equipo con cada tarea por medio de un eje, donde el peso del eje *(EQUIPO_1, tarea_j)* corresponde al costo de asignar la tarea j-esima al equipo 1 y el peso del eje *(tarea_j, EQUIPO_2)* es el costo de asignar la tarea j-esima al equipo 2:

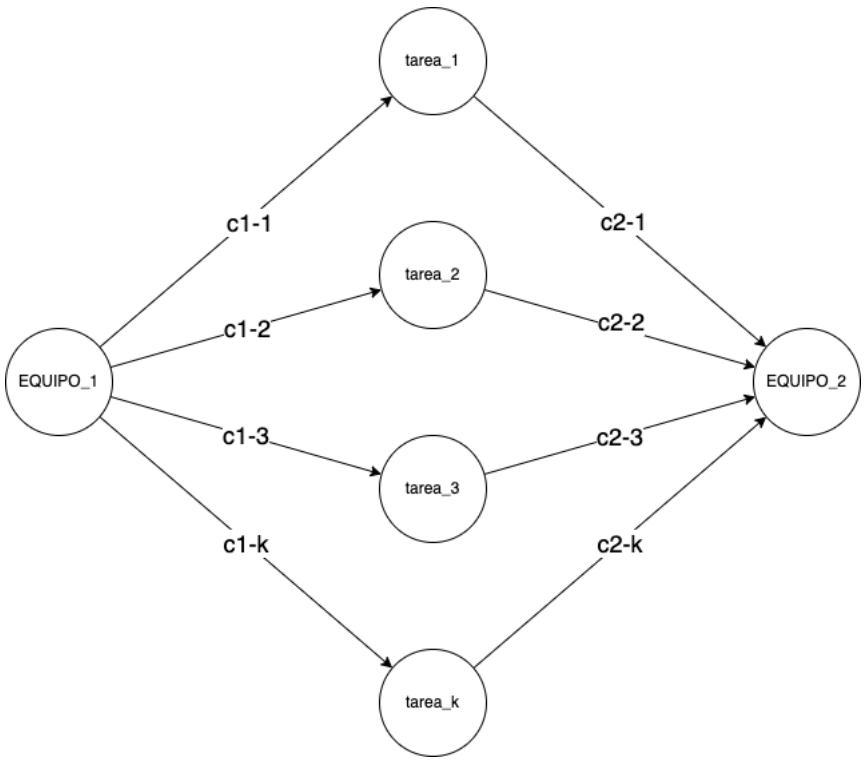


Figura 23: Equipos, tareas y costos de asignación de tareas

Finalmente, para modelar la dependencia entre tareas, se trazan ejes entre las tareas de modo tal que si la tarea j depende de la tarea i, entonces existirá un eje (i, j) y también un eje (j, i). También, el costo de estos dos ejes con direccionalidades inversas modelan el costo del traspaso entre los dos equipos (i.e. que la tarea i realizada por el equipo 1 se pase al equipo 2 para que realice la tarea j y viceversa):

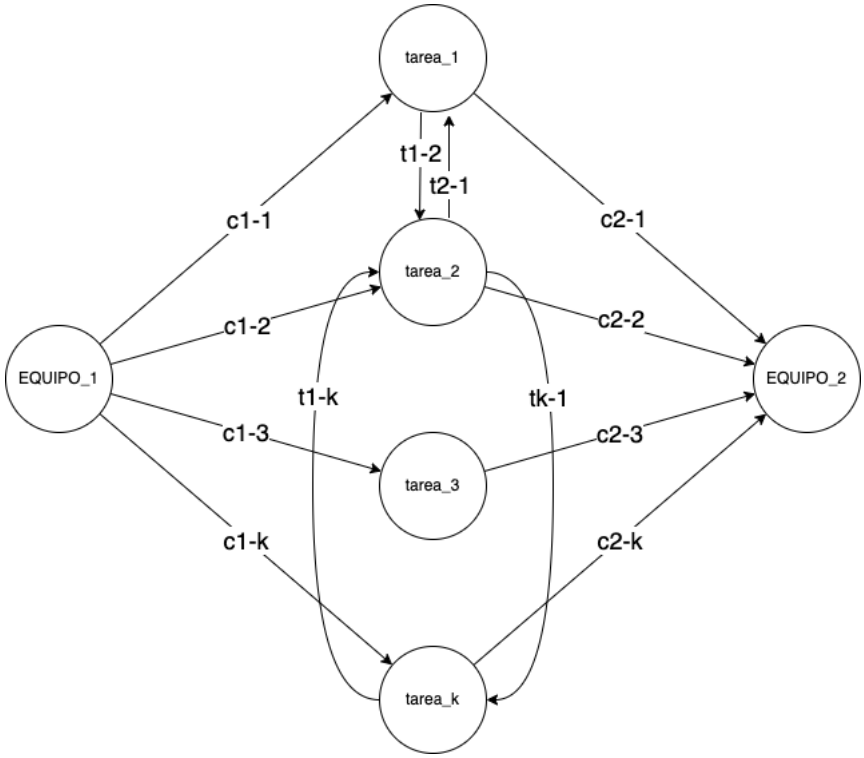


Figura 24: Equipos, tareas, costos de asignación de tareas y traspasos

Así, se obtiene la instancia del problema modelada con un grafo.

ETAPA 2: Procesamiento del grafo por algoritmo adaptado de *Ford-Fulkerson*

Ahora que se cuenta con el grafo, se aplica el algoritmo de *Ford-Fulkerson* a fin de obtener un corte mínimo (y los dos conjuntos asociados) calculando el flujo máximo. Este paso es similar al descrito en la segunda entrega solo que se realiza sobre el nuevo modelo de grafo que se describió en la ETAPA 1. A fin de mostrar su funcionamiento paso a paso, se toma una instancia de un problema, se lo modela y luego se lo resuelve por *Fork-Fulkerson* en la sección **4.2.1**.

4.2. Pseudo código de la solución

Pseudo código de la construcción del grafo

```
Sean N = {tarea_1, tarea_2, .., tarea_k} el conjunto de tareas a realizar
Sean dos equipos 1 y 2, indistinguibles (en eficiencia y velocidad) con costos
distintos (o no) para realizar las tareas del conjunto N.
Crear 2 nodos S y T.
Para cada tarea_i en N
    crear un nodo tarea_i
```

```
        crear un eje entre S y la tarea_i y asignarle el costo de que
        la tarea_i la realice el equipo 1
        Crear un eje entre la tarea_i y T y asignarle el costo de que
        la tarea_i la realice el equipo 2
    Para cada nodo tarea_i
        Si hay una dependencia de la tarea_i con la tarea_j (i!=j):
            crear un eje del nodo tarea_i hacia el nodo tarea_j
            crear un eje del nodo tarea_j hacia el nodo tarea_i
            setearle el mismo costo de trasferencia a ambos ejes
```

Pseudo código del algoritmo de resolución sobre el grafo

```
FF_Adaptado(G):
flujo_total = 0
Para cada eje (u, v) en G:
    flujo(u, v) = 0
Mientras haya un camino, P, de s -> t en el grafo residual G_f:
    b = bottleneck(P)
    flujo_total += b
Para cada eje (u, v) en P:
    Si (u, v) es un eje hacia adelante:
        flujo(u, v) += b
    Sino:
        flujo(v, u) -= b

tareas_equipo2 = [ ]
Mientras haya un nodo tarea i sin visitar:
    tareas_equipo2.agregar(i)
tareas_equipo1 = complemento(tareas_equipo2) (guarda las tareas que no estan
en tareas_equipo2)
Devolver flujo_total, tareas_equipo1, tareas_equipo2
```

4.3. Complejidad

Al igual que la propuesta anterior, la complejidad del algoritmo propuesto puede separarse en dos partes: la complejidad en la construcción de la instancia del problema (el grafo) y la complejidad del posterior procesamiento con el algoritmo de *Ford-Fulkerson* adaptado.

Sean N el conjunto de tareas, D el conjunto de dependencias entre tareas, V el conjunto de vértices y E el conjunto de aristas.

4.3.1. Complejidad temporal

Construcción del grafo:

- Para construir el grafo se itera sobre el conjunto de N tareas - O($|N|$). En cada iteración se asocian los nodos fuente y sumidero con las tarea correspondientes - O(1)-, asignando el peso respectivo a cada arista. O($|N|$)
- En otro ciclo (no anidado) se asocian los nodos con dependencias O($|N|$)

En total obtenemos una complejidad temporal para la construcción del grafo de O($|N|$).

Procesamiento del grafo: La complejidad del procesamiento del grafo se desglosa de la siguiente forma:

- La cantidad de vértices $|V|$ es menor a la de ejes $|E|$
- El grafo residual tiene, a lo sumo, $2|E|$ ejes
- Buscar los caminos de aumento $\rightarrow O(|V| + |E|) \sim O(|E|)$
- La función de aumento toma a lo sumo $O(|V|)$
- Construir un grafo residual toma $O(|E|)$
- El proceso itera a lo sumo C veces, donde $C = \sum C_e$ (C :capacidad total). Esto hace que la complejidad sea pseudo-polinomial.
- Agregar cada nodo alcanzado desde S tiene costo $O(|E|)$

En total obtenemos una complejidad temporal para el procesamiento del grafo de $O(|E|C)$.

Complejidad temporal: Finalmente la complejidad temporal del pseudocódigo es $O(|E|C + |N|)$

4.3.2. Complejidad espacial

Construcción del grafo: La complejidad de la construcción del grafo de la instancia del problema se desglosa de la siguiente forma:

- 1 nodo por tarea $O(|N|)$
- 2 nodos s y t $O(1)$
- 2n ejes para vincular s con los nodos tarea y los nodos tarea con t $O(2|N|)$
- 2 ejes por cada dependencia entre tareas $O(2|D|)$

Por lo tanto, la complejidad espacial en la construcción del grafo es $O(|N| + |D|)$.

Procesamiento del grafo: Además a la hora de procesar el grafo se crea un grafo residual que tiene las mismas dimensiones que el grafo original. Se agrega a la complejidad espacial $O(|N| + |D|)$.

Complejidad espacial: Finalmente la complejidad espacial del pseudocódigo es $O(|N| + |D|)$.

4.4. Ejemplo paso a paso de la solución

A continuación, se aplica la solución presentada en esta nueva entrega sobre el caso del enunciado y se muestra su funcionamiento paso a paso.

Sean las tareas a realizarse: A, B, C y D. Sean, además, los equipos 1 y 2 los encargados de realizar cualquiera de las tareas, con un costo asociado para cada tarea. Sea el archivo que guarda esta información el siguiente:

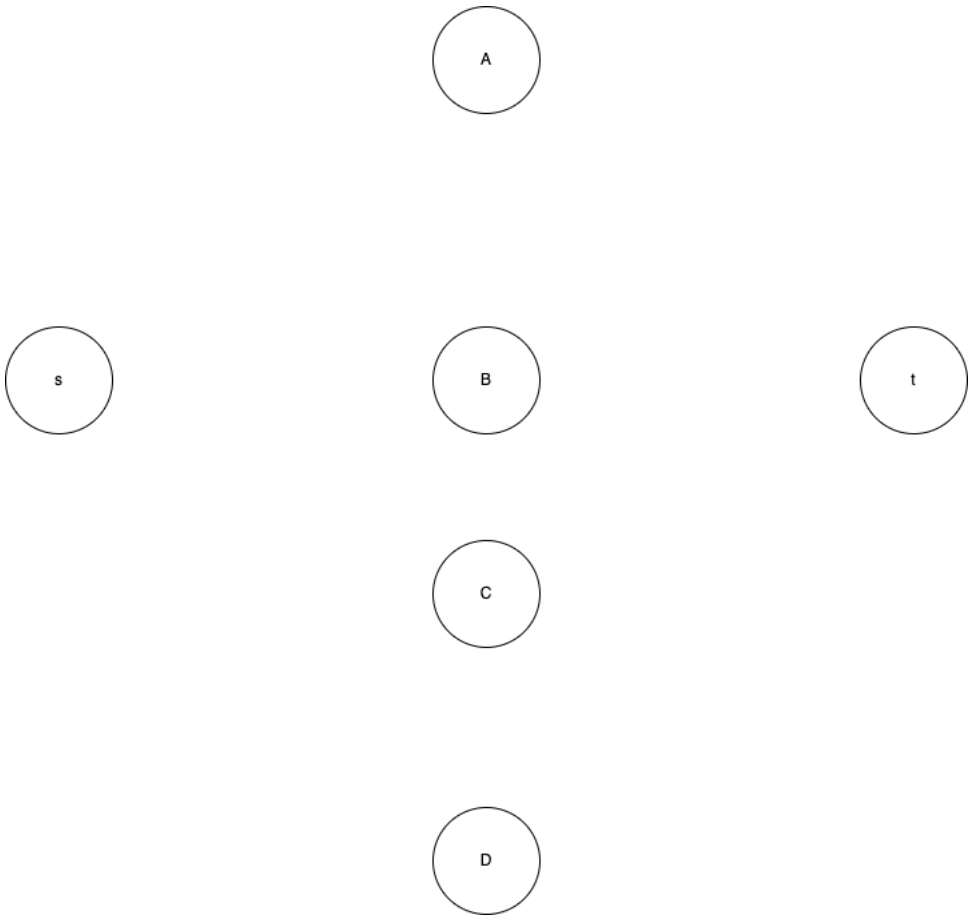
```
A,1,2,B,2
B,5,10,C,3,D,1
C,4,4,D,2
D,1,3
```

La solución presentada tienen 2 etapas: la construcción del grafo y la aplicación del algoritmo de *Ford-Fulkerson* sobre dicho grafo.

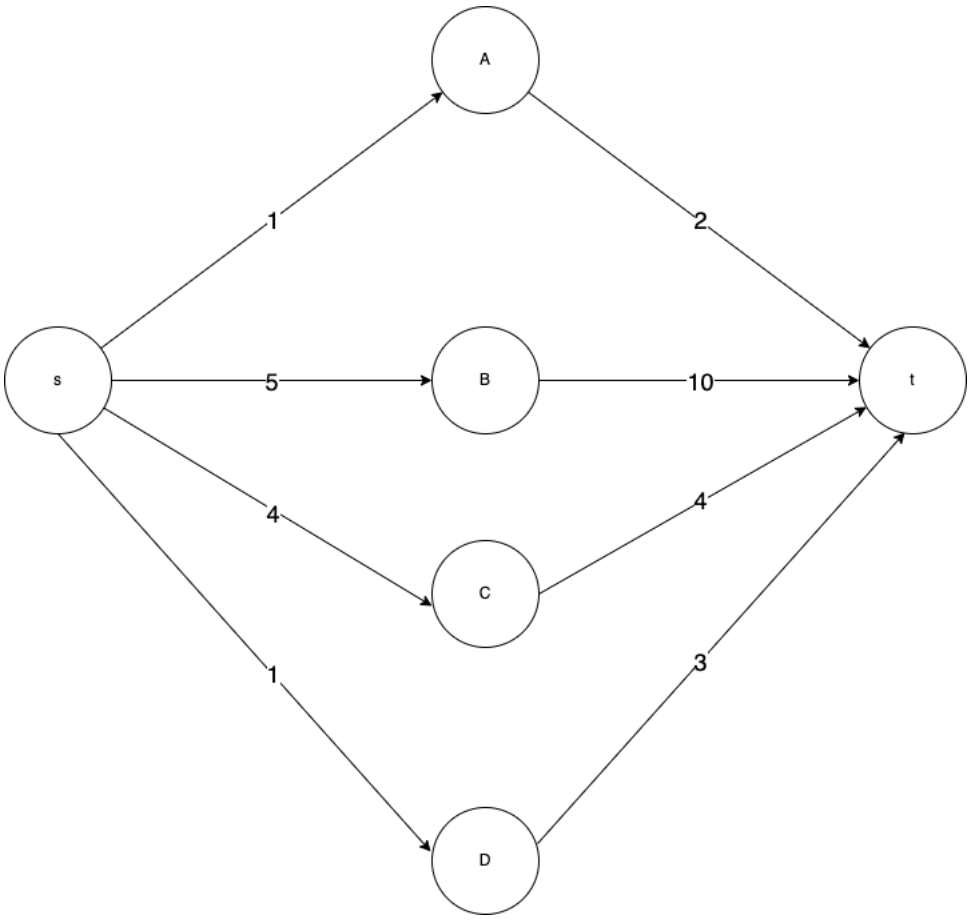
Construcción del grafo : A continuación se detallan los pasos y los criterios que se toman en la construcción del grafo:

1. Se crean 4 nodos para las 4 tareas: A, B, C y D:

2. Se crean 2 nodos s y t que representan a los equipos 1 y 2 respectivamente:



3. Se crean ejes $(s, tarea\_j)$ y $(tarea\_j, t)$ para $j = \{A, B, C, D\}$ y se le asignan por pesos los costos de que cada equipo realice la tarea j-esima:



4. Se crean ejes de ida y vuelta para las dependencias entre tareas y se le asigna por peso el costo de traspaso entre equipos:

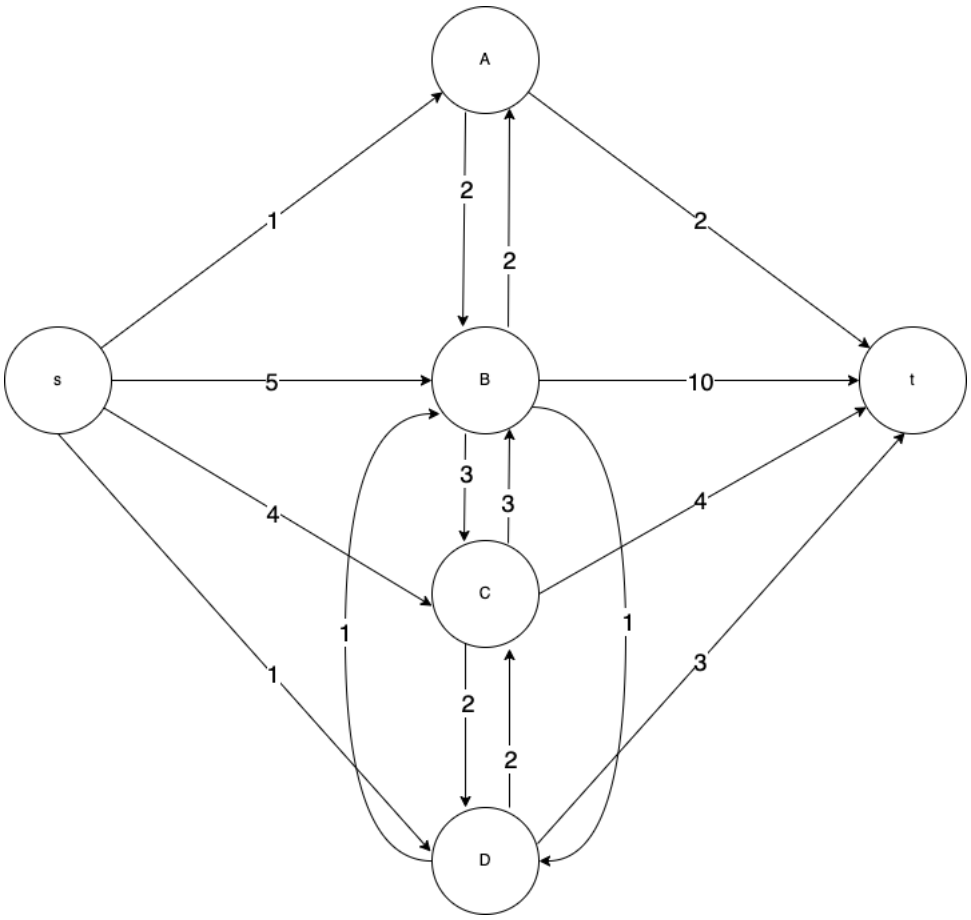


Figura 25: Instancia modelada

Aplicación del algoritmo de *Ford-Fulkerson* : Ahora, se muestra un paso-a-paso de la aplicación de FF y la interpretación de los resultados:

1. Se construye el grafo residual, el cual resulta ser el siguiente:

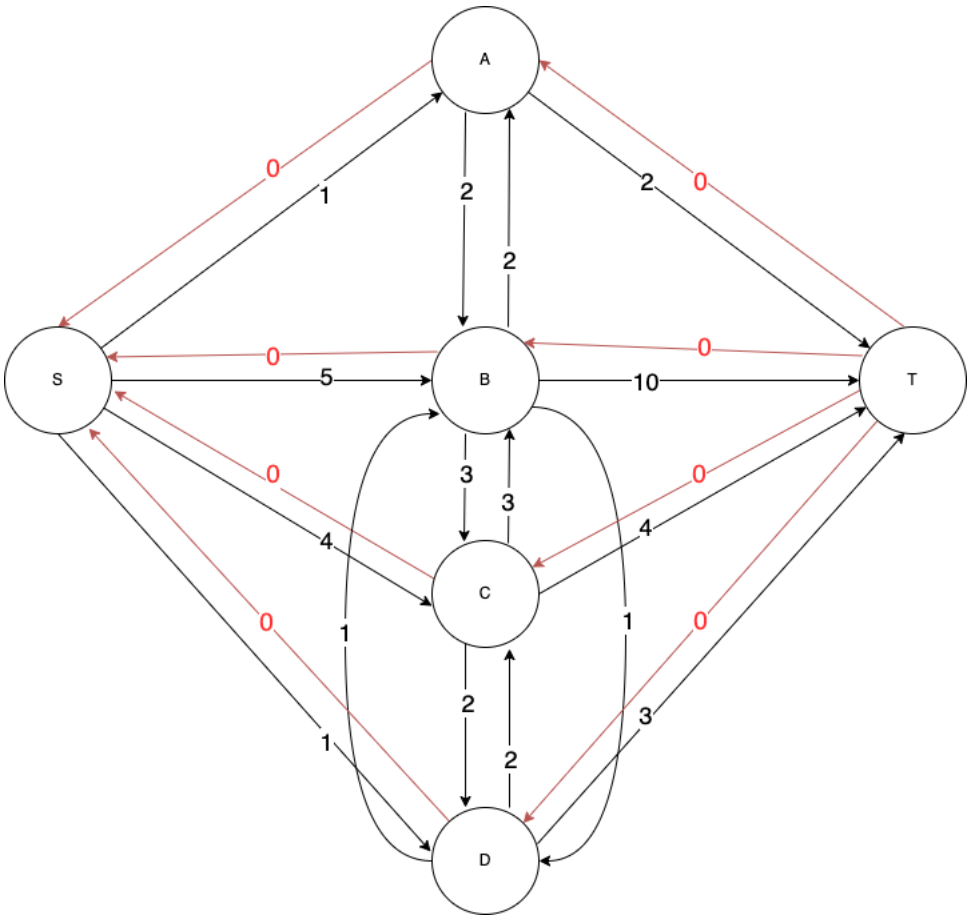


Figura 26: Grafo residual del grafo original.

Tal como puede verse, los ejes inversos (en colorado) en cada caso se han inicializado con valor 0.

2. Se procede a elegir el camino s-t P1 = {S,A,T}. En este caso, el *bottleneck* es 1, por lo que se suma al flujo acumulado (que inicialmente es nulo) y se procede a actualizar el flujo de todos los ejes que conforman P1 y sus inversos:

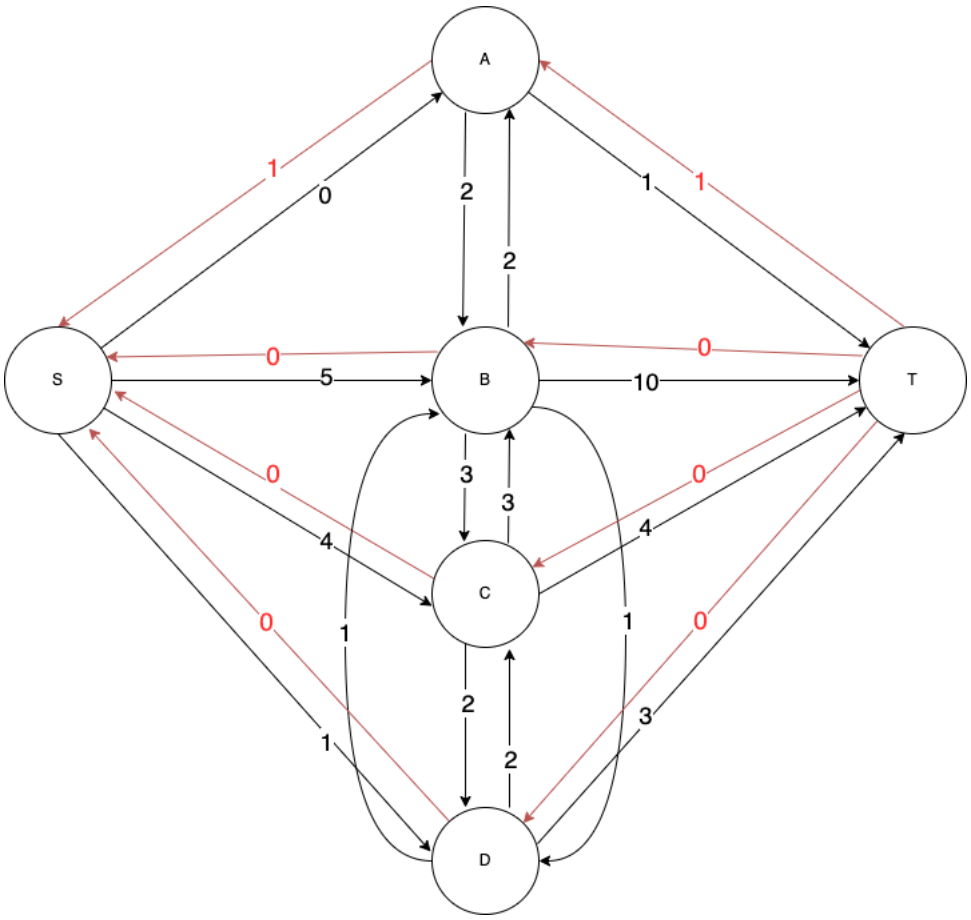


Figura 27: Actualización del grafo residual para P1. Flujo parcial = 1

3. Se elige un nuevo camino S-T, P2 = {S,B,A,T}. El valor del *bottleneck* es 1 y se suma al flujo acumulado, resultando en un nuevo flujo acumulado parcial de valor 2. Se actualiza el grafo según P2:

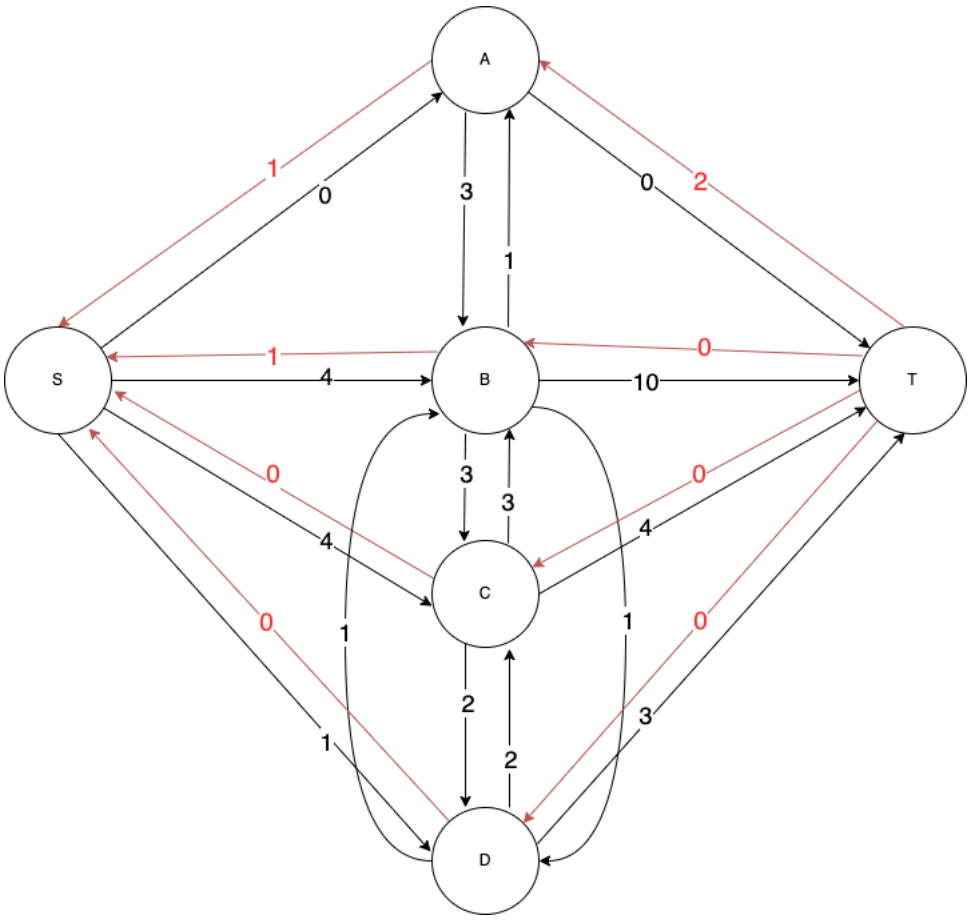


Figura 28: Actualización del grafo residual para P2. Flujo parcial = 2

4. Se observa que el nodo A ha saturado su arista saliente hacia T. Sin embargo, como aun quedan caminos disponibles que vayan de S-T, se elije alguno mas, P3={S,B,T}. En este caso, *bottleneck(P3) = 4*: se suma dicho valor al flujo acumulado (flujo parcial = 6) y se actualizan los ejes de P3:

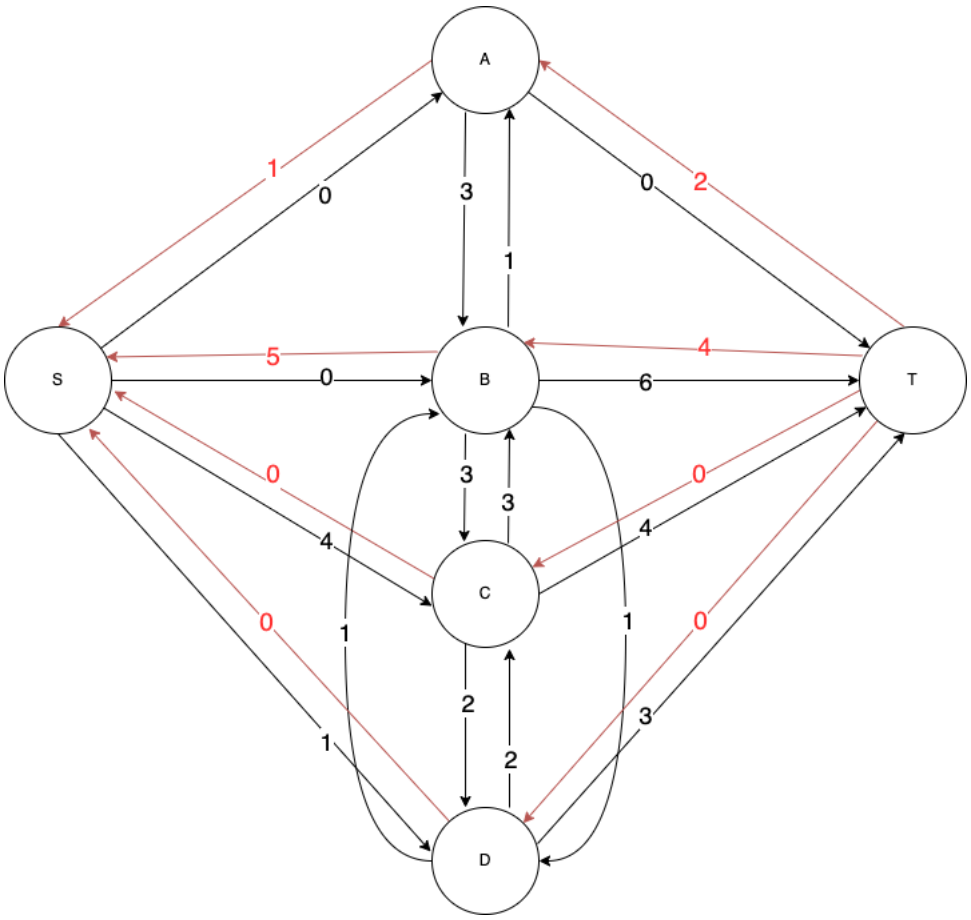


Figura 29: Actualización del grafo residual para P3. Flujo parcial = 6

5. Se busca un nuevo camino que lleve de S a T, ahora P4 = {S,C,B,T}. En este caso, *bottleneck(P4) = 3* y el flujo acumulado es 9. Se actualiza el grafo residual:

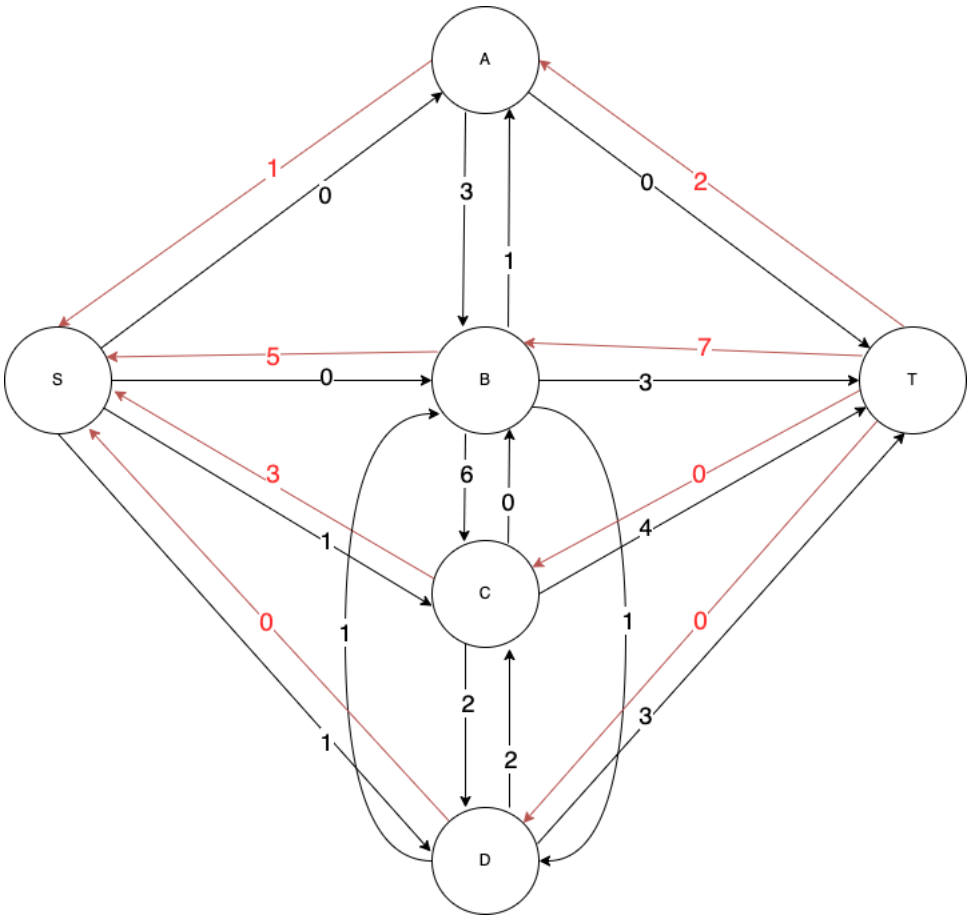


Figura 30: Actualización del grafo residual para P4. Flujo parcial = 9

6. Se busca un nuevo camino, P5 = {S,C,T}. En este caso, *bottleneck(P5) = 1* y el flujo acumulado es 10. Se actualiza el grafo residual:

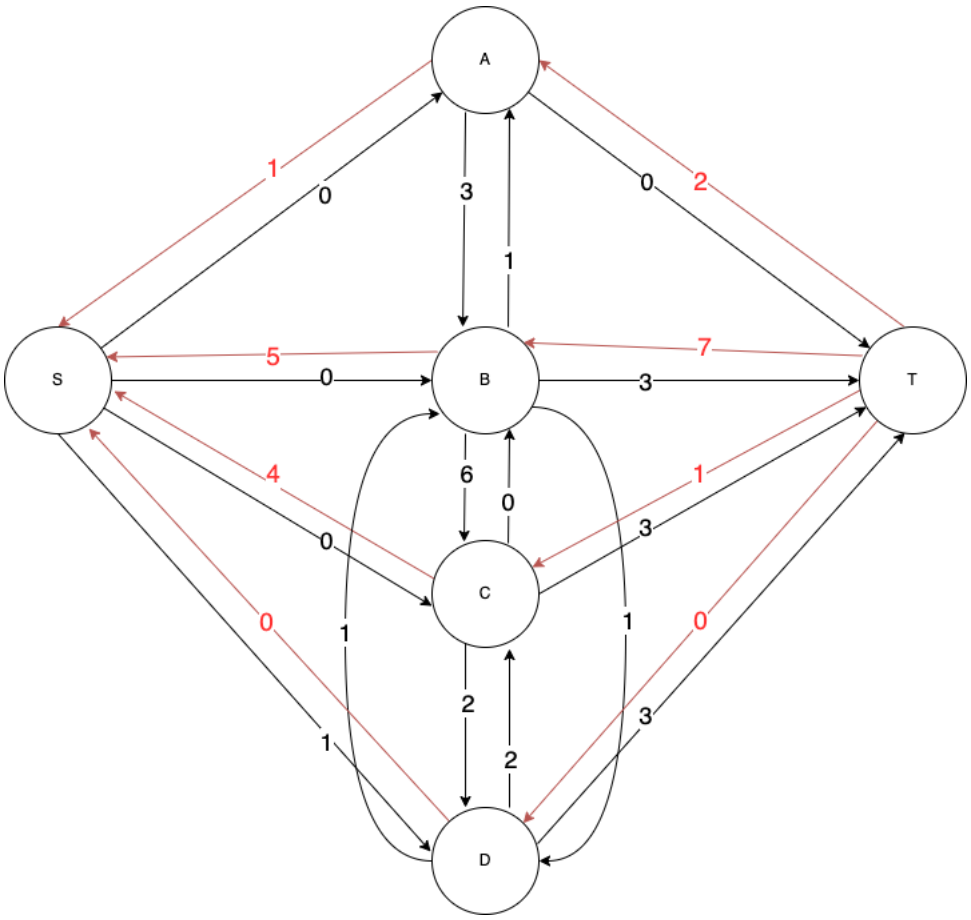


Figura 31: Actualización del grafo residual para P5. Flujo parcial = 10

7. Se busca otro camino de aumento, P6 = {S,D,T}. En este caso, *bottleneck(P6) = 1* y el flujo acumulado es 11. Se actualiza el grafo residual:

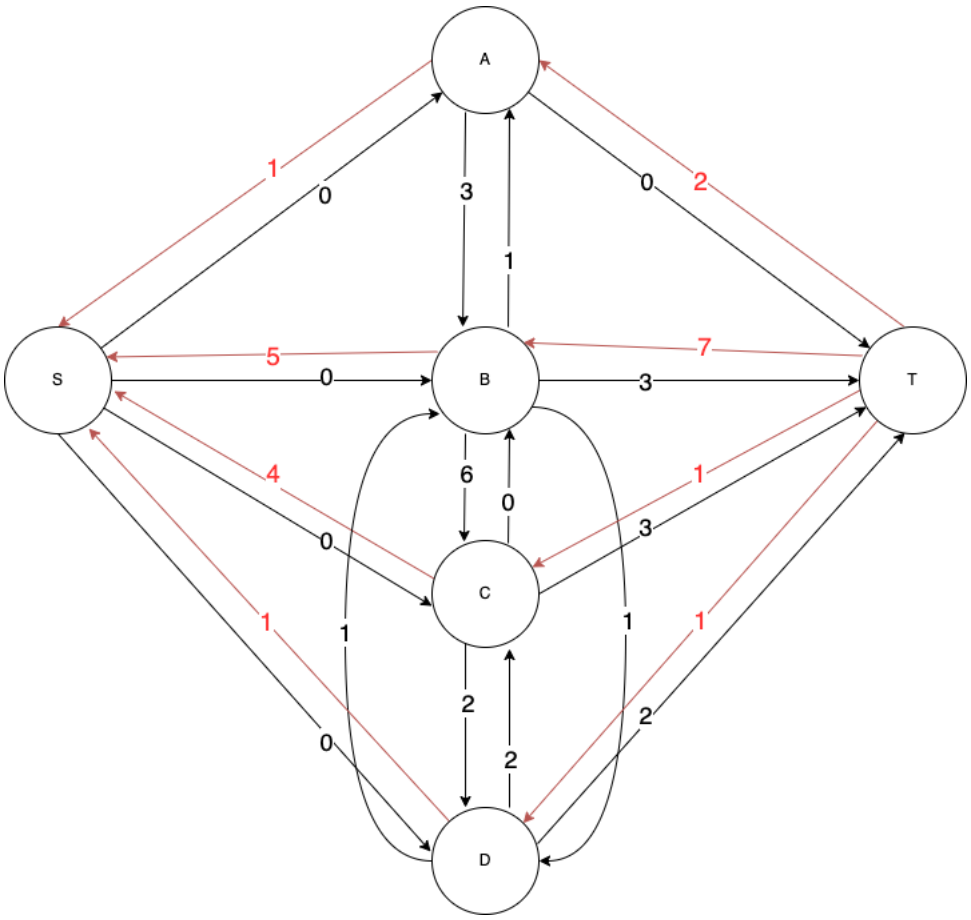


Figura 32: Actualización del grafo residual para P6. Flujo final = 11

Como se puede notar, ya no hay mas caminos de aumento de flujo por lo que se corta la interacción, con un flujo total de 11. Además, se logra el siguiente corte:

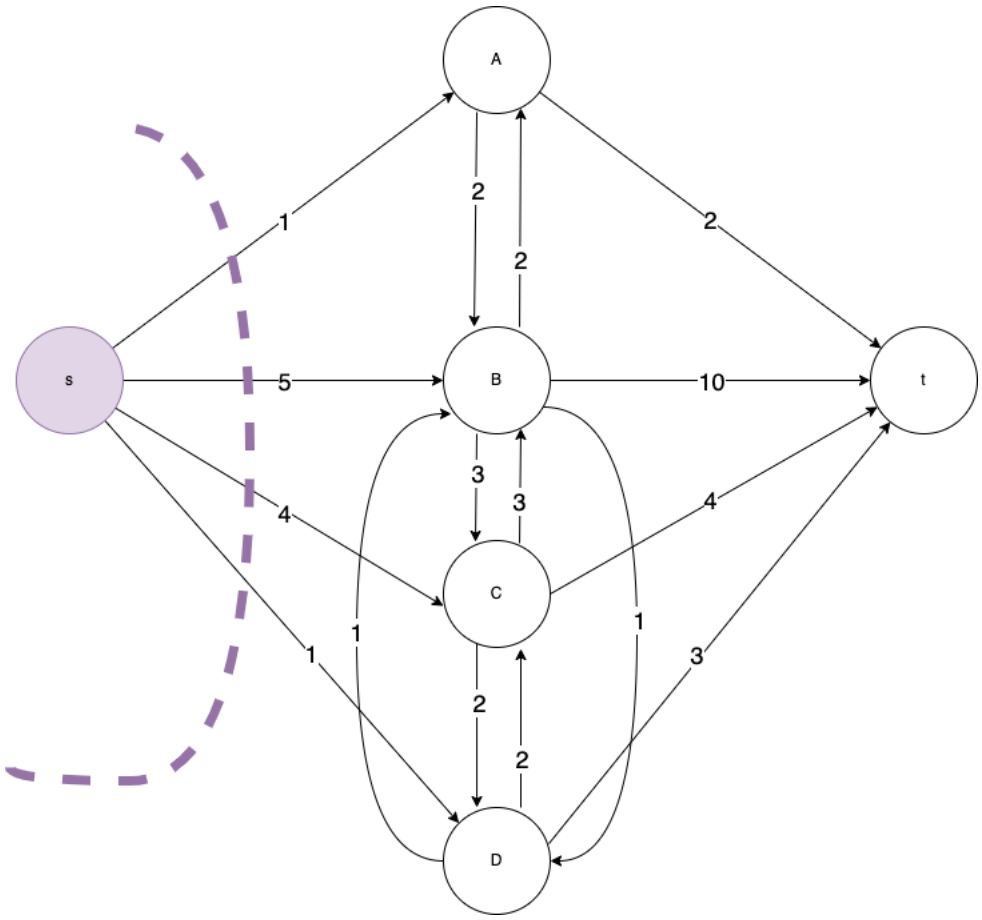


Figura 33: Corte del Flujo final = 11

Como ya no hay mas caminos S-T, se itera sobre el grafo residual buscando nodos alcanzables desde S. Tal como se puede ver, no hay nodos alcanzables desde S por lo que el array tareasEquipo2 esta vacía y tareasEquipo1 = [A, B, C, D] (por ser el complemento). Se devuelve el valor del flujo y los dos vectores, es decir:

```
tareasEquipo1 = [A,B,C,D]
tareasEquipo2 = [ ]
flujoTotal = 11
```

finalizado el algoritmo.

4.5. Programación del algoritmo

El programa se ejecuta de la siguiente manera:

```
python3 ./tp3.py file_path
```

4.6. Complejidad del programa

Complejidad espacial Sin tener en cuenta el costo espacial de generar la instancia del problema en modelo grafo, se consideran las estructuras auxiliares que se emplean para devolver las tareas asignadas a cada equipo. Sea n la cantidad de tareas:

Para el procesamiento del grafo se utilizan dos estructuras auxiliares: una matriz de adyacencia de tamaño [n+2]*[n+2] que tiene un costo aproximado de $O(n^2)$. Por otro lado, se utiliza un diccionario que mapea el nombre de las tareas con un id único, el cual tiene un costo de O(2(n+2)) $\sim O(n)$.

Sea k la cantidad de tareas asignadas al equipo 1, entonces la complejidad espacial de esa lista resulta $O(k)$. Por otro lado, si la cantidad de tareas asignadas al equipo 2 es el complemento, es decir n - k, entonces su complejidad espacial es $O(n-k)$. Por lo tanto, la complejidad espacial es $O(k) + O(n-k) = O(n)$. También se devuelve el flujo que es almacenado en una variable simple, con costo $O(1)$.

Finalmente la complejidad espacial del código es de $O(n^2)$.

Complejidad temporal El algoritmo presentado hace uso de varios anidamientos de ciclos. Podemos separar la complejidad temporal en dos etapas, la construcción y seteo de las estructuras auxiliares para el posterior procesamiento y el procesamiento en sí. En el primer caso, se incurre en un complejidad de O(n) al setear los valores del diccionario y otro ciclo (no anidado) donde se setean los valores de la matriz de adyacencia O(n). Luego, durante el procesamiento en sí, el ciclo **for** mas externo se realiza en la función ***FordFulkerson***, con costo $O(n)$. Dentro de este ciclo, se invoca a la función ***BFS*** que ejecuta un ciclo **while**, de costo $O(n)$ y dentro de este un ciclo **for** también con costo $O(n)$. Las operaciones dentro de este bucle mas interno son operaciones de comparación y asignación de costo $O(1)$. Por lo tanto, hasta acá la complejidad escala hasta $O(n^3)$. Luego, dentro del ciclo **while** mas externo se realizan 2 bucles no anidados con complejidad $O(n)$, con operaciones del orden de $O(1)$ dentro.

Finalmente, cuando no hay ya mas caminos de aumento, se itera sobre la fila 1 del grafo residual, de costo $O(n)$ con operaciones de comparación.

En conclusión, a partir de todo lo desarrollado anteriormente, se puede decir que el algoritmo presenta una complejidad temporal de $O(n^3)$.

4.7. Correcciones 3: Más allá de Bellman-Ford

4.7.1. Correcciones Respuesta 2.2

Enunciamos el problema de decisión como el de encontrar un camino simple de longitud máxima K en un gráfico dado. Un camino se llama simple si no tiene vértices repetidos; la longitud de un camino puede medirse por su número de aristas o (en grafos ponderados) por la suma de los pesos de sus aristas.

Procedemos a probar que el Problema de camino más largo es NP. Llamaremos PCL al Problema de camino más largo. Pensemos en una posible solución al problema.

Sea C(S,I,K) un certificador polinomial del problema que admite como parámetros.

Sea S una posible (a determinar) solución al problema.

Sea I una instancia del problema. Es decir, un grafo G = (V,E)

Sea K un entero positivo igual al K del enunciado del problema de decisión dado.

Algorithm 1 Pseudocódigo del certificador

Verificar que todas las aristas de S existen en el grafo G. Esto lo hacemos en O(E^2). Porque por cada arista en nuestra solución, tenemos que ver en las aristas del grafo G si existe.

for s in S **do**

  if s not in G.E **then** false

También debemos verificar si la suma de los pesos de las aristas de S es menor a K. Si no es pesado, es simplemente verificar que la suma de las aristas sea menor a K. Esto lo hacemos linealmente en O(E) porque a lo sumo el camino tiene todas las aristas.

if Sum(S) >= K **then** true. Donde Sum es una función que discrimina ambos casos detallados anteriormente.

  Sino false

A su vez, el camino debe ser simple, es decir que las aristas no se repitan. Llamamos Dic1 al diccionario de aristas. Llamamos Dic2 al diccionario de vértices.Llamamos (u,v) a una arista. Esto lo hacemos en O(E) porque a lo sumo tendremos que recorrer tantos elementos en S como aristas haya en el grafo G.

for s in S **do**

  Dic1[s] +=1

  if Dic1[s] >1 **then** false

  Dic2[s.u] +=1

  if Dic2[s.u] >1 **then** false

    Dic2[s.v] +=1

  if Dic2[s.v] >1 **then** false

4.7.2. Correcciones Respuesta 2.3

Enunciamos el problema de decisión correspondiente:

"Dado un grafo con peso en sus ejes y un par de nodos al que llamaremos origen y destino. Queremos

saber si existe un camino simple de costo menor a C que los una".

Hasta ahora solamente podemos asegurar que es NP-Hard (detallado en las entregas anteriores) para que sea NP-Completo, debemos probar también que es NP.
Esto lo hacemos encontrando un certificador polinomial que lo certifique.

Ahora procedemos a probar que el problema es NP:
Sea K un entero = C.
Sea C(S,I,K) un certificador del problema "SSPGNC". Sea S un subconjunto de aristas con peso.
Sea I una instancia del problema. Es decir, un grafo G = (V,E)

Algorithm 2 Pseudocodigo del certificador

Sea K un entero = C.

Sea S un subconjunto de aristas con peso.

Sea Acumulador una variable para acumular una suma.

Vemos si en la solución S podemos ir desde s a t. Esto lo hacemos en O(E) porque a lo sumo tenemos que pasar por todas las aristas.

vemos si existen todas las aristas de S en el grafo G.Esto lo hacemos en O(E^2).

for s in S **do**

    if s not in G.E **then** false

    Sino false

Tenemos que verificar que el peso del camino sea menor a K. Esto lo hacemos en O(E) porque a lo sumo tenemos igual cantidad de aristas que las del grafo G.

for cada arista en S **do**

    sumamos el valor de su peso y lo almacenamos en un acumulador Acumulador.

if Acumulador <K **then** true
Sino false

A su vez, el camino debe ser simple, es decir que las aristas no se repitan. Llamamos Dic1 al diccionario de aristas. Llamamos Dic2 al diccionario de vértices.Llamamos (u,v) a una arista. Esto lo hacemos en O(E) porque a lo sumo tendremos que recorrer tantos elementos en S como aristas haya en el grafo G.

for s in S **do**

    Dic1[s] +=1

    if Dic1[s] >1 **then** false

    Dic2[s.u] +=1

    if Dic2[s.u] >1 **then** false

        Dic2[s.v] +=1

    if Dic2[s.v] >1 **then** false

Con esto se prueba que el problema es NP.