Trabajo final: Algoritmos de flujo máximo

Dal Bianco, Pedro Alejandro

Taller de Tecnologías de Producción de Software

Facultad de Informática, UNLP - 2020

Índice

1. Introducción

Este trabajo desarrolla acerca de los llamados problemas de flujo máximo. En primer lugar se define el problema junto con algunos conceptos claves para su comprensión. Luego se presentan dos soluciones distintas para dicho problema: el método de Ford-Fulkerson y los algoritmos llamados de *push-relabel*, cada uno con su respectiva implementación, el algoritmo de Edmonds-Karp y el algoritmo *relabel to front*. Además, se abordan dos problemas de este tipo, y se desarrollan e implementan sus respectivas soluciones utilizando los algoritmos mencionados.

2. Definición del problema del flujo máximo

Los problemas de flujo máximo planteados informalmente, suelen tener una forma similar a la de “dado un conjunto de cañerías conectadas entre sí, cada una con su respectiva capacidad ¿cuál es la máxima cantidad de flujo que se puede transportar de un punto inicial ‘a’ a otro punto final ‘b’?”.

Desde un enfoque más formal, definimos una **red de flujo** como:

- Un **grafo dirigido** $G = (V, E)$.
- Una función **capacidad** c que para cada arista $a \in E$ le asigna una **capacidad** $c(a)$ perteneciente al conjunto de los números enteros positivos.
- Dos vértices $s, t \in V$ origen y destino respectivamente.

Un **flujo** en G es una función f que le asigna a cada arista $a \in E$ un valor $f(a)$ perteneciente al conjunto de los números enteros positivos, también llamado flujo de dicha arista, que debe cumplir dos condiciones:

- El flujo de una arista no puede superar su capacidad
  - $\forall\ a \in E,\ f(a) \leq c(a)$
- Para todo vértice $v \in V$, la suma del flujo entrante de v debe ser igual a la suma del flujo saliente de v
  - $\forall\ v \in V,\ \sum_{(u,v) \in E} f((u,v)) = \sum_{(v,u) \in E} f((v,u))$

Además, el vértice origen s solo tiene flujo saliente, y el vértice destino t solo flujo entrante. Dadas estas condiciones, se cumple que el total del flujo saliente de s es igual al total del flujo entrante de t:

- $\sum_{(s,v) \in E} f((s,v)) = \sum_{(v,t) \in E} f((v,t))$

A este valor se lo conoce como el flujo de una red, y el problema de flujo máximo de una red consiste en, dada una red de flujo, encontrar el máximo flujo posible para esa red.

En las figuras 1.a y 1.b se ve un ejemplo de una red de flujo y de el máximo flujo en dicha red respectivamente.

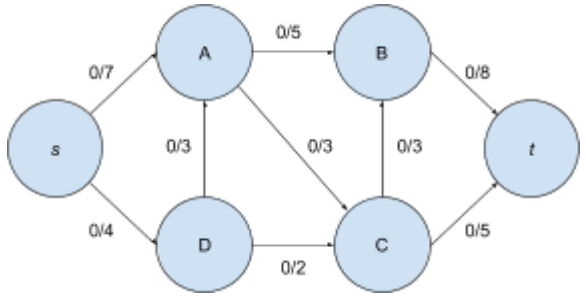


Figura 1.a

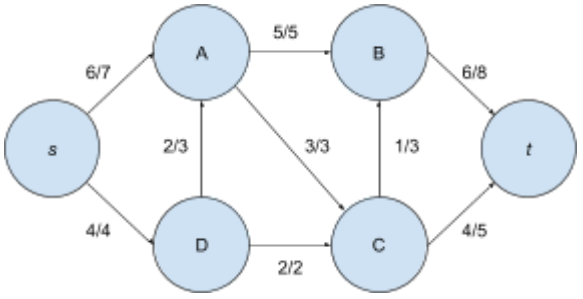


Figura 1.b

---

3. Soluciones al problema del flujo máximo

3.1 Método de Ford-Fulkerson

3.1.1 Definiciones

Para explicar el método de Ford-Fulkerson en primer lugar se deben definir dos conceptos, el de red residual y el de camino de aumento.

La **capacidad residual** de una arista a es la diferencia entre su capacidad y su flujo ($c(a) - f(a)$).

Una **red residual** de una red de flujo es entonces una red con los mismos vértices que la red original y con dos aristas por cada arista de la red. Por cada arista (x,y) en la red original, en la red residual habrá:

- Una arista $(x,y)'$ con un flujo igual a la capacidad residual de (x,y)
- Una una arista invertida $(y,x)'$ con capacidad igual a 0, y flujo igual al flujo de (x,y) negado ($-f(x,y)$).

Un **camino de aumento** es simplemente un camino del vértice origen al destino en la red residual a través de aristas con capacidad residual mayor a 0, que nos servirá para aumentar el flujo en el original.

3.1.2 Método

El método de Ford-Fulkerson funciona de la siguiente manera, dada una red de flujo:

1. Inicializamos el flujo de cada arista en 0
2. Mientras encontremos un camino de aumento incrementamos el flujo de la red de la siguiente manera:
    2.1. Sea m la menor de las capacidades residuales de las aristas pertenecientes al camino encontrado
    2.2. Para cada arista (v,u) perteneciente al camino encontrado, en la red residual se decrementa el flujo de (v,u) y se incrementa el flujo de (u,v) en m unidades, lo que en la red original se ve de la siguiente manera:
        2.2.1. Si la arista (v,u) pertenece a la red original se aumentará su flujo en la red sumandole m
        2.2.2. Si la arista (v,u) no pertenece a la red original, si no que es una arista inversa generada por la existencia de flujo positivo de u a v, se disminuirá el flujo de la arista (u,v) en m
3. Cuando no existan más caminos de aumento, significa encontramos el flujo máximo de la red.

3.1.3 Ejemplo

Se ilustra ahora un ejemplo de uso de este método partiendo de la misma red de la figura 1.a con el flujo de cada arista inicializado en 0, que en primera instancia se corresponde con su red residual. Esto se muestra en la figura 2.a

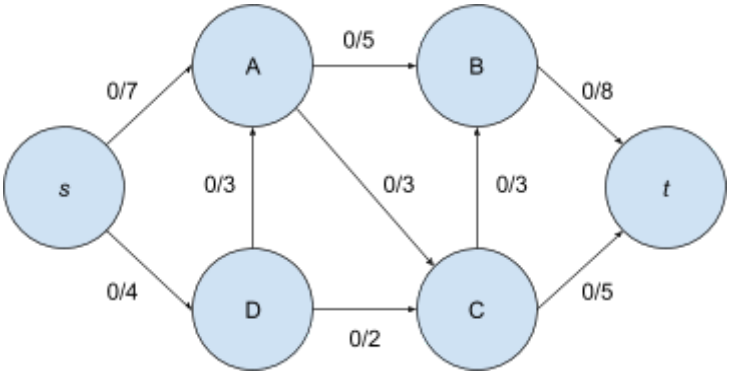


Figura 2.a

Encontramos entonces el camino de aumento s, A, B, t, donde la capacidad mínima entre sus aristas es la de la arista (A, B), de 5, por lo que actualizaremos el flujo de todas las aristas del camino en 5, obteniendo las siguientes red y red residual mostradas en las figuras 2.b y 2.c respectivamente.

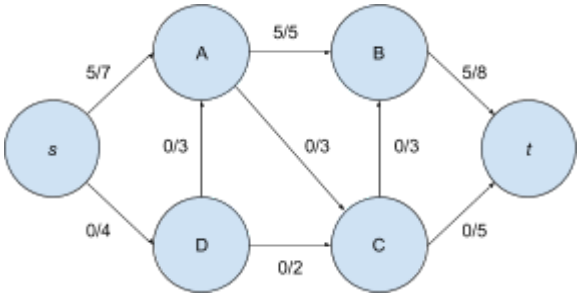


Figura 2.b

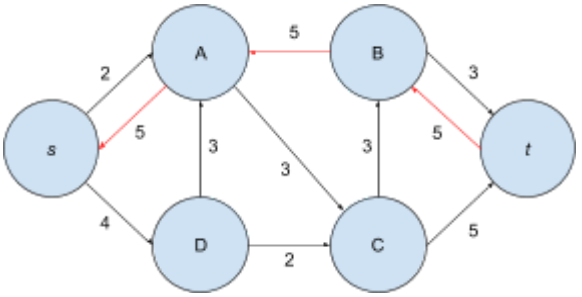


Figura 2.c - Las aristas en rojo son las aristas invertidas, y se omiten las de flujo 0 ya que no sirven para encontrar un camino de aumento

Podemos encontrar ahora el camino de aumento s, D, A, C, t, donde la capacidad mínima entre sus aristas es la de las aristas (D, A) y (A, C), de 3, por lo que actualizaremos el flujo de todas las aristas del camino en 3, y las red y red residual resultantes se ven en las figuras 2.d y 2.e respectivamente.

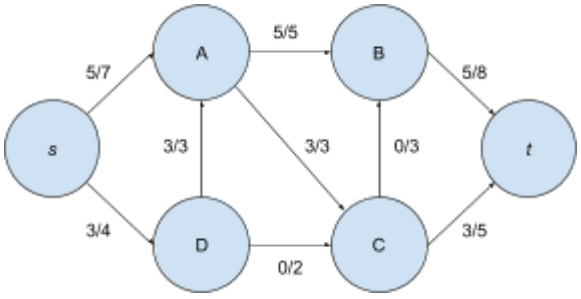


Figura 2.d

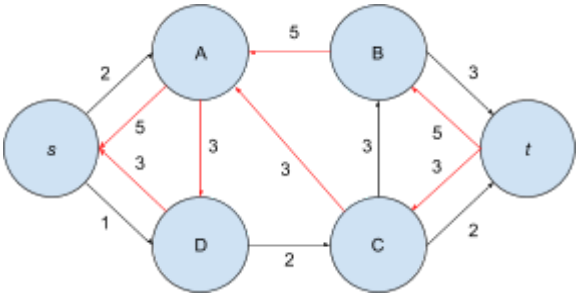


Figura 2.e

Realizamos el mismo procedimiento para el camino s, D, C, B, t, con capacidad mínima 1, lo que resulta en las redes de las figuras 2.f y 2.g.

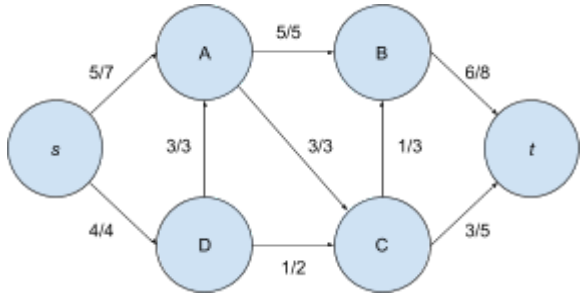


Figura 2.f

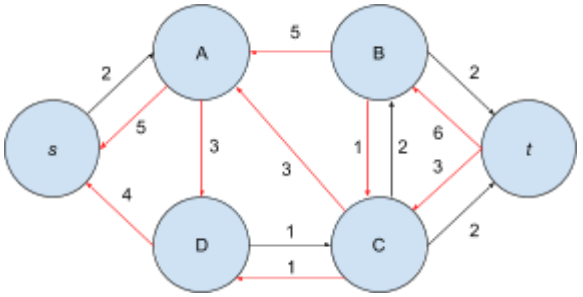


Figura 2.g

Podemos encontrar ahora el camino s, A, D, C, t, con capacidad mínima de 1, pero que además tiene la particularidad de contener la arista (A, D), que no pertenece a la red original, por lo que no podemos enviar flujo del vértice A al vértice D. Esto se puede interpretar de la siguiente manera, dado que la arista (A, D) en la red residual representa que hay un flujo de 3 del vértice D al A, lo que hacemos cuando encontramos un camino que incluye dicha arista es "cancelar" parte de ese flujo, por lo que estamos aumentando la capacidad de flujo de salida de D, y compensando en A, en este caso con flujo proveniente de s.

Por esto es que como se indica en el inciso 2.2.2 de la sección Método, actualizaremos la arista (D, A) de la red original restándole 1, resultando esto en las red y red residual mostradas en las figuras 2.h y 2.i.

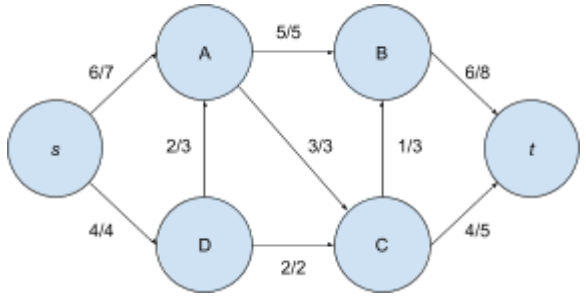


Figura 2.f

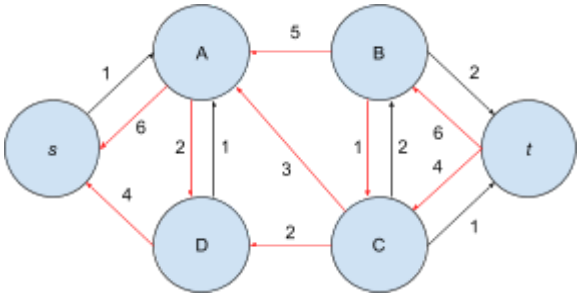


Figura 2.g

Dadas estas redes, ya no es posible encontrar ningún camino de aumento, por lo que hemos encontrado el máximo flujo posible para esta red, que es de 10.

Como aclaración final respecto a este ejemplo, es necesario mencionar que para la implementación de este método no se requiere mantener la red original luego de cada iteración, es suficiente con disponer de la red residual. En este caso se mantuvieron ambas con el fin de clarificar el funcionamiento del método.

3.1.4 Consideraciones

Resulta de interés notar que se lo define como método de Ford-Fulkerson y no como algoritmo ya que no está completamente definido: no define una función para encontrar los caminos de aumento. Sin embargo sabemos que con cualquier implementación de dicha función de orden lineal sobre las aristas del grafo ($O(E)$), dada una red con un flujo máximo F , la complejidad de Ford-Fulkerson será de $O(EF)$.

Esto se debe a que cada iteración incrementara el flujo a lo sumo en 1, por lo que se realizarán como máximo F iteraciones.

3.1.5 Implementación: algoritmo de Edmonds-Karp

El algoritmo de Edmonds-Karp es una implementación del método de Ford-Fulkerson que utiliza un BFS para encontrar caminos de aumento. Se demuestra que esta implementación tiene una complejidad de $O(VE^2)$, independientemente del flujo máximo [3].

Aquí se presenta una implementación realizada en python 3 de dicho algoritmo:

Definidos un grafo *'capacities'* y otro grafo *'flows'* donde se almacenan las capacidades y flujos de las aristas de la red respectivamente, es trivial definir una función para obtener la capacidad residual de una arista (x, y) :

```
def residual(x,y):
    return (capacities[x][y] - flows[x][y])
```

Luego se utiliza un recorrido BFS para encontrar un camino del vértice inicial s al vértice final t. Esto no difiere de un recorrido BFS tradicional con el agregado de que se deben considerar solo las aristas con una capacidad residual mayor a 0. Al finalizar el recorrido se obtiene y devuelve la capacidad residual mínima de las aristas del camino:

```
def findPath(s,t,prev):
    visited = [False for i in range(V)]
    q = queue.Queue()
    q.put(s)
    while not q.empty():
        v = q.get()
        if v == t:
            break
        visited[v] = True
```

```
        availables = list(map(lambda x: x[0],filter(lambda x:
residual(v,x[0]) > 0 and not visited[x[0]],
enumerate(capacities[v][:V]))))
        for adj in availables:
            visited[adj] = True
            prev[adj] = v
            q.put(adj)

    minCapacity = 0
    current = t
    while prev[current] != -1:
        minCapacity = min([residual(prev[current],current),
minCapacity]) if minCapacity != 0 else residual(prev[current],current)
        current = prev[current]
    return minCapacity
```

Por último, la función 'maxFlow' itera mientras encuentre un nuevo camino de aumento válido, actualizando el valor total del flujo de la red y los valores del grafo de flujos:

```
def maxFlow(s,t):
    flow = 0
    prev = [-1 for i in range(V)]
    pathCapacity = findPath(s,t,prev)
    while pathCapacity != 0:
        flow += pathCapacity
        current = t
        while prev[current] != -1:
            flows[prev[current]][current] += pathCapacity
            flows[current][prev[current]] -= pathCapacity
            current = prev[current]

        prev = [-1 for i in range(V)]
        pathCapacity = findPath(s,t,prev)
    return flow
```

En el archivo adjunto *edmonds_karp.py*[1] se encuentra la implementación completa y comentada.

[1] https://drive.google.com/open?id=1ntp11BhAsNFnQBkFZK7Yqsm9onREUnuz

3.1.6 Otras aplicaciones del algoritmo de flujo máximo

Teorema del flujo máximo del corte mínimo

Un **corte s-t** en una red de flujo es una partición del conjunto de vértices en dos conjuntos S y T, tal que uno contenga el vértice origen s y el otro contenga el vértice destino t respectivamente. La **capacidad** del corte está dada por la suma de las capacidades de todas las aristas que cuyo origen es un vértice del conjunto S y su destino es un vértice del conjunto B. Un corte s-t es mínimo si no existe otro corte s-t con capacidad menor a este.

Resulta intuitivo pensar que el flujo máximo de una red está acotado por la capacidad de cualquier corte s-t de esta red, más específicamente por el corte mínimo, ya que no podrá llegar más flujo a los vértices del conjunto T (incluido el vértice destino) del que permita la capacidad de dicho corte. El teorema del flujo máximo del corte mínimo va más allá y establece que el **flujo máximo** de una red de flujo es igual a la **capacidad del corte mínimo** de esta misma red.

Es posible obtener un corte s-t mínimo de la siguiente manera: una vez encontrado un flujo máximo de la red, definir el conjunto S como los vértices alcanzables desde el origen s a través de aristas de la red residual, y $T = V - S$.

Se procede a ilustrar esto a partir de la red de flujo ya presentada en la figura 2.f y su respectiva red residual presentada en la figura 2.g. En la figura 3 se presenta un corte mínimo de esta red, obtenido de la forma descrita anteriormente, donde los vértices coloreados en azul pertenecen al conjunto S y los vértices coloreados en rojo pertenecen al conjunto T. Se observa que la capacidad del corte es de $10 = 5 + 3 + 2$, que es igual al valor del flujo máximo de la red.

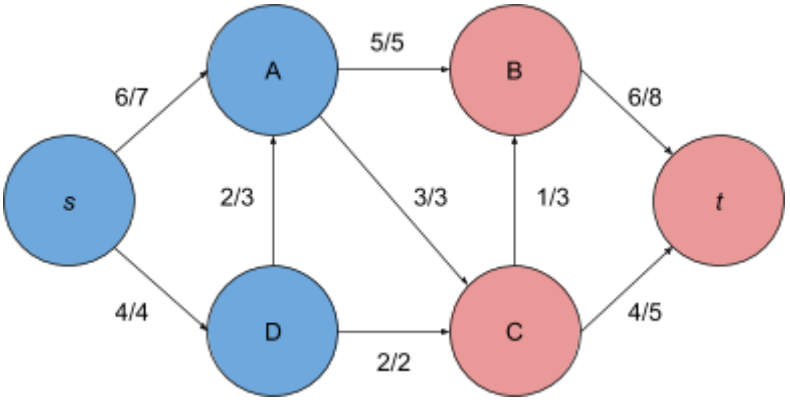


Figura 3

Entonces, utilizando los métodos ya descritos, hemos podido encontrar un corte mínimo en una red de flujo. Esto resulta útil, por ejemplo, en problemas donde dado un grafo

$G = (V, E)$ pesado, se requiera eliminar un conjunto de aristas que sumen el menor peso posible, tal que un vértice t se vuelva inalcanzable desde un vértice s. Si se construye una red de flujo con los s y t dados, donde la capacidad de cada arista este dado por el peso de la arista del grafo original, y se encuentra el corte s-t mínimo de la forma explicada anteriormente, las aristas a eliminar serán las que tengan un origen en un vértice del conjunto S y un destino en un vértice del conjunto T.

Emparejamiento máximo de grafos bipartitos

Para ilustrar este problema, resulta necesario primero enunciar algunas definiciones:

- Un grafo $G = (V, E)$ es **bipartito** si el conjunto de sus vértices se puede particionar en dos conjuntos $L \cup R = V$, tal que L y R son disjuntos y no existen aristas que vayan de un vértice de un conjunto a otro del mismo. Es decir, todas las aristas son entre vértices de L y R.

- Un emparejamiento (*matching*) es un subconjunto de aristas $M \subseteq E$, tal que todo vértice del grafo está conectado **a lo sumo** con una arista de M.

- Un emparejamiento es máximo si su cardinalidad es mayor o igual a la de cualquier otro emparejamiento posible.

Un ejemplo de problema de emparejamiento máximo en un grafo bipartito puede ser: dado un conjunto de empleados y un conjunto de tareas, donde cada empleado es capaz de realizar algunas de esas tareas, pero puede ser asignado sólo a una de estas, encontrar la mejor forma de asignar tareas a los empleados tal que la cantidad de tareas a las que nadie fue asignado sea mínima. En este caso los vértices del grafo bipartito estarán representando al conjunto de empleados y de tareas, y cada arista (u, v) de este representará que el empleado u es capaz de realizar la tarea v.

Es posible encontrar un emparejamiento máximo en un grafo bipartito $G = (V = L \cup R,\ E)$ utilizando un algoritmo de flujo máximo de la siguiente manera:

- Se crea una red de flujo con los vértices del grafo, una fuente s y un destino t.

- Se conecta a los vértices pertenecientes a L al vértice origen a través de una arista de capacidad 1.

- Se conecta a los vértices pertenecientes a R al vértice destino a través de una arista de capacidad 1.

- Para cada arista (u, v) del grafo original se crea una arista (u, v) en la red con capacidad 1, donde el que haya flujo a través de esa arista representará el emparejamiento de u con v.

La figura 4.a muestra un ejemplo de grafo bipartito, donde se ven en rojo las aristas que forman parte de un emparejamiento máximo. La figura 4.b muestra la red de flujo construida a partir de ese grafo tal como se indicó anteriormente, donde las aristas marcadas en rojo es donde una vez encontrado el flujo máximo tendrán un flujo de 1.

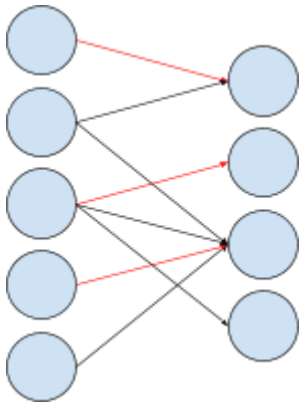


Figura 4.a

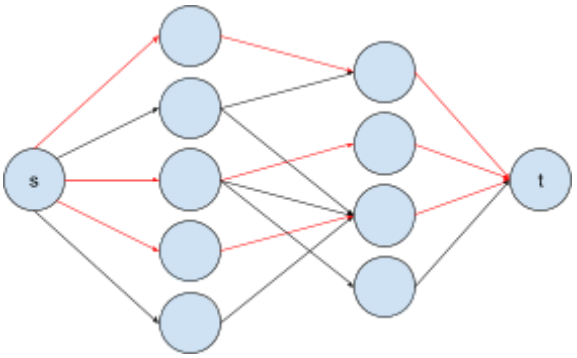


Figura 4.b

3.1.7 Problemas resueltos utilizando el algoritmo de Edmonds-Karp

POTHOLE[2]

Enunciado

Este problema nos plantea que hay un grupo de espeleólogos que quieren estudiar una cueva que consiste de distintas cámaras conectadas, comenzando por una cámara superior y llegando a una cámara inferior. Las cámaras están numeradas de forma ascendente y se sabe que la cámara $n+1$ se encuentra por debajo de la cámara n y los espeleólogos solo pueden descender. Por último aclara que los caminos salientes de la cámara superior y los entrantes a la cámara inferior pueden ser recorridos por a lo sumo un espeleólogo, mientras que el resto de los caminos no tiene límite de capacidad. El problema requiere que indiquemos cuantos espeleólogos pueden llegar a la cámara inferior.

Planteo

Se ve claramente que el enunciado se corresponde con un problema de flujo máximo, representado de la siguiente manera:

[2] https://www.spoj.com/problems/POTHOLE/

- Cada cámara representa un nodo de la red de flujo, con la cámara superior como vértice inicial y la cámara inferior como vértice final.

- Las aristas que parten del vértice inicial o que llegan al vértice destino tienen capacidad 1.

- Sabemos que el flujo máximo de la red puede ser a lo sumo la suma de la capacidad de todas las aristas que parten del vértice inicial, y siendo que todas estas aristas tienen una capacidad de uno, sabemos que el flujo máximo no superará la cantidad de vértices adyacentes al vértice inicial, por lo que se toma este valor como capacidad del resto de las aristas.

Implementación

Teniendo en cuenta dichas consideraciones, y siendo que la entrada del problema llega de la forma de una linea por vertice, listando en cada una la cantidad y sus respectivos adyacentes, se definió la función para construir la red de la siguiente manera:

```
def buildGraph(v):
    resetGraph()
    for i in range(0,v-1):
        adj = list(map(lambda x: int(x)-1,input().split(' ')[1:]))
        if i == 0:
            maxval = len(adj)
        for j in adj:
            capacities[i][j] = (1 if (i == 0 or j == v-1) else maxval)
            capacities[j][i] = 0
```

Luego, se implementaron las mismas funciones para calcular el flujo máximo mostradas en la implementación del algoritmo de Edmonds-Karp, con la salvedad de que dado que no se requiere para la solución del problema almacenar la red de flujos, y que ningún vértice puede estar conectado a un vértice anterior, no se utilizó la red de flujos y se almaceno toda la información en la red de capacidades.

Por último, el programa principal se encarga de procesar la entrada obteniendo el total de casos de prueba y llamando a función *'maxFlow'* con cada uno de ellos, con los primer y último vértices como vértices origen y destino respectivamente. La implementación completa se encuentra en el adjunto *pothole.py*[3].

[3] https://drive.google.com/open?id=1LHYwvxysZAeoA9ifGCR-W4ZkLyRkdAHY

Road construction[4]

Enunciado

Este problema se presenta de la siguiente manera, dadas N ciudades, M materiales y K trabajadores:

- Cada ciudad quiere construir un camino bidireccional a otra de las N ciudades.
- Si una ciudad i quiere construir un camino a la ciudad j, se garantiza que la ciudad j no quiere construir un camino también a la ciudad i.
- Todas las ciudades están conectadas por una secuencia de propuestas de caminos. Es decir que si se construyeran todos los caminos, todas las ciudades estarían conectadas.
- Cada camino puede estar hecho de un subconjunto de los M materiales, definido por la ciudad que desea construirlo.
- Cada uno de los K trabajadores manipula un solo material, y puede construir un solo camino que pueda ser construido con dicho material.

Entonces, debemos encontrar una forma de asignar a cada trabajador a un camino que se pueda construir con el material que este manipula, tal que se construyan los caminos necesarios para que todas las N ciudades se encuentren conectadas, e imprimir qué camino se le asignó a cada trabajador.

Planteo

Para afrontar este problema primero debemos pensar cuáles son los caminos que debemos construir. Dado que cada ciudad tiene una propuesta de camino, en total hay N posibles caminos a construir. Esto planteado en forma de grafo no dirigido, donde cada ciudad es un nodo y cada camino una arista, tenemos N nodos y N posibles aristas, donde por las condiciones planteadas en el enunciado sabemos que son todas distintas. Sin embargo, $N-1$ aristas son suficientes para que el grafo sea conexo. Por esto sabemos que, de construir todos los caminos **seguro se formará un ciclo** en el grafo de ciudades.

Esto lo utilizaremos para definir qué caminos se deben construir. Dado el grafo de ciudades con todos los posibles caminos como aristas, donde se forma el ciclo C, sabemos que si una ciudad $c \notin C$, entonces necesariamente debemos construir el camino propuesto por c que la conecta al resto de las ciudades para que el grafo sea conexo. Ahora, sabiendo que las $|C|$ ciudades pertenecientes a C forman un ciclo, entonces solo debemos construir $|C|-1$ caminos de los propuestos por estas ciudades, y aun así esta componente se mantendrá conexa.

Ahora debemos definir cómo asignamos los caminos a los trabajadores, y aquí es donde lo planteamos en forma de problema de flujo. Para esto creamos una red de flujo que

[4] https://codeforces.com/contest/1252/problem/L

contendrá un nodo por cada solicitud de camino, y uno por cada material para el que haya al menos un trabajador que lo manipule.

Respecto a los nodos de cada solicitud de camino para cada ciudad i:

- Si la ciudad i está fuera del ciclo mencionado anteriormente (es decir que se debe construir el camino solicitado por dicha ciudad), el nodo correspondiente a su solicitud de camino se conecta a la fuente de la red a través de una arista de capacidad 1.

- Si la ciudad i pertenece al ciclo mencionado, se la conecta a un vértice $fuente'$ con una arista de capacidad 1. Este vértice $fuente'$ se conecta a la fuente real de la red con una arista de capacidad $|C| - 1$ (donde C es el ciclo mencionado), y sirve para limitar el flujo que llegara a este conjunto de ciudades para que no se construyan caminos de más que pudieran utilizar recursos necesarios para otros.

Además, este nodo estará conectado a cada nodo de un material m, tal que dicho camino puede ser construido con con ese material m, con una arista de capacidad 1.

Respecto a los nodos de materiales, además de lo ya mencionado, cada uno estará conectado a el nodo destino de la red, a través de una arista con capacidad igual a la cantidad de trabajadores capaces de manipular este material.

Se procede a ilustrar esto con el ejemplo dado en el problema, donde hay 4 ciudades y 5 trabajadores que manipulan los materiales 1 a 5 respectivamente, y las solicitudes de caminos están planteadas de la siguiente manera:

- La ciudad 1 quiere construir un camino a la ciudad 2, con los materiales 1 o 2
- La ciudad 2 quiere construir un camino a la ciudad 3, con los materiales 2 o 3
- La ciudad 3 quiere construir un camino a la ciudad 4, con los materiales 3 o 4
- La ciudad 4 quiere construir un camino a la ciudad 2, con los materiales 4 o 5

Se observa que se forma un ciclo entre las ciudades 2, 3 y 4. La red entonces, construida como se definió anteriormente, se ve en la figura 5.a. Las ciudades están representadas por los nodos *c1* a *c4*, y los materiales por los nodos *m1* a *m5*. La capacidad de las aristas que no está especificada de 1. En la figura 5.b se observa la red luego de maximizar el flujo.

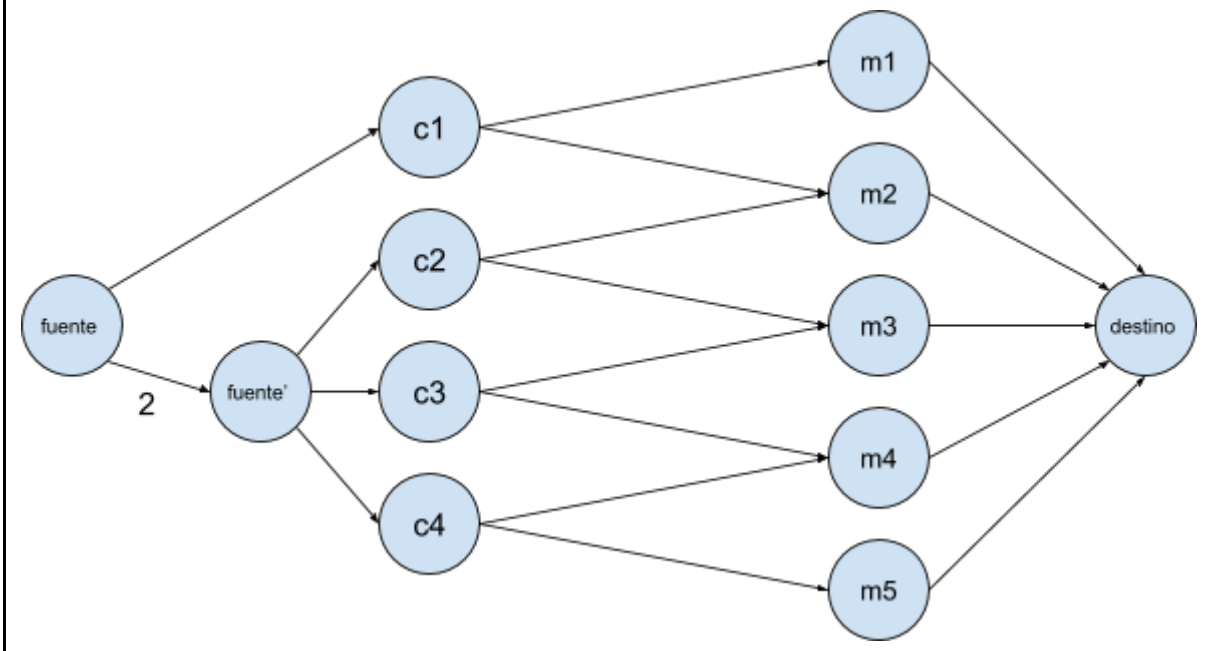


Figura 5.a

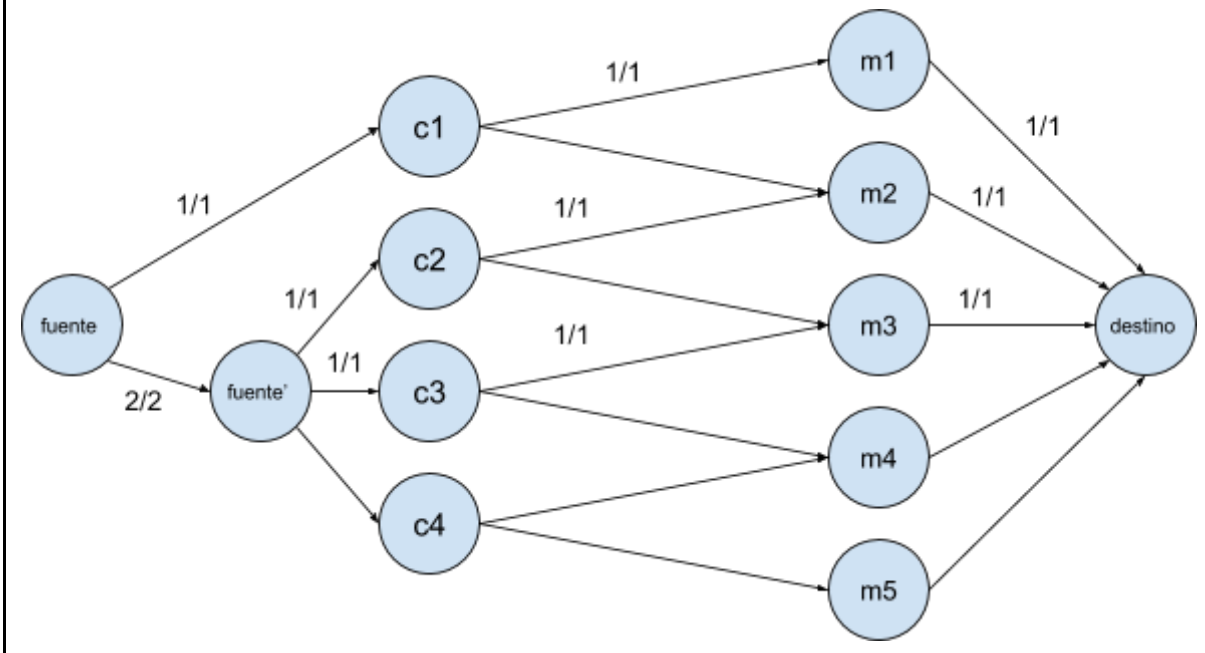


Figura 5.a

A partir de la última imagen, obtenemos que la máxima cantidad de caminos que podemos construir es 3 (el flujo de la red), que el camino de la ciudad 1 se construirá con el material 1, el de la ciudad 2 se construirá con el material 2 y el de la ciudad 3 con el material 3. Y esto es suficiente para que todas las ciudades se encuentren conectadas.

Entonces, teniendo esta red obtendremos que camino le corresponde a cada trabajador viendo para cada uno que material es el que manipula, y viendo para ese material que ciudad le está enviando flujo. En caso de ser varias ciudades puede ser cualquiera de estas, garantizando que elija una que no haya elegido antes otro trabajador. En caso de no encontrar ninguna ciudad que cumpla con estas condiciones, entonces dicho trabajador no construirá ningún camino.

Implementación

La entrada consiste de dos números N y K indicando la cantidad de ciudades y de trabajadores respectivamente. Se leen estos números y se crean arreglos para almacenar los caminos, los materiales posibles para cada camino y la lista de adyacencias del grafo de ciudades.

```
// roads: ciudad i construye camino a roadList[i]
vector<int> roads(N);
// materialsForRoad: el camino de la ciudad i se puede construir con los materiales de materialsForRoad[i]
vector<vector<int>> materialsForRoad(N);
// adjCities: lista de adyacencias en grafo de ciudades y caminos
vector<vector<int>> adjCities(N);
```

Las siguientes N líneas contienen los datos correspondientes a cada ciudad y su solicitud de camino a construir, y se utilizan para llenar los arreglos definidos.

Luego se define un arreglo para almacenar la lista de trabajadores y el material que manipulan, otro para almacenar la cantidad de trabajadores que manipulan determinado material, y un diccionario que asigna a cada material que sea manipulado por al menos un trabajador un índice que permita luego ubicarlo en la red y también obtener la cantidad de trabajadores que lo manipulan.

```
vector<int> workerList(K);
// workers: cantidad de trabajadores que manipulan el material i
vector<int> workers;
// materialToIndex: Dado un material te da el índice para encontrarlo en workers
map<int, int> materialToIndex;
```

A partir luego de la última línea de entrada que contiene la lista de trabajadores se cargan estos arreglos.

Se realiza luego un DFS definido en la función '*findCycle*' sobre el grafo de ciudades, y se carga el arreglo '*cycle*' con verdadero para los vértices que forman parte del ciclo que se forma entre estas.

Entonces es que se comienza a definir la red de flujos, compuesta por los mismos elementos y utilizando las mismas funciones que las definidas anteriormente (salvo por algunas pequeñas diferencias de implementación por cuestiones de eficiencia). Se define el nodo 0 como fuente, el nodo 1 como el extra para limitar el flujo en las ciudades pertenecientes al ciclo, y el último nodo como destino, y procede a crearse la red como fue definida en la sección de planteo.

```
// Un vértice por cada ciudad, uno por cada material, un origen (0) y un destino (V-1), y uno extra para limitar el flujo para los nodos del ciclo (1)
V = (N + 1) + materialToIndex.size() + 2;
capacities.resize(V, vector<int>(V));
adj.resize(V);
```

```
 int source = 0;
 int extraV = 1;
 int sink = V-1;

 // Conecto el vértice para limitar el flujo, y conecto las ciudades
dentro del ciclo a este, mientras que las que estén fuera del ciclo a
la fuente
 addEdge(source, extraV, count(cycle.begin(), cycle.end(), true) - 1);
 for (int i = 0; i < N; ++i) {
   if (cycle[i] && cycle[roads[i]]) {
     addEdge(extraV, i+2, 1);
   } else {
     addEdge(source, i+2, 1);
   }
   for (int mat : materialsForRoad[i]) {
     if (materialToIndex.count(mat)) {
        // Conecta el nodo de la ciudad i (i+2) con el de los posibles
materiales
       addEdge(i+2, N+2+materialToIndex[mat], 1);
     }
   }
 }

 // Conecta los nodos de los materiales con el destino
 for (int i = 0; i < materialToIndex.size(); ++i) {
   addEdge(N+2+i, sink, workers[i]);
 }
```

Hecho esto, se calcula el flujo máximo usando el algoritmo ya descrito e implementado (las funciones '*maxFlow'* y *'findPath*'), y luego si el flujo es menor a la cantidad de ciudades disminuida en 1, significa que no se pueden construir los caminos necesarios, por lo que se imprime "-1" y finaliza el programa. Caso contrario, si es posible construir efectivamente los caminos necesarios se procede a: para cada uno de los K trabajadores buscar que ciudad está enviando flujo a su respectivo material (en la red se ve una arista residual con flujo 1), marcar esa arista como ya utilizada (asignándole un flujo de 0), e imprimir la propuesta de camino para esa ciudad en la posición correspondiente a dicho trabajador. En caso de no encontrar ninguna ciudad se imprime "0 0".

La implementación completa y comentada se encuentra en los archivos adjuntos con el nombre '*road_construction*'.

Aclaración

Se realizaron 2 implementaciones de este problema, que lo resuelven de manera similar, como se describe en la sección anterior. La primer implementación fue realizada en python, pero no se pudo lograr que no se excediera en tiempo. Se buscaron realizar todas las optimizaciones, y algunas se basaron en la implementación oficial[5]. Sin embargo esto tampoco fue suficiente y se supone que es porque el tiempo está definido en función de la velocidad de un programa C++ (no varía en función del lenguaje). Por lo que se tradujo la implementación realizada a C++ y entonces fue aceptada por el juez.

3.2 Algoritmos *push-relabel*

3.2.1 Definiciones

Se presenta en esta sección un enfoque distinto al ya mostrado de Ford-Fulkerson llamado de *push-relabel* (empuje y renombramiento, en español). Este enfoque utiliza los conceptos ya definidos en el anterior, pero trabaja de forma más localizada, analizando de a un vértice a la vez, y en lugar de mantener un flujo válido que se aumenta a medida que avanza, utiliza un **preflujo** f, que se define de la misma forma que un flujo, pero relajando la propiedad de conservación de la siguiente manera:

- Para todo vértice $v \in V$, la suma del flujo entrante de v debe ser **mayor o igual** a la suma del flujo saliente de v

  - $\forall\, v \in V, \sum_{(u,v) \in E} f((u,v)) \geq \sum_{(v,u) \in E} f((v,u))$

Esto da lugar a definir el **excedente** de un vértice v (distinto del vértice origen s), que se corresponde con la diferencia entre el flujo entrante de y el flujo saliente de v:

- $\forall\, v \in V - \{s\},\ e(v) = \sum_{(u,v) \in E} f((u,v)) - \sum_{(v,u) \in E} f((v,u))$

Si el excedente de un vértice v es mayor a 0 ($e(v) > 0$) se dice que ese vértice está **desbordado**.

Por último se define una función de etiquetado o **altura** h, que dado un vértice nos retorna un valor entero que representa la **altura** de ese vértice. Esta función debe cumplir con las siguientes condiciones:

- Para los vértices origen s y destino t siempre se cumple $h(s) = |V|$ y $h(t) = 0$.
- Dados dos vértices $u, v \in V$, si existe una arista (u,v) en la **red residual** (es decir, es posible enviar flujo de u a v), entonces $h(u) \leq h(v) + 1$.

[5] https://github.com/jonathanirvings/icpc-jakarta-2019/blob/master/construction/solution.cpp

Es importante mencionar que en caso de existir una función de altura válida para la red, **no** existe ningún camino de aumento en la red residual. Esto se comprueba observando que un camino de aumento puede contener a lo sumo $|V|-1$ aristas, y la altura de cada vértice en el camino puede superar a la del siguiente por a lo sumo 1 (por la segunda propiedad de la función de altura), por lo que resulta imposible que comenzando por el vértice de origen t con altura $|V|$ se llegue al vértice destino s con altura 0.

Entonces, la idea de los algoritmos de este tipo será la de ir mejorando el preflujo hasta que este sea un flujo, siempre manteniendo una función de etiquetado válida, lo que nos garantiza que no existirán caminos de aumento. Esto lo realiza a través de dos operaciones que tal como lo indica su nombre se llaman ***push*** y ***relabel***.

Operación *push*

Para poder realizar la operación ***push*** entre un vértice u y otro vértice v, u debe estar desbordado ($e(u) > 0$), la capacidad residual de la arista (u,v) debe ser mayor a 0, y $h(u) = h(v) + 1$. Esta operación trata de enviar todo el flujo posible de u a v, que será el valor mínimo entre el excedente de u y la capacidad residual de la arista (u,v).

Se dice que una operación push **satura** si luego de realizar la operación la capacidad residual de la arista (u,v) tiene el valor 0. Si una operación push no satura, podemos garantizar que luego de realizarla el excedente de u será 0.

Operación *relabel*

La operación ***relabel*** para un vértice u aplica cuando u está desbordado y para toda arista (u,v) de la red residual, $h(u) \leq h(v)$. Es decir que u no puede enviar flujo a los vértices para los que tiene capacidad residual por no encontrarse a la altura necesaria.

Entonces, la operación consiste en darle un nuevo valor a la altura de u, tal que $h(u) = 1 + min(\{h(v) : (u,v) \in E_R\})$, siendo E_R el conjunto de aristas de la red residual. En otras palabras, se actualiza la altura de u con un valor que supere por uno a la altura mínima de los vértices a los que les puede enviar flujo.

3.2.2 Algoritmo genérico de *push-relabel*

El algoritmo genérico de *push-relabel* consta de dos partes:

- Una **inicialización**, donde se inicializan todos los flujos en 0, salvo los de las aristas que parten del vértice origen, a las que se las inicializa con un flujo igual a su capacidad. Luego, la altura y el excedente de los vértices de la red se inicializan en 0, salvo en el vértice origen, para el cual se inicializa la altura en |V| y el excedente como el valor opuesto a la suma del flujo saliente de este.

- Una parte iterativa, donde mientras haya un vértice sobre el cual se puede aplicar una operación *push* o *relabel*, se aplica la operación correspondiente.

Sabemos que cuando termine la iteración, ningún vértice estará desbordado, ya que de estarlo se podría aplicar sobre dicho vértice una de las propiedades mencionadas. Entonces, nuestro preflujo será además un **flujo**.

Se demuestra que tanto la operación *push* como *relabel* mantienen las propiedades que hacen de h una función de altura válida, y dado que como se mostró en la definición de la función altura, mientras haya una función de altura válida, podemos garantizar que no existe ningún camino de aumento, entonces al terminar la iteración el flujo encontrado será máximo.

Se demuestra además que este algoritmo pertenece al orden $O(V^2E)$.

3.2.3 Algoritmo *relabel to front*

El algoritmo de *push-relabel* genérico no establece un orden en el que se deben realizar operaciones sobre los vértices. Una implementación de esto es el algoritmo ***relabel to front***, que utiliza una lista de vértices, y a medida que la va recorriendo los **descarga**, es decir, aplica tantas operaciones *push* y *relabel* como sea posible hasta que el vértice no tenga más excedente, y en caso de haber aplicado una operación relabel, se vuelve a colocar este vértice al comienzo de la lista. Es importante notar que la lista a utilizar debe obtenerse a traves de un *sort* topológico del grafo original.

3.2.4 Ejemplo

Se ilustra un ejemplo de la ejecución del algoritmo *relabel to front*, partiendo de la red de la figura 6.a, ya inicializada con el preflujo correspondiente. En la figura 6.b se ve la red residual correspondiente, y se muestra dentro de cada vértice desbordado su excedente. Además se muestra la altura de cada vértice en la tabla a la derecha y la lista de vértices utilizada por este algoritmo.

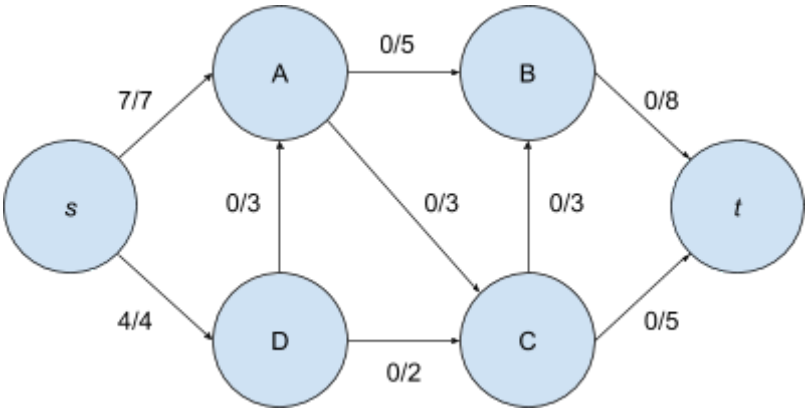
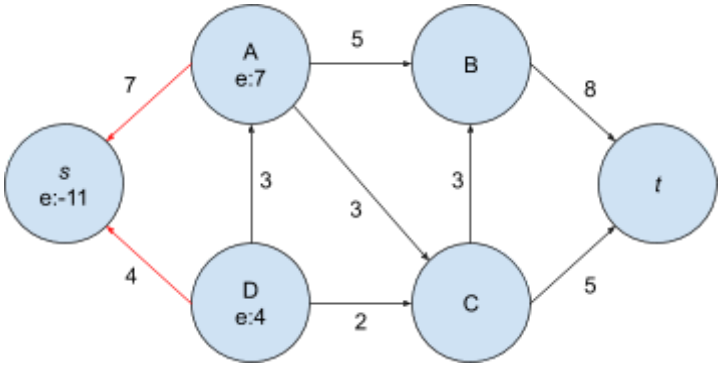


Figura 6.a

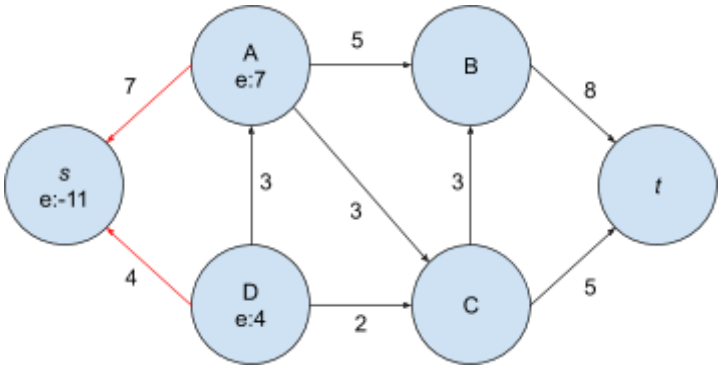


Lista: A, D, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	
2	
1	
0	A,B,C,D,*t*

Figura 6.b

Entonces, se comienza a **descargar** el vértice A, por ser el primero en la lista, y se intenta enviar flujo a los vértices accesibles a través de la red residual (B, C, s), sin embargo, esto no resulta posible para ninguno ya que A se debiera encontrar a una altura que supere por 1 a la del vértice a la que le envia flujo, entonces se aplica *relabel(A)*, resultando en la figura 6.c.

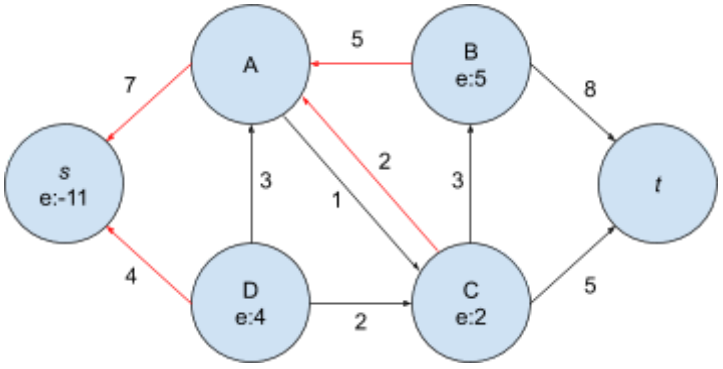


Lista: A, D, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	
2	
1	A
0	B,C,D,*t*

Figura 6.c

Al encontrarse A a altura 1, ahora puede realizar un *push* de 5 a B, y un push de 2 a C. Esto deja a A con excedente 0, por lo que no se pueden realizar más operaciones sobre A, lo que se refleja en la figura 6.d (al haber hecho un *relabel* de A, este vértice se envia al comienzo de la lista, pero al ya ser el primer vértice no se ven cambios).

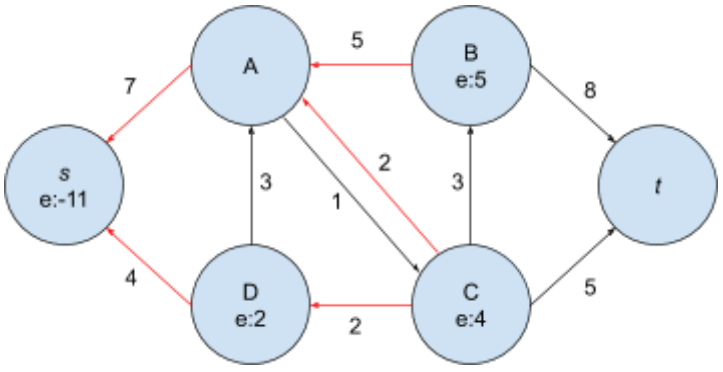


Lista: A, D, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	
2	
1	A
0	B,C,D,*t*

Figura 6.c

Luego, comienza la descarga del vértice D. Al igual que en el caso anterior, deberemos realizar un *relabel* de D actualizando su altura con un valor que supere por 1 la altura mínima de los vértices a los que puedo enviar flujo (en este caso el vértice de menor altura al que puedo enviar flujo es D, con altura 0). Entonces, se realizará un push de 2 (lo permitido por la arista (D, C)) de D a C. El resultado de esto se ve en la figura 6.d.

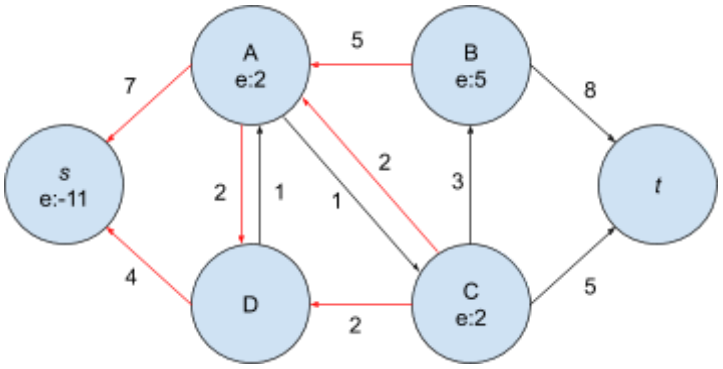


Lista: A, D, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	
2	
1	A,D
0	B,C,*t*

Figura 6.d

Sin embargo, luego de este *push* no finalizó la descarga de D, aún tiene un excedente de 2, por esto es que se vuelve a buscar un vértice al que le pueda transferir flujo (*s* o A), pero no poder hacerlo por no encontrarse a la altura adecuada se realiza de nuevo un *relabel(D)*, esta vez actualizando la altura de D con $h(A)+1$, y luego realizando un *push(D,A)* de una cantidad de flujo 2. Una vez realizada esta operación, efectivamente terminó la descarga de D, (quedó con excedente 0), y por haber realizado un *relabel*, se lo coloca al principio de la lista, resultando esto en la figura 6.e.

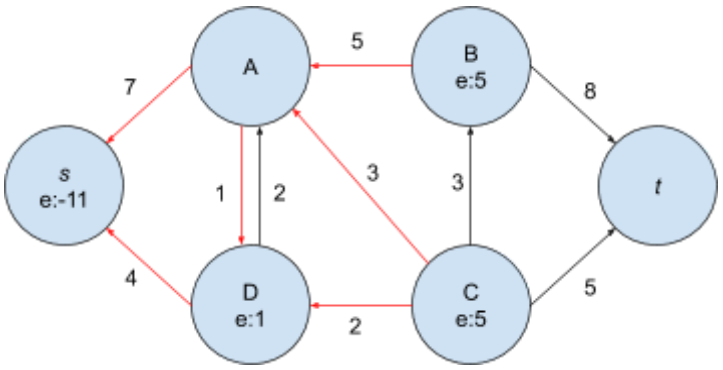


Lista: D, A, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	
2	D
1	A
0	B,C,*t*

Figura 6.e

Ahora, el siguiente elemento de la lista es el vértice A. En este caso, es posible en principio hacer un *push* de A a C, de una cantidad 1. Sin embargo, luego de esto A sigue con un exceso de 1, por lo que continúa la descarga de este vértice. El vértice de menor altura al que le puede trasladar flujo es D, por lo que se actualiza la altura de A con $h(D)+1$ y se hace un *push(A,D)* de cantidad 1. Luego de esto se completó la descarga de A, y se vuelve a colocar al frente de la lista como se ve en la figura 6.f.



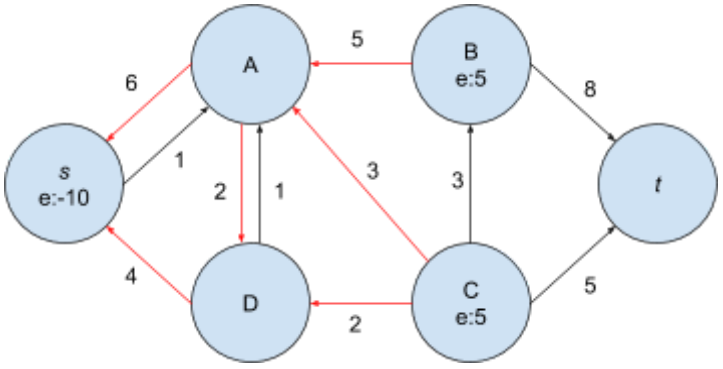
Lista: A, D, C, B

Altura	Vértices
7	
6	*s*
5	
4	
3	A
2	D
1	
0	B,C,*t*

Figura 6.f

Podemos observar que ahora, tanto para A como para D solo quedan como vértices posibles para enviar flujo a ellos mismos y a *s*, por lo que en las siguientes iteraciones se incrementará la altura de A y B hasta que uno supere la altura de *s* (en este caso A), y

luego el excedente será enviado de nuevo al origen, dando como resultado lo que se observa en la figura 6.g.

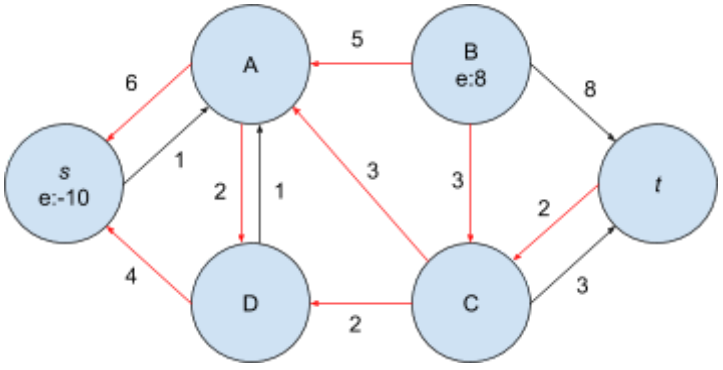


Lista: A, D, C, B

Altura	Vértices
7	A
6	*s*,D
5	
4	
3	
2	
1	
0	B,C,*t*

Figura 6.g

Al avanzar más sobre la lista, no se realizará ninguna operación sobre D, ya que no está desbordado. Luego, para el caso de C, dado que los vértices a los que puede enviar flujo con mínima altura son B y *t* (de altura 0), se actualizará su altura con el valor 1, y luego se hará un push a uno de ellos. En este ejemplo primero a B, enviándole una cantidad de 3 (capacidad de la arista (C,B), y luego los 2 restantes a *t*. El resultado de esto se ve en la figura 6.h.



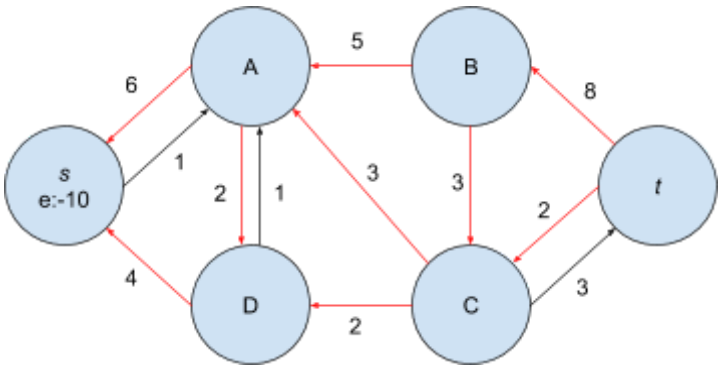
Lista: C, A, D, B

Altura	Vértices
7	A
6	*s*,D
5	
4	
3	
2	
1	C
0	B,*t*

Figura 6.h

Como se realizó un *relabel* sobre C, este es colocado al principio de la lista, sin embargo, dado que ni A ni D están desbordados, procedemos a descargar el vértice B, para el que le aumentaremos la altura a 1, ya que el vértice de menor altura al que puede enviar flujo es

t. Luego, realizará un *push* de 8 a *t*, tras el cual B ya no estará desbordado, tal como se ve en la figura 6.i.



Lista: B, C, A, D

Altura	Vértices
7	A
6	*s*,D
5	
4	
3	
2	
1	C,B
0	*t*

Figura 6.i

Podemos observar que ya no queda ningún vértice con excedente positivo, por lo que terminamos de recorrer la lista de vértices sin realizar ninguna operación sobre ninguno. Entonces hemos encontrado un flujo máximo. La red de flujo correspondiente a la red residual de la figura 6.i se ve en la figura 6.j.

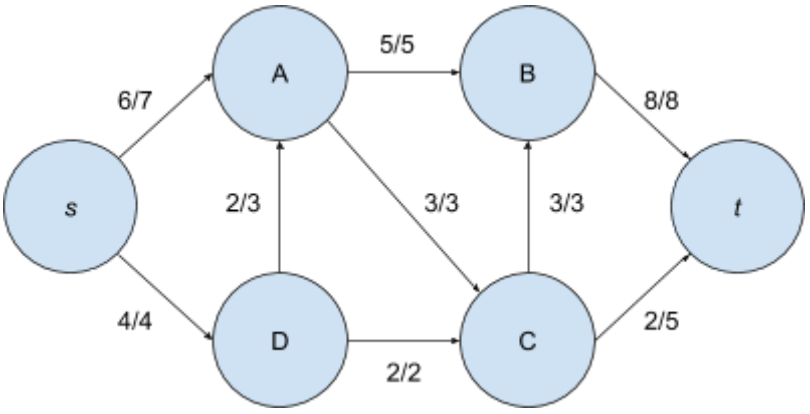


Figura 6.j

3.2.5 Implementación

Se realizó una implementación en *python* de este algoritmo, definiendo las funciones *push*, *relabel* y *discharge* (descargar). Para obtener el siguiente vértice al cual descargar en orden constante se almacenan los vértices a los que se les realiza un push en una cola (*overflowed*). Además se utiliza la lista *seen* como una cola circular para recorrer los vértices

a los que uno puede acceder desde un vértice dado. Esta implementación se encuentra en el archivo *push_relabel.py*[6].

El método *push(u,v)* tal como se lo describió, toma el mínimo entre el excedente de *u* y la capacidad residual de la arista *(u,v)* y lo almacena en *d*. Luego realiza push de esa cantidad *d* actualizando los excedentes de *u* y *v*, y el flujo entre ellos. Por último, si el vértice *v* no se encontraba en la lista de vértices desbordados se lo agrega.

```
def push(u, v):
    d = min([excess[u], residual(u,v)])
    flow[u][v] += d
    flow[v][u] -= d
    excess[u] -= d
    excess[v] += d
    if (d and excess[v] == d):
        overflowed.put(v)
```

El método *relabel(u)* toma la altura mínima de entre los vértices a los que *u* puede enviar flujo y le asigna ese valor incrementado en 1 a la altura de *u*.

```
def relabel(u):
    d = min([height[i] for i in range(N) if residual(u,i) > 0])
    height[u] = d+1
```

El método *discharge(u)* utiliza *seen* para recorrer circularmente los vértices de la red, chequeando si *u* puede enviarles flujo, y en caso de llegar al último vértice aún con excedente, se realiza un *relabel(u)*.

```
def discharge(u):
    while (excess[u] > 0):
        if (seen[u] < N):
            v = seen[u]
            if (residual(u,v) > 0 and height[u] > height[v]):
                push(u, v)
            else:
                seen[u] += 1
        else:
            relabel(u)
            seen[u] = 0
```

[6] https://drive.google.com/file/d/11awlbGoXLIGCz21f16WRwnV0r9-3oGMg/view?usp=sharing

Por último, el método *max_flow* inicializa los arreglos correspondientes y envía todo el flujo posible desde el nodo origen a sus adyacentes, luego, mientras existan vértices desbordados los irá descargando. Por último, sabemos que el flujo máximo en este caso fue el que se logró llevar al nodo destino, por lo que su excedente equivale al flujo máximo de la red.

```
def max_flow():
    global height, flow, excess, seen

    height = [(N if i == 0 else 0) for i in range(N)]
    flow = [[0 for i in range(N)] for j in range(N)]
    excess = [0 for i in range(N)]
    seen = [0 for i in range(N)]

    excess[0] = sum(capacities[0])

    for i in range(1,N):
        push(0, i)

    while not overflowed.empty():
        u = overflowed.get()
        if (u != 0 and u != N-1):
            discharge(u)
    return excess[N-1]
```

3.2.6 Resolución de problemas utilizando este algoritmo: POTHOLE

Básicamente, este algoritmo sirve para resolver cualquier problema de flujo máximo que se pueda resolver utilizando Edmonds-Karp. En este caso se realizó una implementación de el problema POTHOLE, ya explicado en la sección de problemas resueltos, esta vez utilizando este algoritmo *push-relabel*, que se encuentra adjunta en el archivo *pothole_push_relabel.py* [7].

[7] https://drive.google.com/file/d/1ti9CzHbi51LuLtRxJh_KMeXOR7yPI4PC/view?usp=sharing

4. Conclusiones

En este trabajo en primer lugar se definieron los elementos de una red de flujo y presentó el problema del flujo máximo. Luego, se mostraron dos soluciones a este problema, que podrían considerarse duales: el método de Ford-Fulkerson (y su implementación, el algoritmo de Edmonds-Karp), que parte de una red con un flujo de 0 y aumenta iterativamente hasta llegar a un flujo máximo, y los algoritmos de tipo *push-relabel*, más específicamente *relabel to front*, que parte de un preflujo mayor o igual al flujo máximo de la red, y a través de las operaciones *push* y *relabel* transforma este en un flujo.

Se mostraron ejemplos de cómo estos algoritmos trabajan en una red particular y se realizó una implementación de cada uno de estos en *python*. Además se utilizó esta implementación para la resolución de los problemas POTHOLE y Road Construction.

5. Archivos adjuntos

1. **[Implementación]** Algoritmo de Edmonds-Karp: https://drive.google.com/open?id=1ntp11BhAsNFnQBkFZK7Yqsm9onREUnuz
2. **[Problema]** POTHOLE: https://www.spoj.com/problems/POTHOLE/
3. **[Solución]** POTHOLE: https://drive.google.com/open?id=1LHYwvxysZAeoA9ifGCR-W4ZkLyRkdAHY
4. **[Problema]** Road Construction: https://codeforces.com/contest/1252/problem/L
5. **[Solución]** Road Construction (C++): https://drive.google.com/open?id=15lhTyj0mk4UurQfgvEhR11cmPBvU-Yzd
6. **[Solución]** Road Construction (Python3): https://drive.google.com/open?id=1n68KD_Ch8bGp2qVT-iVDImlTPzzKUrqH
7. **[Implementación]** Algoritmo *relabel to front*: https://drive.google.com/file/d/11awlbGoXLIGCz21f16WRwnV0r9-3oGMg
8. **[Solución]** POTHOLE (algoritmo *push-relabel*): https://drive.google.com/file/d/1ti9CzHbi51LuLtRxJh_KMeXOR7yPI4PC

6. Fuentes de información

1. Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms, 3rd Edition; 2009
2. https://cp-algorithms.com/graph/edmonds_karp.html
3. https://www.topcoder.com/community/competitive-programming/tutorials/maximum-flow-section-1/
4. https://cp-algorithms.com/graph/push-relabel.html
5. https://www.topcoder.com/community/competitive-programming/tutorials/maximum-flow-augmenting-path-algorithms-comparison/