75.29 - Teoría de Algoritmos

TP Nº 3

93542 - Bollero, Carlos
98059 - Czop, Guillermo
98017 - Errázquin, Martín
85826 - Escobar, Cynthia

1 Tarjetas coleccionables

Un constructor de máquinas expendedoras de tarjetas coleccionables deportivas nos pide que diseñemos la programación para su nueva máquina que pondrá en diferentes estadios.

Las tarjetas se ofrecen de forma individual o en sobres de diferentes contenidos (por ejemplo podrían ser: 1, 5, 7, 14). Un comprador ingresa el monto que corresponde a X tarjetas y el programa debe entregar la menor cantidad de sobre para cumplir con la solicitud.

Se pide:

1. Analizar la factibilidad de resolverlo mediante un algoritmo greedy
2. Dar pseudocódigo y análisis de complejidad para el punto 1.
3. Resolverlo mediante un algoritmo greedy. Dando su complejidad y pseudocódigo.
4. Resolverlo mediante un algoritmo de programación dinámica. Dando su complejidad y pseudocódigo
5. Programar los 3 algoritmos propuestos.

Información adicional:

- Los programas debe recibir en un archivo de texto la cantidad de tarjetas por sobre. Una por linea
- El programa de factibilidad debe recibir por parámetro el nombre del archivo de sobres y retornar si es factible con greedy o no. en caso de no serlo mostrar un contraejemplo.
- El programa greedy y de programación dinámica debe recibir por parámetro el archivo de sobres y la cantidad de tarjetas a retornar.

1.1 Factibilidad de Solución Greedy

Sea $C = (c_0, c_1, ..., c_{m-1})$ la base de sobres recibida, de tamaño *m*. Se asume que $\forall i = 0, ..., m-2 : c_i < c_{i+1}$, es decir que los elementos de C están estrictamente ordenados. Como primer prueba y de factibilidad a secas, se debe comprobar que $c_0 = 1$, de lo contrario la base nunca va a poder representar todos los números (el ejemplo más simple es intentar representar el 1). Ahora que se sabe que cualquier número X recibido se puede representar en la base C, la pregunta es si la solución greedy es óptima para todo X. Si lo es, se dice que la base C es canónica. Para determinar si la base C recibida es efectivamente canónica se utilizará la cota de Kozen y Zaks, que dice que si una base no es canónica, existe un contraejemplo X' que cumple

$$c_2 < X' < c_{m-1} + c_{m-2} \tag{1}$$

Luego, resta hallar el contraejemplo. Si bien en el paper de Kozen y Zaks se brinda un método para hallar el contraejemplo, el mismo es de orden exponencial respecto de *m*, y como lo que se busca es un valor tal que $sol_{greedy} \neq sol_{dp}$, basta con resolver cada elemento del intervalo anterior por ambas vías y chequear que sean iguales. Si para alguno difieren, entonces se ha hallado el contraejemplo. La justificación de por qué este método es de menor orden que el del paper se brinda a continuación. Nótese que en los casos en que C sea vacía o cuando su primer elemento no sea 1, entonces C no es una base. Además, si C tiene 1 ó 2 elementos siempre será canónica.

1.2 Pseudocódigo y análisis de complejidad de chequeo de base canónica

Sea C el arreglo de tamaño m como antes mostrado.

```
es_factible_greedy(C):
    m=len(C)
    si m==0 OR C[0] != 1:
        lanzar excepción("C no es una base")
    si len(C) <= 2:
        return True, None
    para x = C[2]+1,...,C[m-1]+C[m-2]-1:
        s1,t1 = sobres_greedy(x,C)
        s2,t2 = sobres_dp(x,C)
        si s1 != s2:
            return False, (x,s1,s2)
    return True, None
```

Inicialmente se calcula el tamaño *m* de la base C. Luego, se tienen en cuenta los casos particulares. Por último, para todos los x desde C[2]+1 hasta C[m-1]+C[m-2]-1 se calculan ambas soluciones, que como se verá más adelante son $O(m)$ y $O(mx)$ respectivamente para greedy y DP. Luego, si para algún x no coinciden se devuelve False y una tupla que contiene al contraejemplo con sus respectivas representaciones. De lo contrario, se devuelve True y no hay contraejemplo. Como en el caso de bases canónicas se itera sobre todos los valores del intervalo, este algoritmo se ejecuta en $O(m(m + mx)) = O(m^2 + m^2x) = O(m^2x)$.

Nota: Para todas las soluciones, incluido el cálculo de factibilidad sólo se calcula el tamaño de C para cumplir con la firma requerida. En la práctica sólo debiera calcularse en el chequeo de factibilidad y luego ser pasado como parámetro.

1.3 Solución Greedy

Sea C con el formato antes mostrado y tamaño m, x un número natural, / la división entera (asumida O(1)). Se asume acceso a los arreglos en O(1). S es tal que s_i es la cantidad de sobres de tamaño C[i] a entregar, para todo i de 0 a m-1.

```
sobres_greedy(x,C):
    m=len(C)
    S = array(tamaño=m)
    tam = 0
    para c = m-1,m-2,...,0:
        peso_actual = C[c]
        coef = x / peso_actual
        x -= peso_actual * coef
        S[c] = coef
        tam += coef
    return S,tam
```

1.3.1 Razonamiento

Considérese el peso de un sobre como la cantidad de tarjetas que trae. El algoritmo itera sobre los pesos de los sobres en orden decreciente (índice reverso sobre C por estar ordenado) y para cada uno calcula la cantidad máxima de sobres *coef* de ese peso que puede entregar (que es el cociente entero entre la cantidad X de tarjetas a entregar y el peso actual). El resto es el nuevo número X a representar con los restantes pesos, que se calcula como la diferencia entre el X original y el producto del coeficiente entero por el peso actual para asegurar que se calcula en O(1)(*). Además, tam es una variable que indica la cantidad de sobres utilizados, o dicho de otra forma, el "tamaño" de S. Si bien se puede calcular por separado sin alterar el orden de la solución (es la suma de los elementos de S, resoluble en O(m)), se explicitó en clase que era importante devolverlo acompañando a S y por tanto se lo agrega al algoritmo como una magnitud más. Nuevamente, asumiendo $c_o = 1$ siempre habrá solución para todo X.
(*) También se puede calcular como $X = X\%peso\_actual$.

1.3.2 Complejidad

Nótese que c toma exactamente m valores, y en cada ciclo se efectúan todas operaciones asumidas O(1). Luego, el algoritmo corre en tiempo $O(m)$.

1.4 Solución DP

Para las mismas condiciones que la solución greedy,

```
sobres_dp(x,C):
    m=len(C)
    min_sobres = array(tamaño=x+1)
    min_sobres[0] = 0
    ult_idx_sobre = array(tamaño=x+1)
    para x_parcial = 1,...,x:
        min_sobres_act = x+1
        ult_idx_sobre_act = -1
        para idx_sobre = 0,...,m-1:
            peso_actual = C[idx_sobre]
            si x_parcial < peso_actual:
                break
            posible_tam = 1 + min_sobres[x_parcial - peso_actual]
            si posible_tam < min_sobres_act:
                min_sobres_act = posible_tam
                ult_idx_sobre_act = idx_sobre
        min_sobres[x_parcial] = min_sobres_act
        ult_idx_sobre[x_parcial] = ult_idx_sobre_act
    S = array(valor=0,tamaño=m)
    tam = 0
    mientras x>0:
        idx_sobre = ult_idx_sobre[x]
        peso = C[idx_sobre]
        S[idx_sobre] += 1
        x -= peso
        tam += 1
    return S, tam
```

1.4.1 Razonamiento

A saber:

- Para cada n=0,...,x-1 min_sobres_n da la cantidad mínima de sobres para representar el número n en la base C; es decir el tamaño de la solución óptima de n. Cada elemento se inicializa con x+1 que hace las veces de infinito, dado que la máxima cantidad de sobres para representar un número n es n sobres de 1 tarjeta.
- Para cada n=0,...,x-1 $ult\_idx\_sobre_n = k$ indica que la solución óptima de n es un sobre de C[k] tarjetas más la solución óptima de n-C[k].
- Por simplicidad y facilidad de lectura el caso base queda cubierto imponiendo que cubrir el 0 requiere 0 sobres.

El algoritmo se basa en que $\forall x_{parcial} = 1, ..., x$ s.t. $x_{parcial} - C[j] >= 0$:

$$min\_sobres[x_{parcial}] = \min_{j=0,...,m-1} 1 + min\_sobres[x_{parcial} - C[j]] \tag{2}$$

La solución se obtendrá de la siguiente manera: para cada x_parcial desde 1 hasta x se obtendrá su solución óptima. Teniendo en cuenta la ecuación anterior, para cada x_parcial la solución óptima será 1 sobre más la solución óptima de otra ya calculada. Para cada tamaño de sobre $C[idx\_sobre]$ que sea a lo sumo x_parcial se calcula el posible menor tamaño y se lo guarda si es menor al mínimo hasta ahora obtenido, además del sobre utilizado. Cuando x_parcial = x haya sido resuelto, se habrá encontrado la solución óptima para el mismo, y sólo resta ir recuperando cada sobre utilizado para obtener la representación S. Por último, tam funciona como en la solución greedy, incrementándose en uno por cada sobre agregado a la solución.

1.4.2 Complejidad

El ciclo while realiza en el peor caso (con $C = (1)$) *x* iteraciones, por lo que es $O(x)$. El ciclo de x_parcial realiza exactamente *x* pasos, y el ciclo de idx_sobre a lo sumo *m* ciclos. Cada iteración del ciclo de idx_sobre se resuelve en O(1) dado que sólo hace operaciones y accesos a memoria asumidos en tiempo constante. Entonces, la primer parte del algoritmo corre en $O(mx)$. Así, el algoritmo se ejecuta en $O(mx+x) = O(mx)$, donde *x* es el número a representar y *m* es la cantidad de elementos de la base.

1.5 Programación de las 3 partes

2 Redes de Subterráneos

La Ciudad de buenos aires cuenta con una red de subterráneos compuesta por 6 lineas.
Un análisis de frecuencia en hora pico ha determinado:

- La demanda de flujo de pasajeros entrantes y salientes por cada estación por hora.
- La capacidad máxima de vagones por tramo por hora entre estaciones a causa del estado de las vías.
- La cantidad máxima de personas que pueden transportarse por vagón.

Por cuestiones contractuales se debe tener por linea una cantidad mínima de vagones por hora. Por último, la cantidad máxima de vagones por hora del sistema está limitada por el número total de vagones disponible en la flota.

Determinar si dados estos valores es factible proporcionar un servicio adecuado.
Consideraciones:

- Considerar que el número de personas que entra y sale sumado de todo el sistema es 0.
- Existe la cantidad necesaria de locomotoras para la demanda.
- Se pueden conmutar vagones entre estaciones.

Se pide:

1. Analizar el problema teóricamente y explicar paso a paso como determinar la factibilidad.
2. Dar pseudocódigo y análisis de complejidad.
3. Programar la solución y proporcionar dos set de datos de prueba (uno factible y el otro no factible).

2.1 Análisis y factibilidad

Para analizar y modelar el problema de la red de subterráneos de la ciudad de Buenos Aires, planteamos las siguientes variables basadas en los datos, restricciones y condiciones iniciales:

Variable	Descripción	Unidad
i	Estación perteneciente a alguna de las 6 líneas de subterráneos	
p_{max}	Cantidad máxima de personas por vagón	$\frac{personas}{vagon}$
$C_{i,j}$	Capacidad del tramo que va desde la estación i a la estación j	$\frac{vagon}{hora}$
$f_{IN}(i)$	Demanda de flujo entrante de la estación i	personas
$f_{OUT}(i)$	Oferta de flujo saliente de la estación i	personas
VL_i	Cantidad de vagones en la línea i	vagones
$min(VL_i)$	Mínima cantidad de vagones en la línea i	vagones
VS	Cantidad de vagones en el sistema	vagones
VF	Cantidad de vagones en la flota	vagones

El problema consiste en determinar si dada una configuración inicial para las variables previamente definidas, se puede brindar, durante una hora, un buen servicio de transporte. Brindar un buen servicio consiste en:

- el tráfico de un tramo no supera la capacidad máxima de transporte de dicho tramo.
- la cantidad de vagones por línea supera la cantidad mínima, es decir $min(VL_i) \leq VL_i$
- la cantidad de vagones necesarios para transportar a todas las personas en el sistema no supere la cantidad de vagones en la flota (VF).
- a nivel sistema, la cantidad total de tráfico saliente debe ser igual al total de tráfico entrante.

Para analizar la factibilidad de que se brinde un servicio apropiado modelaremos este problema como una red de flujo donde:

- la red de subtes será el grafo $G = (V, E)$.
- cada vértice $v \in V$ corresponde a una estación de alguna de las líneas A,B,C,D,E,H.
- cada arista $e \in E$ corresponde a un tramo (i, j) con i, j estaciones de la red. Es decir, $e = (u, v), u, v \in V$.
- la demanda neta del vértice v es $d(v) = f_{IN}(v) - f_{OUT}(v)$
- la capacidad máxima de la arista e es $c(e) = C_{u,v} * p_{max}$
- para cada arista $e \in E,$ el flujo debe ser $0 \leq f(e) \leq c(e)$

Calcularemos la demanda $d(v) = f_{IN}(v) - f_{OUT}(v)$ para cada $v \in V$, y los clasificaremos de la siguiente manera:

$$\begin{cases} \text{Si d(v) > 0} & \Rightarrow \ v \text{ es un vértice sumidero ya que recibe más flujo del que envía.} \\ \text{Si d(v) < 0} & \Rightarrow \ v \text{ es un vértice fuente ya que envía más flujo del que recibe.} \\ \text{Si d(v) = 0} & \Rightarrow \ v \text{ es un vértice interno.} \end{cases} \tag{3}$$

De lo anterior se puede observar que en la red de subterráneos cada una de las estaciones puede producir o recibir tráfico, con lo que existen múltiples fuentes y sumideros. Definimos S como el conjunto de todos los vértices fuentes $S = \{v \,/\, d(v) < 0\}$, y T como el conjunto de todos los vértices sumidero $T = \{v \,/\, d(v) > 0\}$.

Recordemos que para que exista un flujo factible $\Rightarrow \sum_v d(v) = 0$. Definimos así

$$D = \sum_{u \in T} d(u) = \sum_{v \in S} -d(v)$$

Una vez llevado el problema original a una red de flujos la forma más sencilla de encontrar un camino factible (o determinar que no es posible brindar un buen servicio) es reduciendo el problema de la red de subtes a un problema de flujo máximo de la siguiente manera:

2.1.1 Reducción

Sea X= Problema de la red subterránea e Y= Problema de Flujo Máximo de s' a t':

2.1.2 Entrada de Y

- Grafo $G' = (V', E')$
- Vértice s' y t'

Construiremos el grafo G' a partir del grafo G, agregando los nuevos nodos s' y t', que serán fuente y sumidero respectivamente.
Conectamos la nueva super-fuente s' con cada vértice $v_i \in S$ mediante las aristas

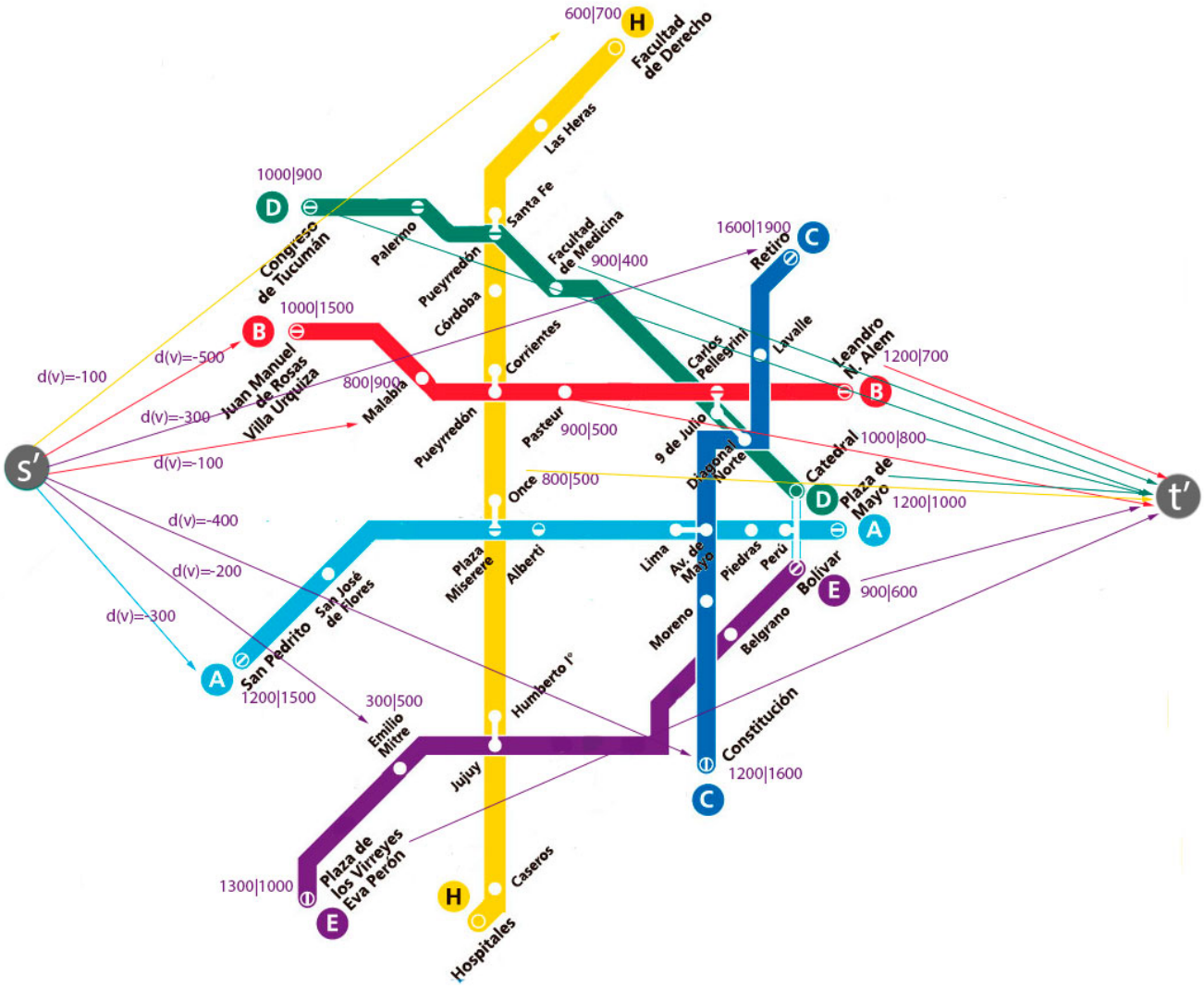
$$e_{s',v} = (s', v) \,/\, c(e_{s',v}) = -d(v),$$

y conectamos cada vértice $u_j \in T$ con el super-sumidero t' mediante las aristas

$$e_{u,t'} = (u, t') \,/\, c(e_{u,t'}) = d(u)$$

Tomando como ejemplo la siguiente tabla de demandas de flujo entrante y saliente, en el gráfico (4) se ilustran como quedarían incorporados los nuevos super-fuente s' y super-sumidero t' a las fuentes y sumideros originales de la red de subterráneos, para los que se visualiza f_{IN} y f_{OUT}.
Se asume que para el resto de los nodos se cumple $f_{IN} - f_{OUT} = 0$.

Estación v	$f_{IN}(v)$	$f_{OUT}(v)$	Demanda $d(v)$	Fuente	Sumidero
Rosas	1000	1500	-500	✓	
Malabia	800	900	-100	✓	
Pasteur	900	500	400		✓
Alem	1200	700	500		✓
Congreso de Tucumán	1000	900	100		✓
Fac Medicina	900	400	500		✓
Catedral	1000	800	200		✓
Fac Derecho	600	700	-100	✓	
Once	800	500	300		✓
San Pedrito	1200	1500	-300	✓	
Plaza de Mayo	1200	1000	200		✓
Pza de los Virreyes	1300	1000	300		✓
Emilio Mitre	300	500	-200	✓	
Bolivar	900	600	300		✓
Retiro	1600	1900	-300	✓	
Constitución	1200	1600	-400	✓	



Una vez creado el nuevo grafo G', resolveremos el problema Y, es decir que buscaremos el máximo flujo de s' a t', teniendo en cuenta que no puede haber un flujo de s' a $t' > D$ ya que el corte (A, B) con $A = s'$ sólo tiene capacidad D (recordemos que $D = \sum_{v \in S} -d(v) = \sum_{u \in T} d(u)$ y que la capacidad de las aristas $e_{s',v} = -d(v), v \in S$). Supongamos que hay un flujo factible f con demandas $d(v)$ en G entonces si enviamos flujo de $-d(v)$ sobre cada arista (s', v) y flujo de $d(v)$ sobre cada arista (v, t') obtenemos un flujo $f'_{s',t'} = D$ en G', que será máximo (por la limitación D previamente indicada). Análogamente supongamos que hay un máximo flujo $s' - t'$ en G' de valor D entonces quiere decir que cada arista saliente de s' y cada arista entrante a t' están totalmente saturados con flujo. Por esto, si eliminamos los vértices ficticios s' y t' obtenemos un flujo f en G que cumple la condición de que $f_{IN}(v) - f_{OUT}(v) = d(v)$ para cada vértice $v \in V$. Con esto podemos asegurar que

si hay un flujo $f'_{s',t'} = D$ en G' habrá un flujo factible en G.

2.2 Pseudocódigo

```
V=[]
E=[]
g=(V,E)
# demandas
d=[]

# capacidades
c = []

S=[]
T=[]

para cada vértice v en V:
    d[v] = f_in(v) - f_out(v)
    if d[v] < 0:
        S.add(v)
    else if d[v] > 0:
        T.add(v)

copiar G' a partir de G

agregar vértice s' a G'
agregar vértice t' a G'

para cada vértice v en S
G'.addEdge(s',v,-d[v])

para cada vértice u en T
G'.addEdge(t',u,d[u])

flujo = 0
para cada arista (u, v) in G':
    flujo(u, v) = 0

mientras haya un camino p de s' -> t' en el grafo residual G_f:
    capacidad_residual(p) = min(capacidad_residual(u, v) : for (u, v) in p)
    flujo = flujo + capacidad_residual(p)
    para cada arista (u, v) en p:
        if (u, v) es una arista de retroceso:
            flujo(u, v) = flujo(u, v) - capacidad_residual(p)
        else:
            flujo(u, v) = flujo(u, v) + capacidad_residual(p)

demandaLineaA = sum(demanda vertices Linea A en flujo) % maxCantidadDePersonasXvagon
demandaLineaB = sum(demanda vertices Linea B en flujo) % maxCantidadDePersonasXvagon
demandaLineaC = sum(demanda vertices Linea C en flujo) % maxCantidadDePersonasXvagon
demandaLineaD = sum(demanda vertices Linea D en flujo) % maxCantidadDePersonasXvagon
```

```
demandaLineaE = sum(demanda vertices Linea E en flujo) % maxCantidadDePersonasXvagon
demandaLineaH = sum(demanda vertices Linea H en flujo) % maxCantidadDePersonasXvagon

demandaSistema = demandaLineaA + demandaLineaB + demandaLineaC + demandaLineaD +
demandaLineaE + demandaLineaH

si demandaLineaA < minLineaA
    exit('No es factible brindar un buen servicio')

si demandaLineaB < minLineaB
    exit('No es factible brindar un buen servicio')

si demandaLineaC < minLineaC
    exit('No es factible brindar un buen servicio')

si demandaLineaD < minLineaD
    exit('No es factible brindar un buen servicio')

si demandaLineaE < minLineaE
    exit('No es factible brindar un buen servicio')

si demandaLineaH < minLineaH
    exit('No es factible brindar un buen servicio')

si demandaSistema > vagonesEnLaFlota
    exit('No es factible brindar un buen servicio')

print ('Es factible brindar un buen servicio')
```

Dado que la solución utiliza Ford Fulkerson se supone a priori que la complejidad es O(|E|*f) dónde |E| es el número de aristas del grafo y f es el flujo máximo del grafo. Pero para este caso hace falta analizar la complejidad incurrida al transformar el grafo inicial en el usado en la resolución del algoritmo.
Para la transformación se itera sobre todos los |V| vértices del grafo para calcular la demanda del mismo y clasificarlos como fuentes o sumideros. Esta operación es de órden O(|V|). Como el grafo es conexo se sabe que como mínimo debe haber |V|-1 aristas (el grafo sería un árbol) por lo que la complejidad de esta operación en el mejor de los casos es parecida, y para los demás no es superior a la de la resolución mediante Ford-Fulkerson.
O(|V|) $\simeq O(|E| * f)$, suponiendo que |E| = |V|-1 y f no es lo suficientemente grande.
$O(|V|) \leq O(|E| * f)$,para otros casos.
Por lo tanto la complejidad de la solución propuesta es de O(|E|*f)

3 Requisitos y ejecución

3.1 Parte 1

- Requisitos:
  Python 3, Numpy
- Ejecución:
  python collectableCards.py numbreDeArchivo modo cantCartas

  Donde modo: "A" (análisis de factibilidad), "G" (greedy) o "D" (dinámico)
  en el caso de elegir modo "A" no es necesario insertar la cantidad de cartas.

3.2 Parte 2

- Requisitos:
  Python3
- Ejecución Punto 2:
  - ./subtes.py variablesFilepath demandasFilepath capacidadesFilePath

  En el archivo de variables se cargan las variables con el siguiente formato:

  ```
  #maxPersonasXvagon,minVagonesLineaA,minVagonesLineaB,
  minVagonesLineaC,minVagonesLineaD,minVagonesLineaE,minVagonesLineaH,vagonesFlota
  ```

  En el archivo de demandas, se cargan las demandas entrante y salientes por estación:

  ```
  # demandas estación [personas] =  estación,f_in,f_out
  ```

  En el archivo de capacidades se cargan la cantidad máxima de vagones soportadas por las vías de un tramo i,j:

  ```
  # capacidades x tramo [vagones] = estacion\_i,estacion\_j,capacidad
  ```

4 Código

4.1 Parte 1

```
import numpy as np

def analisys_greedy(packets):
  m = len(packets)
  if(m == 0 or packets[0] != 1):
    print("ERROR")
    sys.exit(1)
  if(m <= 2): return True, None
  for cards in range(packets[2]+1, packets[m-1]+packets[m-2]):
    s1, q1 = greedy(cards, packets)
    s2, q2 = dynamic(cards, packets)
    if(np.not_equal(s1, s2).all()):
      return False, (cards, s1, s2)
  return True, None

def greedy(cards, packets):
  m = len(packets)
  Sol = np.zeros(m, np.int8)
  q = 0
  for c in range(m-1,-1,-1):
    size = packets[c]
    coef = cards // size
    cards -= coef * size
    Sol[c] = coef
    q += coef
  return Sol, q
```

```
def dynamic(cards, packets):
  m = len(packets)
  min_packets = np.zeros(cards + 1, np.int8)
  last_packet_idx = np.zeros(cards + 1, np.int8)
  for x in range(1, cards+1):
    current_min_packets = cards + 1
    current_last_packet_idx = -1
    for i in range(0, m):
      current_size = packets[i]
      if(x < current_size): break
      q = 1 + min_packets[x - current_size]
      if(q < current_min_packets):
        current_min_packets = q
        current_last_packet_idx = i
    min_packets[x] = current_min_packets
    last_packet_idx[x] = current_last_packet_idx
  S = np.zeros(m, np.int8)
  q = 0
  while(cards > 0):
    idx_packet = last_packet_idx[cards]
    size = packets[idx_packet]
    S[idx_packet] += 1
    cards -= size
    q += 1
  return S, q
```