FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 3

[75.29 / 95.06] Teoría de Algoritmos I
Segundo cuatrimestre de 2021
Fecha de entrega 2 de Noviembre de 2021, 19:00
Primera Entrega

Alumno	Padrón	Email
Fernando Martin Tejelo	98911	ftejelo@fi.uba.ar
Mariano Lazzarini	106352	mlazzarini@fi.uba.ar
Tobias Luciano Rivero Trujillo	106302	trivero@fi.uba.ar
Martin Pata Fraile de Manterola	106226	mpata@fi.uba.ar
Matias Gabriel Fusco	105683	mfusco@fi.uba.ar

Índice

1. Parte 1: Una campaña publicitaria masiva pero mínima

Una empresa de turismo que vende excursiones desea realizar una campaña publicitaria en diferentes vuelos comerciales con el objetivo de llegar a todos los viajeros que parten del país A y que se dirigen al país B. Estos viajeros utilizan diferentes rutas (algunos vuelos directos, otros armando sus propios recorridos intermedios). Se conoce para una semana determinada todos los vuelos entre los diferentes aeropuertos con sus diferentes capacidades. Además parten del supuesto que durante ese periodo la afluencia entre A y B no se verá disminuida por viajes entre otros destinos.

Desean determinar en qué trayectos simples (trayecto de un viaje que inicia desde un aeropuerto y termina en otro) poner publicidad de forma de alcanzar a TODAS las personas que tienen el destino inicial A y el destino final B. Pero además desean que siempre que sea posible seleccionen la combinación que tenga el menor número de vuelos comerciales. Esto es porque pagan tanto por cantidad de vuelos como por pasajeros que cumplan con la condición de ser del país de origen A y con destino final B.

Se pide:

1. Proponer una solución algorítmica que resuelva el problema de forma eficiente. Explicarla paso a paso. Utilice diagramas para representarla.
2. Plantear la solución como si fuese una reducción de problema. ¿Puede afirmar que corresponde a una reducción polinomial? Justificar.
3. ¿Podría asegurar que su solución es óptima?
4. Programe la solución.
5. Compare la complejidad temporal y espacial de su solución programada con la teórica. ¿Es la misma o difiere?

1.1. Proponer una solución algorítmica que resuelva el problema de forma eficiente. Explicarla paso a paso. Utilice diagramas para representarla

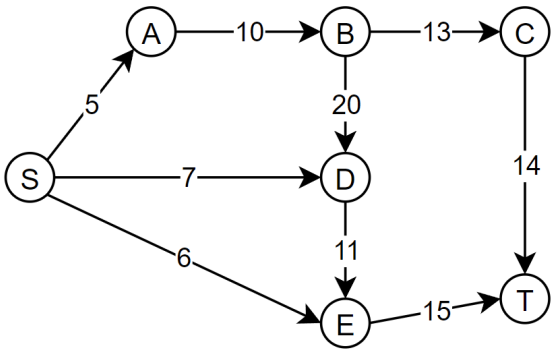
1.1.1. Calcular cantidad de pasajeros máximos deade A hasta B

1.1.1.1 Convertir las ciudades y trayectos en una red de flujo

Para resolverlo, primero convertimos los datos de entrada en un grafo. Tomando cada ciudad como un nodo de este grafo, y cada viaje entre dos ciudades, como una arista que tiene como peso la cantidad de pasajeros que puede llevar el avión de una a otra ciudad.

Convertimos la ciudad .ª"(la ciudad origen) en un nodo Fuente, y la ciudad "B"(la ciudad destino) en un nodo Sumidero.

En nuestro ejemplo usaremos directamente el nombre "S"para la ciudad fuente; y "T"para la ciudad sumidero.



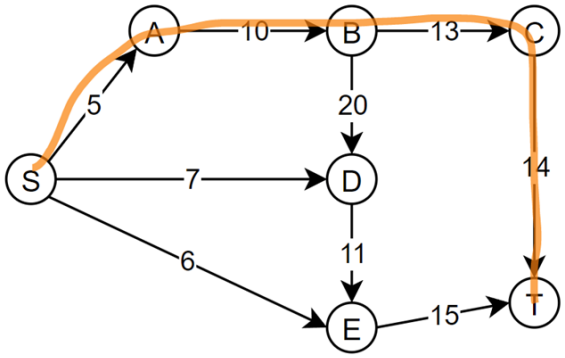
1.1.1.2 Usar Ford-Fulkerson para determinar el flujo máximo

A partir del paso anterior, tenemos generada una red de flujo, podemos calcular el flujo máximo, que sería equivalente a la cantidad de pasajeros máximos que pueden ir desde A hasta B. Como no queremos perder el grafo original, hacemos una copia del grafo original, para no destruirlo, y con esta copia es sobre la que va a operar Ford-Fulkerson

1.1.1.3 Ejecutar Ford-Fulkerson

Primero buscamos un camino que nos lleve del nodo fuente (S) hasta el nodo sumidero (T). Para esto usamos una version modificada de DFS, en la que vamos avanzando desde nodos conocidos, hacia nodos desconocidos, hasta llegar al nodo objetivo (T); o hasta no poder avanzar más. En el caso de no poder avanzar más, volvemos un nodo para atras en el camino, y seguimos buscando nodos no visitados.

Una vez tenemos un camino encontrado, como el siguiente:

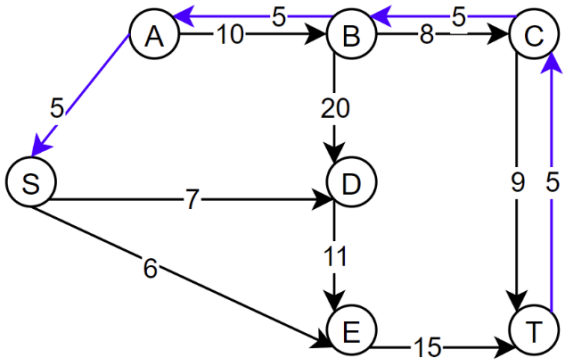


Camino: S-A-B-C-T

Bottleneck: 5 (S-A)

Flujo maximo: 0

Buscamos cual es la arista con menor peso en el camino, este será nuestro bottleneck (en este caso 5). Lo que queremos hacer es aumentar ese bottleneck, por lo que al peso de todas las aristas del camino, le restamos el bottleneck; y agregamos una arista con sentido contrario a la del camino, y le asignamos el peso del bottleneck.

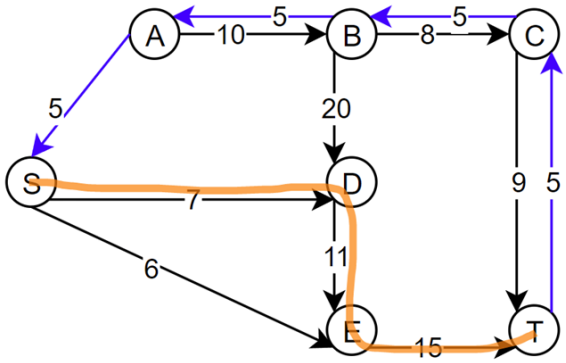


Camino: S-A-B-C-T

Bottleneck: 5 (S-A)

Flujo maximo: 0+5

Una vez terminado de modificar los pesos para aumentar el bottleneck del camino, sumamos este bottleneck a un contador general donde almacenamos el flujo máximo que estamos pudiendo transportar. Luego de esto volvemos al principio: buscamos de nuevo un camino desde S hasta T:

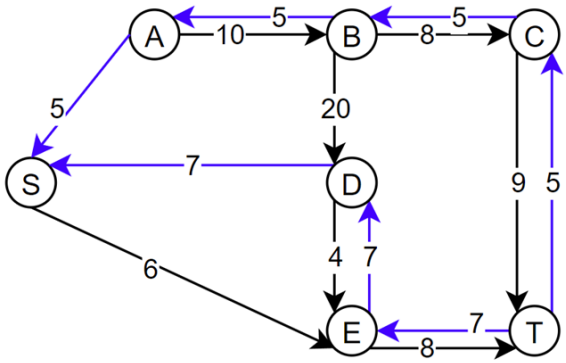


Camino: S-D-E-T

Bottleneck: 7 (S-D)

Flujo maximo: 5

Buscamos de nuevo el menor peso del camino (7), y aumentamos este bottleneck, restando este peso de las aristas; y agregandolo aristas de sentido contrario con ese peso. Sumamos este bottleneck al contador de flujo maximo.

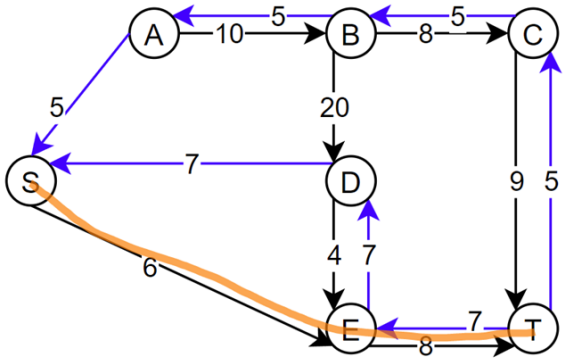


Camino: S-D-E-T

Bottleneck: 7 (S-D)

Flujo maximo: 5+7

Volvemos a buscar un nuevo camino; y buscamos el peso minimo (6):

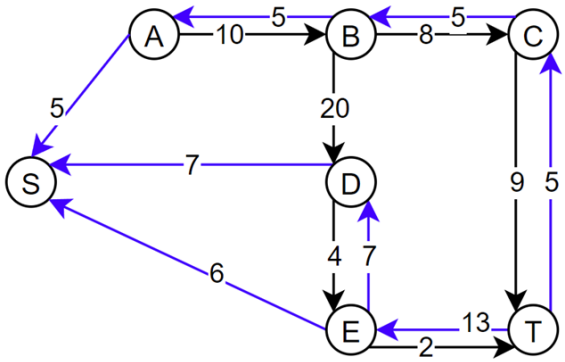


Camino: S-E-T

Bottleneck: 6 (S-E)

Flujo maximo: 12

Aumentamos el bottleneck, en este caso hay una arista con sentido contrario ya creada, por lo que en vez de crear la arista y asignarle el peso del bottleneck, a esta arista ya existente, le sumamos al peso el bottleneck. Luego sumamos el bottleneck al flujo maximo.

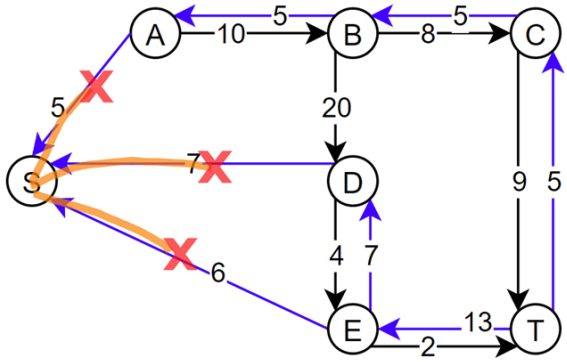


Camino: S-E-T

Bottleneck: 6 (S-E)

Flujo maximo: 12+6

En este momento, intentamos buscar un nuevo camino desde S hasta T; pero no existe ninguno, por lo que sabemos que llegamos al flujo maximo que fuimos almacenando.



Camino: -

Bottleneck: -

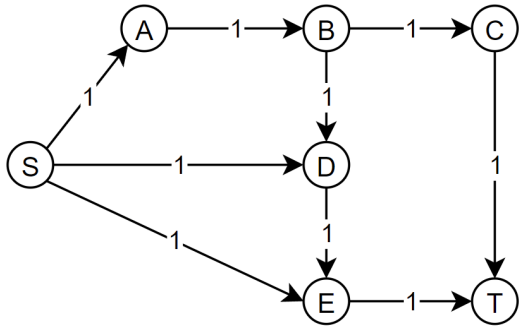
Flujo maximo: 18

Tambien podemos ver el corte minimo del grafo, pero que en este caso no nos tiene utilidad.

1.1.2. Calcular los viajes donde poner publicidades

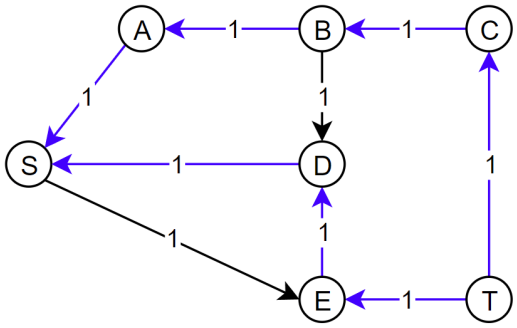
1.1.2.1 Convertir las ciudades y trayectos en una red de flujo

Para esta segunda parte del problema, convertimos las ciudades en nodos, y los viajes en aristasde peso 1, y al igual que antres, converiremos a la ciudad A en nodo fuente, y B en nodo sumidero.

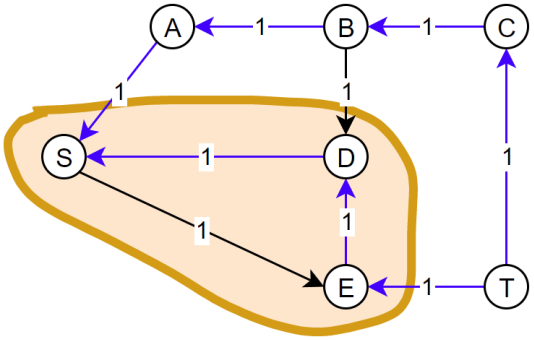


1.1.2.2 Usar Ford-Fulkerson para determinar el corte minimo

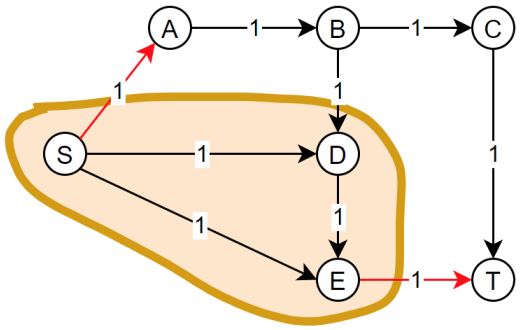
Al grafo obtenido del paso anterior, le ejecutamos el algoritmo de Ford Fulkerson, y quedaría un grafo residual como el siguiente:



A partir de este grafo residual, podemos determinar el corte minimo que divide el grafo a la "mitad".
Para hacer eso, primero almacenamos en un set todos los nodos que se pueden alcanzar desde el nodo fuente (S). Esto se puede hacer con un DFS.



Y a partir de este set de nodos que se pueden alcanzar desde S, podemos, en el grafo de peso 1 original, buscar todas las aristas que salen desde un nodo del set, y terminan en un nodo fuera del set.



Estas aristas, serían los viajes minimos donde habría que poner la publicidad, para alcanzar a todas las personas que viajen de la ciudad inicial a la final.

Como los lados tienen peso 1, el corte minimo, además de tener el flujo minimo que pase del grafo con el nodo Fuente al grafo con el nodo sumidero; es el corte que corta la cantidad mínima de aristas.Por lo que las publicidades hay que ponerlas en estos viajes que van del grafo con la Fuenta,al grafo con el Sumidero.

1.1.3. Pseudocódigo

A continuación se adjunta el Pseudocódigo utilizado:

1.1.3.1 Ford-Fulkerson

```
1  ford_fulkerson(grafo, fuente, sumidero):
2    grafo_residual = copia(grafo)
3
4    capacidad_maxima = 0
5
6    mientras exista un camino desde fuente hasta sumidero:
7      cuello_de_botella = nodo con menor peso en camino
8      capacidad_maxima = capacidad_maxima + cuello_de_botella
9
10     para cada arista del camino:
11       augment(grafo, arista, cuello_de_botella)
12
13
14   devolver capacidad maxima y grafo residual
15   (esta ultimo ser a solo si queres encontrar el corte minimo,
16   o hacerle alguna otra operacion)
17
18
19 augment(grafo, arista, cuello_de_botella):
20   si cuello_de_botella >= peso_arista:
21     eliminar arista
22
23   sino:
24     peso_arista = peso_arista - cuello_de_botella
```

1.1.3.2 Buscar aristas que cruzan el corte mínimo

```
1  calcular_aristas_corte_minimo(grafo, grafo_residual, nodo_fuente)
2    set_vertices_fuente = vertices alcanzables desde
3                          nodo_fuente en grafo_residual
4
5    aristas_corte_minimo = []
6
7    por cada nodo en set_vertices_fuente:
8      para cada vecino de nodo:
9        si vecino no esta en set_vertices_fuente:
10         agregar (nodo, vecino) a aristas_corte_minimo
11
12   devolver aristas_corte_minimo
```

1.1.3.3 Solucion general al problema propuesto

```
algoritmo_solucion_general(grafo_viajes, ciudad_a, ciudad_b):
  grafo_peso_1 = copia de grafo_viajes pero con aristas de peso 1

  capacidad max pasajeros = ford_fulkerson(grafo_viajes, ciudad_a, ciudad_b)
  (solo almacenamos el flujo maximo)

  grafo_residual_peso_1 = ford_fulkerson(grafo_peso_1, ciudad_a, ciudad_b)
  (solo almacenamos el grafo residual)

  conexiones publicitarias =
  calcular_aristas_corte_minimo(grafo_peso_1, grafo_residual_peso_1, ciudad_a)

  devolver capacidad maxima pasajeros, conexiones publicitarias
```

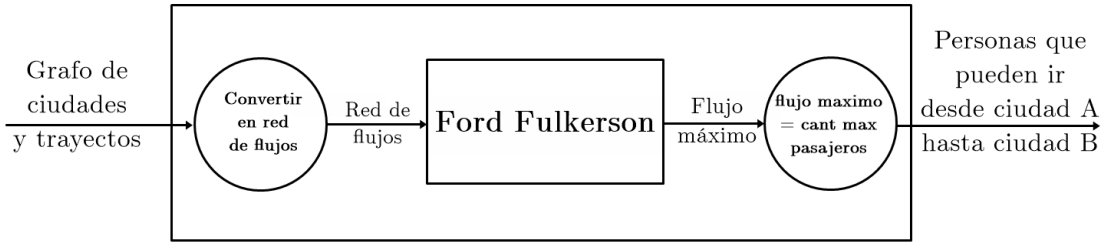
1.2. Plantear la solución como si fuese una reducción de problema. ¿Puede afirmar que corresponde a una reducción polinomial?

La solucion propuesta; resulta una reduccion de 2 problemas; el problema de calcular la cantidad máxima de personas que van desde la ciudad A hasta la ciudad B, y el problema de calcular cuales son los mejores vuelos donde poner publicidad.

Ambos casos se pueden resolver reduciendo el problema, a un problema de flujo maximo.

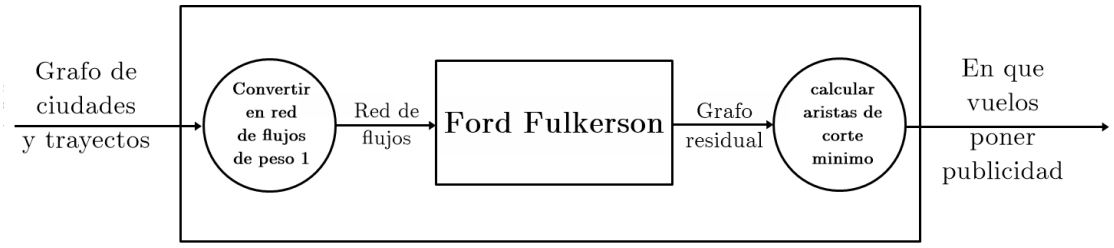
1.2.1. Cantidad máxima de pasajeros

En el caso de calcular la cantidad máxima de pasajeros, resulta más sencillo, ya que solo se necesita hacer una transformacion de las ciudades y vuelos, a un grafo de flujos. Y con Ford-Fulkerson podemos calcular el flujo maximo, que se reinterpreta como la cantidad máxima de personas que pueden ir desde la ciudad inicial a la ciudad final.



1.2.2. Vuelos donde poner publicidades

Para calcular donde poner las publicidades, se tiene que transformar la lista de vuelos entre ciudades, a una red de flujo, pero con pesos con valor 1. Una vez ejecutado Ford-Fulkerson, y obtenido el grafo residual, podemos calcular las aristas que cruzan el corte minimo, para saber los viajes minimos donde poner publicidad, y que igualmente alcanzen a todos los pasajeros que viajen desde ciudad A hasta ciudad B.



1.3. ¿Podría asegurar que su solución es óptima?

En cuanto al calculo de la máxima cantidad de pasajeros que pueden ir desde A hasta B, podemos asegurar que nuestro algoritmo es óptimo, pues hacemos uso de Ford-Fulkerson, y esta comprobado que este devuelve siempre la solución óptima en cuanto al flujo maximo (tenemos una demostracion disponible en un video del campus <https://www.youtube.com/watch?v=wnbhg0QJAWc> a partir del minuto 25).
Para la parte de encontrar en que vuelos conviene poner las publicidades, también utilizamos propiedades de Ford-Fulkerson. Sabemos que luego de que este algoritmo finalice, nos queda en el grafo residual el corte mínimo del grafo. Vamos a tener una componente conexa que consiste de todos los nodos alcanzables desde 's' en el grafo residual, y sabremos que todas las aristas (en el grafo original) que empiecen en un nodo que sea parte de esta componente conexa, pero terminen en uno que no lo es, formaran parte del corte mínimo.
Como el corte mínimo es el peso mínimo de aristas que necesitamos sacar para que no se pueda llegar de la fuente al sumidero, la suma de todos estos pesos en este caso va a ser igual a la cantidad de aristas que tendríamos que sacar, pues hicimos que todas las capacidades sean 1. Entonces, vamos a saber exactamente cuales son las aristas sobre las que tendríamos que poner las publicidades, manteniendo la cantidad al mínimo posible. De esta manera, aseguramos una solución óptima.

1.5. Compare la complejidad

1.5.1. Calculo de complejidades del codigo

Primero calculamos la complejidad de nuestro código, para poder compararla con la complejidad teórica

1.5.1.1 Complejidad Ford Fulkerson

```
def flujo_ford_fulkerson(grafo, s, t):
    capacidad_maxima = 0
    grafo_residual = grafo.copy()                                                    O(V+E)

    camino = obtener_camino(grafo_residual,s,t)                                      O(V+E)
    while camino:
        capacidad_residual_camino = min_peso(grafo_residual, camino) #peso minimo de camino (cuello de botella)   O(V)
        capacidad_maxima += capacidad_residual_camino                                                              O(1)

        nodo_anterior = None                                                                                       O(1)

        for nodo_actual in camino:
            if (nodo_anterior != None): #Si nodo anterior es None, no se ejecuta la actualizacion del grafo, para poder te
            y anterior.
                actualizar_grafo_residual(grafo_residual, nodo_anterior, nodo_actual, capacidad_residual_camino)   O(1)

            nodo_anterior = nodo_actual                                                                            O(1)


        camino = obtener_camino(grafo_residual,s,t)                                  O(V+E)

    return capacidad_maxima, grafo_residual
```

O(V) — O(C*(V+E))

Nota: La variable C en el calculo de la complejidad corresponde a la suma de las capacidades de las aristas salientes del nodo 's', que resulta de calcular el peor de los casos en cuanto a la cantidad de caminos posibles desde 's' a 't' que se pueden encontrar a medida que avanza el algoritmo. En el peor caso, cada una de esas aristas va a disminuir en 1 en cada iteración, y como dejaría de haber un camino disponible cuando su capacidad sea cero, podemos argumentar que en el caso menos conveniente la cantidad de veces que podemos encontrar un camino es la suma de estas capacidades.

- **Complejidad Temporal:** O(C*(V+E))

- **Complejidad Espacial:** O(V+E); por hacer una copia del grafo. Si no se hiciera copia del grafo (modificando el grafo que nos proveen): O(V)

1.5.1.2 Complejidad obtener camino

```
def obtener_camino(grafo, s, t):
    camino = dfs(grafo, s, t)
    return camino[::-1]
```

O(V+E)
O(V)

- **Complejidad Temporal:** O(V+E)
- **Complejidad Espacial:** O(V)

1.5.1.3 Complejidad DFS

```
#Recibe un grafo, el nombre del nodo inicial (nodo_actual), y el nodo objetivo (al que se quiere encontrar el camino). Devuelve una lista con los nod
#Con el objetivo en la posicion 0; y el nodo inicial en la posicion N
def dfs(grafo, nodo_actual, objetivo, visitados = None):
    if visitados == None: #Para que la primera llamada recursiva no tenga que enviar el set
        visitados = set()

    if nodo_actual in visitados:    #Devuelve nada (lista vacia) si se ya paso por ese nodo
        return []

    if nodo_actual == objetivo:     #Llegó al objetivo, empieza a armar la lista.
        visitados.add(nodo_actual)
        return [nodo_actual]

    visitados.add(nodo_actual)  #Agrega el nodo actual al set de nodos visitados
    for vecino in grafo.adyacentes(nodo_actual):    #Se busca sigue el DFS en cada uno de los vecinos del nodo actual
        resultado = dfs(grafo, vecino, objetivo,  visitados)

        if resultado:
            resultado.append(nodo_actual)
            return resultado

    return []       #En el caso de que no se haya logrado alcanzar el nodo objetivo, se devuelve una lista vacia
```

Este algoritmo es de complejidad O(V+E), pues un recorrido DFS que corta cuando se llega al nodo objetivo. En el peor de los casos, puede ser que tengamos que recorrer el grafo entero para llegar a nuestro objetivo, y devuelve el camino que se recorrió. Vamos a recorrer todos los vértices 1 sola vez (si están en el set de visitados los ignoramos), y para cada nodo miramos todos sus nodos adyacentes, lo que es equivalente a todas sus aristas salientes. De esta manera, como solo se realizan operaciones constantes además del for para recorrer el grafo, podemos decir que se recorren todas los nodos y todas las aristas 1 sola vez, por lo que la complejidad resulta O(V+E).

- **Complejidad Temporal:** O(V+E)
- **Complejidad Espacial:** O(V)

1.5.1.4 Complejidad obtener camino

```
def obtener_camino(grafo, s, t):
    camino = dfs(grafo, s, t)
    return camino[::-1]
```

O(V+E)
O(V)

- **Complejidad Temporal:** O(V+E)
- **Complejidad Espacial:** O(V)

1.5.1.5 Complejidad actualizar grafo residual

```
def actualizar_grafo_residual(grafo_residual, u, v, valor):
    peso_anterior = grafo_residual.peso(u,v)                    O(1)
    peso_anterior_residual = grafo_residual.peso(v,u)           O(1)

    if peso_anterior <= valor:      #si el nuevo peso es menor a 0
        grafo_residual.borrar_arista(u,v)                        O(1)
    else:                                                              O(1)
        grafo_residual.cambiar_peso(u,v,peso_anterior - valor)  O(1)

    if not grafo_residual.estan_unidos(v,u):
        grafo_residual.agregar_arista(v,u, valor)               O(1)

    else:                                                                        O(1)
        grafo_residual.cambiar_peso(v, u, peso_anterior_residual + valor)O(1)
```

- **Complejidad Temporal:** O(1)
- **Complejidad Espacial:** O(1)

1.5.1.6 Complejidad min peso

```
def min_peso(grafo, camino):
    min_peso = float('inf')  O(1)

    for i in range(len(camino)-1):
      peso = grafo.peso_arista(camino[i],camino[i+1]) O(1)
      if peso < min_peso:                                     O(V)
        min_peso = peso   O(1)

    return min_peso
```

- **Complejidad Temporal:** O(V)
- **Complejidad Espacial:** O(1)

1.5.1.7 Complejidad calcular Aristas corte minimo

```
def calcular_aristas_corte_minimo(grafo, grafo_residual, s):
    set_nodos_lado_fuente = grafo_residual.set_unilateralmente_conexo_desde(s)        O(V+E)
    #Set que contiene a los nodos que se pueden acceder luego de augmentar el grafo

    conexiones = []                                                                   O(1)

    for nodo in set_nodos_lado_fuente:
        if nodo not in grafo:
            raise IndexError("Nodo perteneciente al set no existe en el grafo al que el set re

        for nodo_vecino in grafo.adyacentes(nodo):
            if nodo_vecino not in set_nodos_lado_fuente:                              O(1)  O(E)
                conexiones.append([nodo, nodo_vecino])

    return conexiones
```

- **Complejidad Temporal:** O(V+E)
- **Complejidad Espacial:** O(V)

1.5.1.8 Complejidad Solución

```
grafo_peso_uno = grafo.copy_con_pesos(1)    #Crea un grafo con las mismas ciudades y con las aristas en las r

capacidad_maxima_pasajeros , _ = ff.flujo_ford_fulkerson(grafo, nodo_s, nodo_t)      O(C*(V+E))

_,  grafo_residual_peso_1 = ff.flujo_ford_fulkerson(grafo_peso_uno, nodo_s, nodo_t)  O(C*(V+E))

print("La mayor cantidad de pasajeros que pueden ir desde la ciudad", nodo_s, "hasta la ciudad", nodo_t, "sor
print()

conexiones_publicidades = ff.calcular_aristas_corte_minimo(grafo, grafo_residual_peso_1, nodo_s) O(VE)
```

- **Complejidad Temporal:** O(VE) + O(C*(V+E)) (depende de la proporción en la diferencia de tamaño entre las variables involucradas)
- **Complejidad Espacial:** O(V+E)

1.5.2. Comparando con la solución teórica

En cuanto a complejidad temporal:
Podemos ver que para el Ford-Fulkerson planteado en el pseudocódigo, la complejidad es la mismo, puesto que este primero, para cada vez que exista un camino desde la fuente hasta el sumidero, hace falta modificar las aristas de los nodos en el camino (O(V-1)) y luego buscar un camino (O(V+E)).
En ”calcular-aristas-corte-mínimo” ocurre algo similar. En el pseudocódigo, vemos que lo mas costoso es calcular el ”lado” del grafo residual que es alcanzable desde ’s’, que es O(V+E). Es lo mas costoso en el algoritmo ya que el paso siguiente es recorrer los nodos que son alcanzables, que es O(V). Entonces, también coincide con la solución en python.
Por ultimo, la ”solución general” que es la totalidad de la solución puesta en funcionamiento, es idéntica a la que programamos, y como para todas las funciones utilizadas dentro de ella la complejidad temporal coincide entre pseudocódigo y código, tienen la misma complejidad temporal. Entonces, vemos que la complejidad temporal de nuestra solución teorica coincide con la de la que

hicimos en python.

En cuanto a complejidad espacial:
Como vimos anteriormente, las soluciones teóricas y las programadas hacen exactamente lo mismo, por lo que la complejidad espacial termina siendo la misma. Ford-Fulkerson es O(V+E) por la copia del grafo. Calcular las aristas alcanzables desde 's' es O(V), igual que el proceso para obtener caminos (dfs). En la solución general, cuando se calcula las complejidades totales, la mas grande es la de Ford-Fulkerson a la hora de copiar el grafo, por lo que la complejidad espacial de la solución termina siendo O(V+E).

2. Parte 2: Equipos de socorro

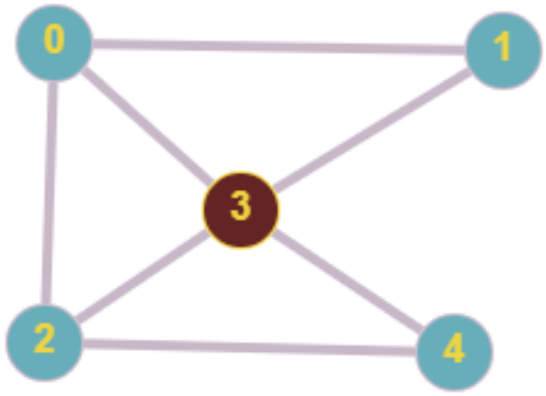
Parte 2: Equipos de socorro El sistema ferroviario de un país cubre un gran conjunto de su territorio. El mismo permite realizar diferentes viajes con transbordos entre distintos ramales y subramales que pasan por sus principales ciudades. Dentro de su proceso de mejoramiento del servicio buscan que ante una emergencia en una estación se pueda llegar de forma veloz y eficiente. Consideran que eso se lograría si el equipo de socorro se encuentra en esa misma estación o en el peor de los casos en una estación vecina (que tenga una trayecto directo que no requiere pasar por otras estaciones). Como los recursos son escasos desean establecer la menor cantidad de equipos posibles (un máximo de k equipos pueden solventar). Se solicita nuestra colaboración para dar con una respuesta a este problema.

Se pide:

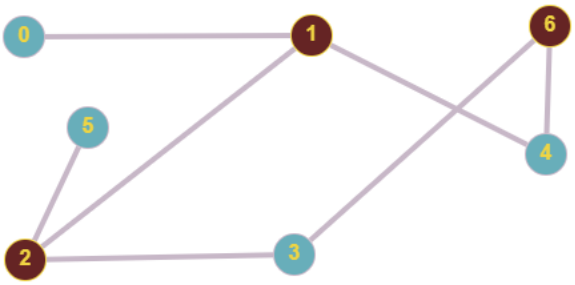
1. Utilizar el problema conocido como "set dominante" para demostrar que corresponde a un problema NP-Completo.
2. Asimismo demostrar que el problema set dominante corresponde a un problema NP-Completo.
3. Con lo que demostró responda: ¿Es posible resolver de forma eficiente (de forma "tratable") el problema planteado?

2.1. Breve explicación: set dominante

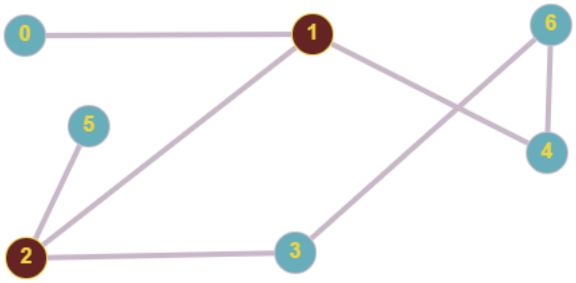
Antes de desarrollar la solución explicaremos brevemente el problema del set dominante. El problema consiste en dado un grafo G(v,e) el conjunto dominante sera un subconjunto de v, v', para el cual cada elemento que no pertenezca al conjunto estará unido como mínimo a un elemento del mismo.
Por ejemplo:



Llamaremos v' al subconjunto de vértices rojo y v al complemento de v'. En este primer ejemplo vemos un set dominante cuyo cardinal es igual a 1, ya que para todo elemento en v podemos encontrar una unión a un elemento de v'. Se presenta 2 ejemplos mas a continuación.



Nuevamente podemos observar que v' es un conjunto dominante ya que para cada elemento en v, existe al menos una unión con un elemento de v'. Ahora observemos que pasa si retiramos uno de elementos en v'.



En este caso al retirar el vértice 6 del conjunto v' encontramos que este set deja de ser dominante

ya que el elemento 6 mismo no tiene unión a ningún elemento de v'. Con el concepto de set dominante ya en claro pasamos a desarrollar la problemática planteada.

2.2. Demostración que el problema es NP

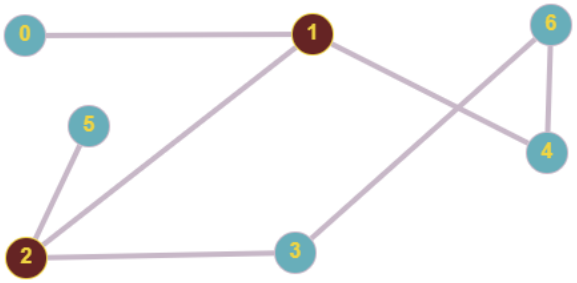
Para demostrar que la problemática que se nos presenta es NP.
Partiremos con lo siguiente:

- Tendremos una instancia del problema, un grafo G(v,e).(tomamos como ejemplo la imagen 2)
- k sera el numero máximo de equipos que puedan solventar.
- Tenemos un certificado t, el cual sera un subconjunto de nodos del grafo G. En este caso->[1,2,6] todo el complemento de v'.
- Reduciremos el problema pedido a un problema de "set dominante".

Empezaremos con la demostración, en primer lugar demostraremos que tanto el Problema de equipo de socorro como el Problema de set dominante son efectivamente NP.
Para esto haremos uso de certificados que en tiempo polinomial nos puedan contestar si una instancia del problema es una solución o no del problema.
En el caso de Set Dominante, que a partir de ahora llamaremos SD, tomaremos la instancia anteriormente mencionada. Podemos ver que polinomial mente podemos decidir si el subconjunto dado es o no solución del problema de la siguiente manera:

1. Recorremos cada vértice del subconjunto
2. Para cada uno de esos vértices verifico en otro set si están o no(sin incluir a los del subconjunto que estoy recorriendo obviamente), si no están lo agrego sino, no hago nada.
3. Cuando termino con de recorrer todo mi subconjunto, si la el tamaño del set mas el tamaño de nuestro subconjunto tiene el mismo tamaño que nuestra instancia del grafo, entonces la respuesta es si de lo contrario no.

Con la instancia elegida, hacemos un seguimiento:



Tenemos nuestra posible solución->[1,2,6] y nuestro conjunto auxiliar en principio vació->[].

1. Primero empezamos en 1, sus adyacentes son: 0,2,4 entonces 0 no pertenece al conjunto solución y tampoco al conjunto auxiliar entonces agregamos al conjunto auxiliar y proseguimos. El vértice 2 pertenece al conjunto solución, por lo tanto no hacemos nada, el 4 no pertenece al conjunto solución y tampoco al conjunto auxiliar entonces lo agregamos al mismo. Terminamos con el vértice 1 entonces pasamos al siguiente.

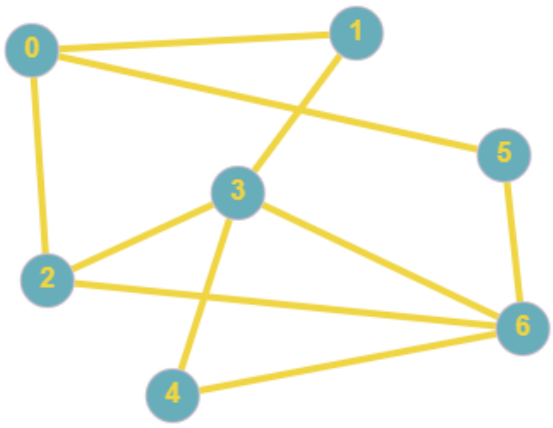
2. El vértice 2 tiene como adyacentes a los vértices: 1,5,3. El vértice 1 esta en el set solución por lo tanto no hacemos nada, el vértice 5 por otro lado, no esta en ninguno de los 2 sets, por lo tanto lo agrego al auxiliar. Finalmente el elemento 3 tampoco esta en ninguno de los sets por eso lo agrego, como no quedan mas adyacentes pasamos al siguiente vértice.

3. como el elemento 6 no tiene adyacentes que no hayan sido ya visitados, no haremos nada.

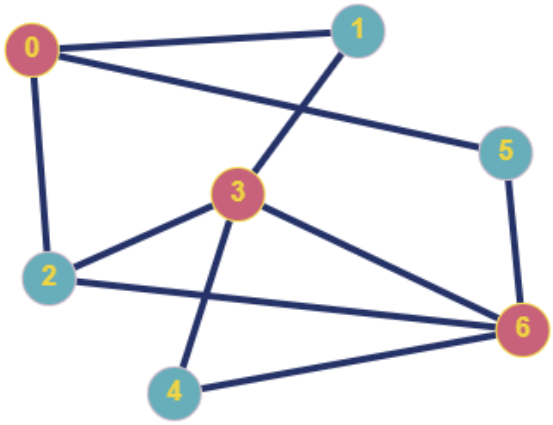
Como todas las comprobaciones de si un elemento esta o no en un set son O(1) entonces no afectara la complejidad total y la certificación seguirá siendo polinomial (V*E) (V Cantidad de vértices y E cantidad de aristas).Con esto demostramos que podemos en tiempo polinomial decidir, dada una instancia del problema y una solución si dicha solución es valida o no, por lo tanto demostramos que SD es NP.

2.3. Set Dominante es NP-Completo

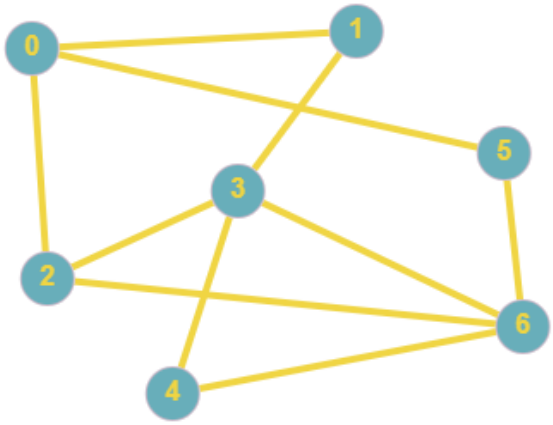
Ya pudimos probar que dada una instancia G de un grafo cualquiera y un certificado t, podemos decidir en tiempo polinomial si existe un set dominante de tamaño K y por lo tanto Set Dominante, a parte de ahora SD, es NP.
Para poder llegar a demostrar que SD, es NP-Completo, trataremos de reducir el problema Vertex Cover, el cual ya sabemos que es NP-Completo a SD. Primero explicaremos brevemente Vertex-Cover(VC).
El Problema Vertex-Cover, o problema de cobertura, tiene como objetivo decidir si, dado un grafo G, es posible tener un subconjunto de V vértices para el cual, cada arista tenga acceso a al menos un vértice de este subconjunto.
Por ejemplo dado el siguiente grafo. Es posible hacer una cobertura de k=3?



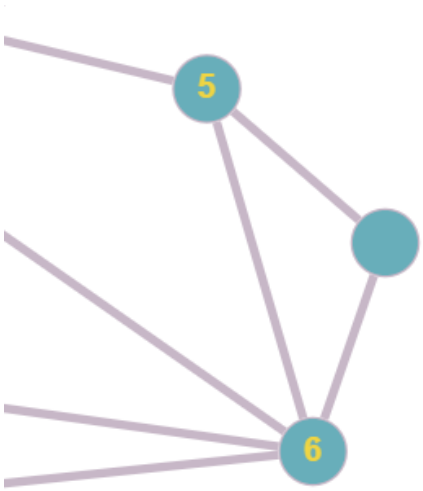
Si observamos bien, si es posible ya que si tomamos un subconjunto con los vértices: 0,3 y 6, vemos que todo arista tiene conexión a al menos un vértice del subconjunto anterior.



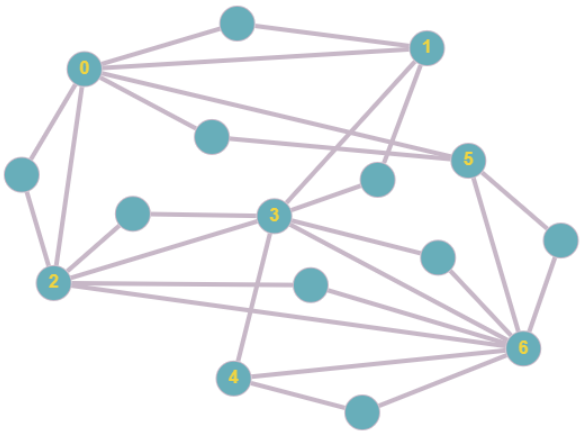
Una vez explicado procedemos a reducir polinomial mente VC a SD: si logramos esto podremos probar que SD es NP-Completo. Partimos del siguiente grafo G.



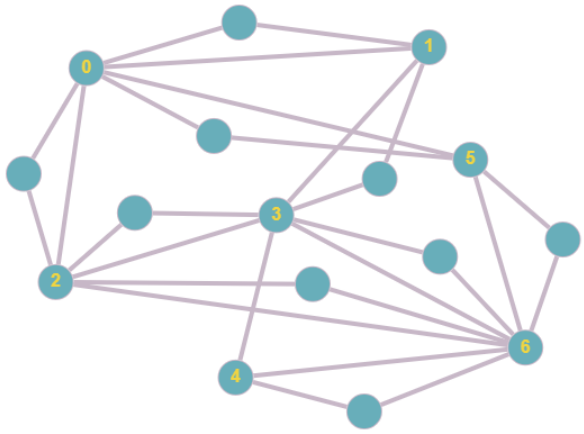
Ahora tenemos que poder transformar esta instancia de problema VC, en una de SD en tiempo polinomial, para que nuestra caja negra, que resuelve SD actué. La transformación procederá de la siguiente manera.
por cada arista que conecte 2 vértices: Agregaremos un vértice compartido por esos 2 vértices:



Estos nuevos vértices sera parte de mi instancia de SD, lo que nos dará como resultado el siguiente grafo G'.



El cual tendrá por cada arista en el grafo original, un vértice que conectara ambos vértices que conectaba la arista original.
Ahora si prestamos atención cada cada uno de los vértices agregados, domina a los 2 vértices que conecta, ambos del grafo original y a si mismo, de igual manera se puede pensar con los otros 2 vértices, del grafo original(dominara a si mismo, y a otros 2 vértices) con la particularidad de que podría dominar inclusive a mas vértices (ya que puede estar unido a otros vértices).Con esto podemos asegurar que siempre habrá un set dominante mínimo que este compuesto solo de los vértices originales, ya que cualquier vértices nuevo, puede ser reemplazado por uno de los originales(por la estructura misma que le dimos).
Ahora bien podemos observar lo siguiente, si tenemos un vértice que domina un vértice nuevo, por la construcción anteriormente explicada también cubrirá la arista del grafo original. Entonces si tenemos un set dominante (de solo los vértices originales) que domine todo los vértices nuevos, estará a su vez, cubriendo todas las aristas del grafo original. Observemos si elegimos, como parte de nuestro set dominante el 0,3 y 6, podremos ver que no solo dominamos a todos los vértices nuevos, sino que, al mismo tiempo tenemos cubiertas todas las aristas originales:



También estamos cubriendo muchas otras aristas, pero si no nos importa ya que son parte del nuevo grafo. Encontramos un set dominante que no incluye ninguno de los vértices nuevos y que es solución de VC en el grafo original.Por lo tanto como pudimos reducir un problema de VC a uno de SD, en tiempo polinomial (agregar los V vértices y E aristas nuevos, con una implementación rápida de grafo, tiene en total, coste polinomial) permite demostrar que como VC es NP-C, es decir es NP y dentro de NP, es de los problemas mas difíciles de NP al reducirlo a un problema de set dominante, esto hace que SD sea NP-Hard y por lo tanto **como SD es NP y NP-Hard quiere decir que SD es NP-C**.

2.4. Problema de equipos de socorro es NP-Completo

Teniendo en cuenta la demostración anterior, pensemos lo siguiente.Lo que se busca en el problema de socorro, es dado mapa o las ubicaciones de las estaciones poder cubrir todas las demás, (o al menos estar a 1 estación de distancia) con la menor cantidad de equipos posible, llamaremos K a este valor. Ahora podemos transformar este problema en un problema de set dominante? La respuesta es si.Primero tomaremos dicho mapa o datos y construiremos un grafo que tenga como vértices las distintas estaciones, donde cada arista representara la conexión inmediata de una estación con otra. Si lo que buscamos es dado un numero k, encontrar si es posible cubrir todas las estaciones, en nuestro caso vértices.Entonces una vez transformado en un grafo, la pregunta pasa a ser, es posible encontrar un set dominante de tamaño k, para este mapa? Con lo cual hemos reducido el problema de socorro a uno de SD. Como anteriormente demostramos que SD es efectivamente NP-Completo, entonces, problema de socorro es NP-Hard, ahora bien si podemos demostrar que problema de socorro es NP, entonces el mismo sera NP-Completo.
Pensemos lo siguiente, teniendo la información sobre las estaciones y sus estaciones "vecinas",dadas en forma de grafo(estación = vértice y representando quienes son vecinos como aristas), dado un determinado certificado que indique donde poner a los equipos y una posible solución. Podemos verificar en tiempo polinomial si dicha solución nos dará un resultado correcto o no. Para probar esto podemos tener:

- Un set vació, que llamaremos SA(set auxiliar).
- Un set de los vértices de mi posible solución.
- Recorriendo el grafo

Lo que haremos es por cada vértice, de nuestra solución ir a sus adyacentes, si dicho adyacente no esta en el set SA ni en el set solución (operaciones O(1)), lo agregamos al SA, y pasamos al siguiente, de esta forma continuaremos hasta que no queden mas vértices en el set solución (ya lo

hayamos visitados, todos los posibles) Cuando esto termine verificamos si

$$size(SA) + size(solucion) = size(vertices\ del\ grafo)$$

. Si la igualdad se cumple, el certificado responderá afirmativamente de lo contrario no. Con esto demostramos que Problema de socorro, es un problema de decisión y que además puede resolverse en tiempo polinomial por lo tanto Socorro es NP.Como socorro es NP y NP-Completo entonces, Socorro es un problema NP-Completo.Con esto finaliza nuestra demostración.

2.5. Es posible resolver de forma eficiente "problema de equipos de socorro?

Con lo visto hasta sabemos que el problema a tratar es parte de los problemas NP-Completo. La respuesta a si el problema de decisión se puede resolver de manera tratable el problema es afirmativa, lo que es mas ya definimos una manera de decidir si una solución arrojara un resultado satisfactorio o no.Ahora la pregunta es podemos resolver el **problema** de manera eficiente? La respuesta a esto dependerá de una incógnita aun inconclusa:

$$Es\ \ P = NP?$$

Si la respuesta fuera afirmativa entonces tanto los problemas NP como los P, serian iguales entonces podríamos afirmar que como el problema de decisión es NP, entonces el problema también es P y por lo tanto el problema de socorro se puede resolver el tiempo polinomial, de otra forma tendríamos que encontrar un solución que demuestre que se puede resolver el problema en tiempo polinomial.

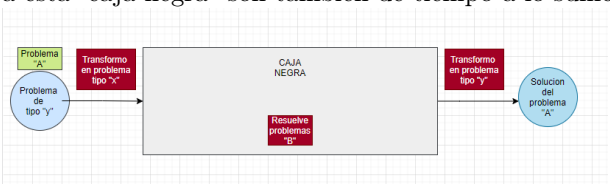
3. Parte 3: Un poco de teoría

1. Defina y explique qué es una reducción polinomial y para qué se utiliza.
2. Explique detalladamente la importancia teórica de los problemas NP-Completos.
3. Tenemos un problema A, un problema B y una caja negra NA y NB que resuelven el problema A y B respectivamente. Sabiendo que B es P.
   a) Qué podemos decir de A si utilizamos NA para resolver el problema B (asumimos que la reducción realizada para adaptar el problema B al problema A es polinomial).
   b) Qué podemos decir de A si utilizamos NB para resolver el problema A (asumimos que la reducción realizada para adaptar el problema A al problema B es polinomial).
   c) ¿Qué pasa con los puntos anteriores si no conocemos la complejidad de B, pero sabemos que A es NP-C?

3.1. Reducción polinomial

Muchas veces a la hora de encarar una problemática, no se conoce o tiene una forma de resolverla, pero en muchos casos, se puede ver o llegar a la conclusión de que dicha problemática es similar o se puede obtener a través del resultado de otra. Esta es la principal función de la reducción polinomial, el de llevar un problema desconocido al dominio de un problema conocido para resolverlo de forma tratable (tomando como la definición de eficiente a los problemas que se pueden solucionar en tiempo polinomial).

Una forma de pensarlo es la siguiente: Se tiene un problema "A" de tipo "y" para el cual no se tiene una solución, lo que si se tiene es una "caja negra" que resuelve un problema "B" de tipo "x", si se puede reducir polinomial mente los problemas de tipo "y" en problemas de tipo "x" entonces puedo resolver el problema polinomial mente, siempre y cuando pueda hacer en dicha complejidad la transformación de un problema tipo "y" a uno de tipo "x".Se considera también que las llamadas a esta "caja negra" son también de tiempo a lo sumo polinomial



Si se cumple todo lo anterior entonces, se dice que "y" es polinomial mente reducible en tiempo a "x"

Otra utilización de la reducción polinomial es como medida de complejidad tanto para compara como clasificar, ya que teniendo la certeza o el dato de que un problema A pertenece cierta clase o tiene cierta complejidad, al probar que un problema B se reduce a uno A podemos acotar y clasificar el problema A que no conocíamos.
Que es lo que se logra con esto? A poder reducir polinomial mente un problema A a otro B, permite saber que el problema A sera **a lo sumo tan complicado como el problema B** en cuanto al coste de su realización.Además de permitir saber si un problema esta en la misma categoría que otro por ejemplo en cuanto a si un problema A existe o no en P, si puedo reducirlo a un problema que exista en P entonces necesariamente A existe en P, porque en el peor de los casos mi problema A se reduce en otro problema que ya es parte de P. Utilizando reducción puedo demostrar otras propiedad como la transitividad y la equivalencia.

3.2. Importancia de los problemas NP-Completos

Antes de desarrollar la importancia respecto a los problemas NP-Completos, es necesario una pequeña explicación respecto a 3 clases: P, NP y NP-Hard.
Se conoce como P al conjunto de problemas que pueden resolverse en tiempo polinomial, problemas como flujo máximo, camino mínimo, entre muchos otros algoritmos ya tratados. Por otro lado tenemos al conjunto o clase NP , que contiene todos los problemas de decisión que por medio de un certificado permiten decidir en tiempo polinomial si una instancia del problema es verdadera o no, responden si o no por ejemplo decidir si se puede abarcar X cantidad de flujo dado una red de flujo F. Además definimos NP-Hard como el conjunto de problemas X que para todo problema Y que exista en NP lo podemos reducir polinomial mente a X. Entonces en este caso diremos que X pertenece a NP-Hard y se dice que X es al menos igual de difícil que cualquier problema NP.
Una vez entendido esto, podemos pasar a explicar NP-C y su importancia, ya que un problema es NP-C si y solo si:

- Es NP Y Es NP-Hard

Cuando un problema pertenece a NP-Completo se puede tomar como que es el problema mas difícil dentro de NP. Su importancia Teórica radica en que desde la demostración del Teorema

de Levin, donde se desarrolla y demuestra que el problema conocido como SAT es NP-Completo permitió a su vez comenzar a probar que mas y mas problemas eran NP-Completo a través de la reducción como se detallo anteriormente.Lo que permite expandir aun mas la capacidad de clasificar y acotar muchos otros problemas, lo que comenzó como unos 21 problemas reducidos a NP-Completo hoy en día son ciento y cientos de algoritmos clasificados gracias a dicho Teorema.

La ultima característica que permitiría definir, el conjunto NP-Completo aun sigue en duda, ya que radica en que incluirá o no dicho conjunto en base a una de las preguntas del milenio: **P = NP**, pero que mientras la duda exista se obtendrá una sola respuesta

3.3. Teniendo un problema A ...

A) Si utilizamos NA para resolver el problema B teniendo en cuenta que la transformación de un problema al otro es polinomial. Como B es P por lo tanto B se puede solucionar en tiempo polinomial. Entonces NA resuelve al problema transformado B en tiempo polinomial, como NA resuelve problemas A y resolvió en tiempo polinomial el problema entonces A pasa a ser P, ya que puede solucionarse en tiempo polinomial

B) En este otro caso como transformo problema A en una instancia de problema B y B es P, entonces NB resuelve polinomial mente los problemas de la forma B entonces A se resuelve en tiempo polinomial y finalmente A pertenece a P.

C) a) Si sabemos que A es NP-Completo, sabemos que dentro de es de los problemas mas difíciles de resolver dentro de NP, como NA resuelve problemas de A, sabemos que B se resolverá en el peor de los casos como NP-Completo.

C) b) Si se sabe que A es NP-Completo sabemos que NB podrá resolver A en al menos NP-Completo por lo tanto B sera NP-Completo.

4. Referencias

"NP-Complete Reductions: Clique, Independent Set, Vertex Cover, and Dominating Set", David Taylor: Algorithms with Attitude, SJSU, San Jose State University, Computer Science Departamet.