DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Flujo máximo por método *Push and Relabel*

Agosto 2019 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Cynthia Liberman	443/15	cynthia.lib@gmail.com
María Belén Ticona	143/16	ticona.belu@gmail.com

Docente	Fecha Final
Francisco J. Soulignac	Agosto 2019

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción **4**
- 1.1. Historia . . . 4
- 1.2. Conceptos y nociones básicas de flujo . . . 4

2. Método *push-relabel* **6**
- 2.1. Algoritmo genérico *push-relabel* . . . 6
  - 2.1.1. Aplicación del Teorema Flujo Máximo-Corte Mínimo . . . 7
  - 2.1.2. Correctitud y terminación del algoritmo . . . 8
- 2.2. Mejoras en complejidad . . . 9
  - 2.2.1. Conjunto de vértice activos - $O(n^3)$ . . . 9
  - 2.2.2. Conjunto de vértice activos por *highest label* - $O(n^2\sqrt{m})$ . . . 9
  - 2.2.3. Corte de Árboles Dinámicos . . . 10
  - 2.2.4. Capacidades acotadas o *excess scaling*- $O(nm + n^2 logU)$ . . . 12
  - 2.2.5. Red Compacta . . . 12
- 2.3. Versiones Implementadas . . . 13
  - 2.3.1. Algoritmo *relabel-to-front* . . . 13
  - 2.3.2. Algoritmo *relabel-to-back* . . . 15

3. Conclusiones **15**

4. Apéndice **17**
- 4.1. Implementación del algoritmo *relabel-to-front* . . . 17
- 4.2. Implementación del algoritmo *relabel-to-back* . . . 17

Presentación

El presente trabajo consiste en una presentación del clásico problema de flujo máximo desde el enfoque conocido como *push-relabel*. Este mismo se presenta como Trabajo Final de la materia Algoritmos y Estructuras de Datos III, en el cual se detalla el funcionamiento método, se realiza un análisis de complejidad del algoritmo genérico, así como también se realiza un recorrido de las distintas mejoras conocidas. Por último, se presentan dos implementaciones mejoradas del algoritmo genérico.

1. Introducción

En teoría de grafos uno de los problemas más relevantes es el Problema de Flujo Máximo, o su equivalente, Problema de Corte Mínimo. A lo largo de la materia Algoritmos y Estructuras de Datos III se lo ha encarado por medio del método de caminos de aumento, propuesto por Ford-Fulkerson en 1956 -el cual si bien a priori no es polinomial puede lograrse por medio de ciertas consideraciones-. Fueron Edmonds-Karp quienes introdujeron el concepto del camino de aumento más corto en 1972, proponiendo una versión estrictamente polinomial del algoritmo de Ford-Fulkerson.

El enfoque a desarrollar en este trabajo es el propuesto por Goldberg y Tarjan conocido como *push-relabel*, el cual se basa en el concepto de *preflujo*. Esta alternativa mantiene un preflujo en los ejes el cual se actualiza por medio de operaciones de tipo *push*. Además, introduce el concepto de *relabeling* de los nodos para tener una cota de la distancia de los vértices hacia la fuente y el sumidero. Al tratarse de operaciones locales aplicables a un eje y vértice respectivamente, este enfoque permite una gran flexibilidad usada para poder diseñar diversos algoritmos eficentes.

En este trabajo primero se realiza una presentación del enfoque *push-relabel* al problema de flujo máximo, seguido de los conceptos básicos de flujo. El desarrollo se centra en su metodología, presentando su algoritmo genérico, analizando su complejidad y profundizando sobre la evolución histórica del algoritmo. Finalmente se exhiben dos versiones mejoradas del algoritmo que fueron implementadas como parte del trabajo.

1.1. Historia

El método *push-relabel* se basa en el concepto clave de *preflujo*, introducido por primera vez por el ruso Karsanov en 1973 como una mejora al método de flujo por bloqueo. Esta última permitía encontrar el camino de aumento más corto de manera más eficiente que el algoritmo de Edmonds-Karp, hallando un flujo de bloqueo. El preflujo se lo puede entender como una relajación que permite que el algoritmo cambie el flujo de un solo arco por medio de una operación tipo *push*, en lugar de cambiarlo en un camino de aumento.

Goldberg y Tarjan fueron quienes en 1988 desarrollaron este enfoque del problema basándose en la aplicación de operaciones locales de tipo *push* y *relabel*, las que permiten darle una flexibilidad al algoritmo pudiendo alcanzar buenos tiempos de ejecución en la práctica.

El punto clave de este enfoque es la relajación del principio de *conservación de flujo*, permitiendo que el flujo que ingresa a un nodo pueda superar al flujo de salida del mismo, siendo la diferencia entre estos valores el *exceso* de un nodo. Notemos que intuitivamente es deseable que el flujo sea *pusheado* hacia el vecino más cercano al sumidero, razón por la cual se introduce una noción de distancia conocido como renombre o *relabeling* válido de los nodos, el cual permite saber a donde enviar el flujo.

1.2. Conceptos y nociones básicas de flujo

A continuación se presentan algunos conceptos y nociones básicas de flujo necesarios para comprender lo explicado en este trabajo práctico.

Una red se puede representar mediante un grafo orientado conexo $G = (V, E)$, con $n = |V| \wedge m = |E|$, que tiene dos vértices distinguidos: una fuente s y un sumidero t, con grados de salida y entrada positivos respectivamente. Cada eje $(v, w) \in E$ tiene una capacidad no negativa $c(v, w) \geqslant 0$, por lo que la función de capacidades de la red es una función $c : E \to \mathbb{R}_{\geqslant 0}$.

Decimos que un *flujo factible* f en la red así definida es aquel que cumple con las siguientes leyes:

- Ley de Capacidades: para todo eje su flujo no supera la capacidad asociada al eje, cumpliéndose luego que: $0 \leqslant f(e) \leqslant c(e)\ \forall e \in E$.
- Ley de Conservación: para todos los vértices de la red, excepto para la fuente s y el sumidero t, la suma de los flujos que entran al vértice es igual a la suma de los flujos que salen de él.

$$\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e) \qquad \forall v \in V \backslash \{s, t\} \quad donde,$$

$$In(v) = \{e \in E, e = (w \to v), w \in V\}$$
$$Out(v) = \{e \in E, e = (v \to w), w \in V\}$$

- Ley de Antisimetría: para todo eje vw, el flujo de v a w es igual al opuesto al flujo de w a v. $f(vw) = -f(wv) \forall vw \in E$.

En un problema de *flujo máximo* lo que se desea es determinar la máxima cantidad de flujo que se puede hacer llegar desde la fuente s al sumidero t sin superar las restricciones de capacidad de la red.

Diremos entonces que el valor del flujo F es la suma de los flujos que entran, menos la suma de los flujos que salen del sumidero t.

$$F = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(t)} f(e)$$

Como veremos más adelante será de mucha utilidad la posibilidad de enviar flujo de regreso a s en el método *push-relabel*. De allí que consideremos lo que se conoce como la *red residual*, para la cual se tiene la definición de *capacidad residual* $c_f(v, w)$ de un par de vértices (v, w) como:

$$c_f(v,w) = \begin{cases} c(v,w) - f(v,w) & si \quad (v,w) \in E \\ f(w,v) & si \quad (w,v) \in E \\ 0 & \text{en cualquier otro caso} \end{cases}$$

Luego, diremos que un par de vértices (v, w) es un *eje residual* si $c_f(v, w) > 0$, quedando entonces definido el conjunto de ejes residuales como E_f.

A partir de estas definiciones, dada una red de flujo $G = (V, E)$ y un flujo f, se considera la *red residual* $G_f = (V, E_f)$ donde E_f consiste de ejes con capacidades que representan cuanto se puede variar el flujo en los ejes de G.

$$\forall (v \to w) \in E \begin{cases} (v \to w) \in E_f & si \quad f((v \to w)) < c((v \to w)) \\ (w \to v) \in E_f & si \quad f((v \to w)) > 0 \end{cases}$$

De esta forma siempre que pueda incrementarse el flujo en el eje vw de G, este permanecerá en G_f con una capacidad residual $c_f = c(v, w) - f(v, w)$. Si este valor de c_f resultara nulo, significaría que no es posible aumentar el flujo en esa dirección y por lo tanto el eje $vw \notin E_f$. Pero la red residual G_f podrá además contener ejes que no estén en G, representando cuánto puede decrecer el flujo en un eje. De este modo por cada eje vw en G con flujo positivo, se tiene al eje wv en G_f con una capacidad residual $c_f(w, u) = f(v, w)$. Esto significa que dado un flujo en el eje vw definido en G podemos enviar como máximo esta cantidad de flujo de regreso en la direccción opuesta por el eje wv definido en G_f.

Para el método *push-relabel* también resulta necesario considerar la noción de *distancia* $d_G(v, w)$ desde v a w en G como la mínima cantidad de ejes de un camino de v a w en G, siendo $d_G(v, w) = \infty$ en caso de no existir ese camino.

Para poder demostrar la correctitud del método, se introduce el concepto de *camino de aumento* como un camino simple entre s y t en la red residual G_f.

Por otro lado, resulta importarte tener en cuenta la equivalencia entre el problema de flujo máximo y corte mínimo dado por Ford-Fulkerson en 1962, por lo que resulta necesario comprender el concepto de corte en un red.

Se define un corte $(S, \overline{S})$ en una red $G = (V, E)$ a una partición del conjunto de vértices V, es decir $S \cup \overline{S} = V \wedge S \cap \overline{S} = \varnothing$, tal que $s \in S$ y $t \in \overline{S}$.

La capacidad del corte es la suma de las capacidades de los ejes que salen del corte:

$$c(S, \overline{S}) = \sum_{e \in S\overline{S}} c(e)$$

Teorema 1 (Teorema de Máximo Flujo - Corte Mínimo) *Si f es un flujo en una red $G = (V, E)$ con una fuente s y un sumidero t, luego las siguientes condiciones son equivlentes:*

1. *f es un flujo máximo en G.*
2. *La red residual G_f no contiene caminos de aumento.*
3. *$|f| = c(S, T)$ para algún corte (S, T) de G.*

Donde $T = \overline{S}$

Cuando $|f| = c(S,T)$ se dice que f define un flujo máximo y el corte (S,T) una capacidad mínima.

Una vez definidos los conceptos de flujo necesarios para poder estudiar el método *push-relabel*, definamos formalmente el problema de flujo máximo a resolver:

Input: Un grafo dirigido $G = (V,E)$ con un nodo fuente s y un sumidero t. Capacidades $c(e) \geqslant 0$ para todos los ejes.
Output: Un máximo flujo factible $f(e)$ que satisface:

- restricciones de capacidad: $0 \leqslant f(e) \leqslant c(e)$ en todos los ejes
- conservación de flujo para todos los nodos excepto s y t
- el valor del flujo $f(e)$ esta maximizado respetando todos los flujos factibles

2. Método *push-relabel*

El método *push-relabel* presentado por Goldberg y Tarjan es el que se desarrolla a continuación en base a su publicación de 1988 [1], en el que realizan la generalización y formalización del método para el problema de flujo máximo el cual había sido presentado por primera vez en 1986.

En base a aquella publicación aquí se presentan tanto las características generales de la metodología, así como su algoritmo genérico y el análisis de su complejidad $O(n^2m)$. Además, dado el Teorema 1 Máximo Flujo-Corte Mínimo resulta importarte ver que con esta metodología se puede hallar no solo el valor de flujo máximo, sino también el corte mínimo.

Dada la flexibilidad del método *push-relabel* han surgido distintas variantes que permitieron alcanzar mejores tiempos de ejecución que el algoritmo genérico, entre las cuales aquí se exponen en este trabajo dos versiones $O(n^3)$ junto a sus implementaciones: *relabel-to-front* y *relabel-to-back*. Para facilitar las complejidades se considera $m > n$ de aquí en más.

2.1. Algoritmo genérico *push-relabel*

A diferencia de otros enfoques del problema de flujo máximo, el método *push-relabel* plantea una modificación en las leyes planteadas previamente que se cumplen para un flujo *factible*.

En primer lugar, se considera lo que se denomina *preflujo* como una función evaluada en los reales para los pares de vértices que cumplan con la Ley de Capacidades y Ley de Antisimetría, pero una versión relajada o más débil de la Ley de Conservación de Flujo planteada previamente:

$$\sum_{e \in In(v)} f(e) \geqslant \sum_{e \in Out(v)} f(e) \quad \forall v \in V \backslash \{s\}$$

En otras palabras, esta relajación en la conservación del preflujo permite que para todo vértice distinto a la fuente, el flujo total de entrada sea al menos tan grande como el total de flujo de salida.

El algoritmo *push-relabel* basado en el concepto de preflujo consiste básicamente en examinar aquellos vértices distintos a s y t que posean un flujo en exceso positivo para poder *pushearlo* hacia los vértices que se estiman estar más próximos al sumidero t, haciendo que llegue la mayor cantidad de flujo allí. En caso de que t no sea alcanzable desde un vértice, se *pushea* el exceso de regreso a la fuente s por medio de los vértices que se estiman estar más cercanos a esta.

De esta forma, cuando no quedan vértices con exceso positivo distintos a s y t el preflujo f se vuelve un flujo válido y este resulta ser el flujo máximo.

Notemos que a partir de esta noción del algoritmo surgen dos cuestiones a tener en cuenta antes de analizarlo en mayor profundidad:

1. cómo mover el flujo en exceso entre los vértices
2. cómo estimar la distancia de un vértice cualquiera hacia s o t.

Para la primera cuestión consideremos la definición previa de *capacidad residual* $c_f(vw) = c(v,w) - f(v,w)$. Cuando vw es una arista saturada su capacidad residual es nula, en cambio si no es saturada su capacidad residual es positiva pudiéndose mandar flujo desde v a w. Si v tiene exceso positivo y $c_f(vw) > 0$, entonces se puede mover una cantidad $\delta = min(e(v), c_f(vw))$, incrementando $f(v,w)$ en δ. Observemos que la otra forma en la cual un eje vw tiene capacidad residual positiva se produce cuando wv tiene flujo positivo, en este caso mover el exceso de v disminuye el flujo en wv.

Para la estimación de las distancias se tiene una función de renombramiento o *relabeling* de los vértices con la cual se tiene una cota inferior de la distancia entre un vértice v y el sumidero t en el grafo residual G_f. Se define un *labeling válido* como la función $d : V \to \mathbb{R}^+$ tal que:

- $d(s) = |V|$
- $d(t) = 0$
- $d(v) \leqslant d(w) + 1 \ \forall vw \in E_f$

Observemos ahora que si $d(v) \geqslant n$ entonces $d(v) - n$ es una cota inferior de la distancia entre v y la fuente s. Análogamente se puede ver que si $d(v) < n$, entonces $d(v)$ es una cota inferior de la distancia entre v y t.

Consideremos también la definición de un nodo *activo* $v \in G - \{s, t\}$ con $d(v) < \infty$ y $e(v) > 0$.

El algoritmo genérico inicia con un preflujo f igual a la capacidad en los ejes salientes desde s y cero en el resto de los ejes, tomando un labeling inicial d. A medida que se ejecuta, se van aplicando operaciones básicas de tipo *push* o *relabel* en los vértices activos, hasta que no quede ninguno de ellos.

La operación de *push* desde v a w modifica el preflujo aumentando el valor de $f(v, w)$ y $e(w)$ por un $\delta = min(e(v), c_f(v, w))$, mientras que disminuye $f(w, v)$ y $e(v)$ por la misma cantidad de preflujo. Decimos que un *push* es *saturante* cuando luego de aplicada la operación, se tiene que $c_f(vw) = 0$, de lo contrario se lo denomina *no saturante*. Un operación *push* solo es aplicable en nodo v hacia w cuando $c_f > 0$ y $d(v) = d(w) + 1$.

Por otro lado, las operaciones *relabel* en un nodo v actualizan su valor por el valor máximo permitido por un *labeling* válido. Solo se aplican sobre cuando v es activo y para todo vecino w que cumple $c_f(v, w) > 0$ se tiene $d(v) \leqslant d(w)$.

A continuación se presentan los pseudocódigos del algoritmo genérico para el cálculo de flujo máximo, junto al correspondiente a las operaciones básicas *push* y *relabel* y la inicialización del preflujo.

```
Push-Relabel(G,s,t,c):
    Mientras que exista una operacion que aplicar
    Elegir la operacion y realizarla
    Retornar f

Push(v,w):
    Enviar δ = min(e(v), c_f(v,w)) unidades de flujo de v a w haciendo
        f(v,w) = f(v,w) + δ,  f(w,v) = f(w,v) − δ
        e(v) = e(v) − δ,  e(w) = e(w) + δ

Relabel(v):
    Aumentar la altura de v como d(v) = 1 + min(d(w) : (v,w) ∈ E_f)

Init-Preflow(G,c):
    ∀e ∈ E saliente de la fuente, f(e) = c(e). Para cualquier otro, f(e) = 0
    Iniciar label d(s) = n y ∀v ∈ V − {s} hacer
        d(v) = 0 y e(s) = e(s) − c(s,v)
        Si v es adyacente a la fuente, e(v) = c(s,v) sino e(v) = 0
```

2.1.1. Aplicación del Teorema Flujo Máximo-Corte Mínimo

Recordemos que el método *push-relabel* permite resolver tanto el problema de flujo máximo como el problema de corte mínimo, de acuerdo al Teorema 1 de Ford-Fulkerson.

Para conocer el flujo máximo, se redefine como vértices activos a todo vértice $v \in V \backslash \{s, t\}$ tal que $e(v) > 0 \wedge d(v) < n$. De este modo no es necesario devolver el exceso que no pudo llegar a t hacia la fuente s, ya que con esta redefinición un vértice con $d(v) \geqslant d(s)$ es considerado inactivo.

Cuando el algoritmo termina, el exceso $e(t)$ en el sumidero es el valor del flujo máximo, y el corte $(S, \overline{S})$ tal que $\overline{S}$ contiene exactamente aquellos vértices desde los cuales t es alcanzable en G_f es un corte mínimo.

2.1.2. Correctitud y terminación del algoritmo

- **Correctitud**

La correctitud del algoritmo se puede analizar, suponiendo que el mismo termina, de forma análoga a la demostración de Ford-Fulkerson a través del concepto de camino de aumento.

Teorema 2 *Un flujo f es máximo si y solo si no hay un camino de aumento, es decir, t no es alcanzable desde s en G_f.*

En primer lugar, veamos que dado un *labeling* válido, las operaciones básicas no lo rompen. Sin ir más lejos, considerando una operación *push* de v a w, esta podría agregar el eje wv o eliminar vw en G_f. Como el *push* ocurre cumpliéndose $d(w) = d(v) - 1$, la adición del eje wv no altera el *labeling* válido, así como tampoco lo hace la eliminación de una arista.

Lema 1 *El algoritmo mantiene el invariante de que d es un labeling válido.*

Notemos que si el algoritmo *push-relabel* termina y todos los *labels* de la distancias son finitos, luego no quedan vértices activos por examinar, por lo que todos tienen exceso nulo y f se torna un flujo válido (cumpliéndose las tres leyes mencionadas previamente).

Del siguiente lema se puede entender que siempre que se tenga un preflujo f y d un *labeling* válido, entonces en particular no se tiene un camino de aumento en G_f.

Lema 2 *Si f es un preflujo y d un labeling válido para f, luego el sumidero t no es alcanzable desde la fuente s en la red residual G_f.*

A partir de estas observaciones se tiene que al finalizar el algoritmo *push-relabel*, f se vuelve un flujo válido y además el *labeling* es válido por ser invariante del algoritmo, luego no se tiene un camino de aumento en G_f por el Lema 2 y por lo tanto, f resulta ser un flujo máximo por el Teorema2.

- **Terminación**

Para poder comprender la terminación del algoritmo, se considera el caracter creciente del *relabeling* y que el mismo se encuentra acotado para todos los vértices. La cantidad de operaciones *push* también presenta una cota superior, por lo que al cabo de cierta cantidad finita de operaciones básicas el algoritmo termina eventualmente.

En primer lugar consideremos la operación *relabel* comenzando por la intuición del siguiente lema:

Lema 3 *Si f es un preflujo y v es un vértice con exceso positivo, luego la fuente s es alcanzable desde v en la red residual G_f.*

Observemos que dado un vértice v con exceso positivo, luego el preflujo recibido tuvo que haber llegado desde la fuente s a través de una serie de ejes con flujo entrante positivo. Para cada uno de estos ejes le corresponde un eje en sentido opuesto con capacidad residual positiva en la red, ergo v está conectado con s habiendo un camino entre ellos.

Ahora notemos que el *relabeling* es creciente y además el valor de *labeling* de cada vértice se encuentra acotado por $O(V)$.

Lema 4 *Dado un vértice v, la función de distancia $d(v)$ nunca decrece. Una aplicación de la operación relabel sobre v incrementa $d(v)$.*

Lema 5 *En cualquier momento de la ejecución del algoritmo y para cualquier vértice $v \in V$, se cumple que $d(v) \leqslant 2n - 1$.*

Como la operación de *relabel* afecta solamente a los nodos activos, el Lema 5 vale para los vértices no activos trivialmente. Los activos por definición tienen exceso positivo. Luego, dado un vértice activo v en G, por el Lema 3, existe un camino P de v hacia s en G_f puesto que la fuente es alcanzable desde v. Dicho camino de longitud l no puede ser más largo que $n - 1$ y además por ser un *labeling* válido se tiene que dado un $(v_j, v_{j+1}) \in P$, se cumple $d(v_j) \leqslant d(v_{j+1}) + 1$. Por lo tanto para un nodo activo v se cumple que $d(v) \leqslant d(v_l) + l \leqslant d(s) + (n - 1) = n + (n - 1)$.

Lema 6 *El número de operaciones de relabel es a lo sumo $2n-1$ por vértice y a lo sumo $(2n-1)(n-2) < 2n^2$ en total.*

Ahora bien solo resta analizar las cotas superiores para las operaciones de *push* saturantes y no saturantes.

Dada una operación de *push saturante* desde v a w, el eje desaparece de G_f, por lo que no puede enviar más flujo en ese mismo sentido, a menos que se retorne desde w a v. Pero para que esto ocurra, primero debe haber un incremento de distancia de al menos dos unidades en w. En caso de ocurrir, de manera análoga se requerirá que la distancia de v aumente en al menos unidades para cambiar su *label*. De esta forma por cada dos *push saturantes* se tiene que $\Delta(d(v) + d(w)) \geqslant 4$, luego por la Lema 5 se cumple que $d(v) + d(w) \leqslant 4n - 3$ en el último *push* e inicialmente $d(v) + d(w) \geqslant 1$. La diferencia de la suma de los labels dividido por 2 da la cantidad total de *push* saturantes para los dos nodos menos uno, el push inicial. De modo que se tienen $2n - 1$ *push saturantes* por cada eje. Luego se entiende que valga el siguiente lema:

Lema 7 *El número de operaciones de push con saturación es a lo sumo* $2nm$.

En cuanto a los *push no saturantes*, para comprender la idea establezcamos una analogía definiendo la energía del flujo como la sumatoria de las distancias de los vértices activos. Como tanto al iniciar como al finalizar el algoritmo no hay nodos activos puesto que ninguno tiene exceso, luego las operaciones que aumentan y disminuyen la energía deben compensarse a lo largo de la ejecución.

Notemos que por cada *push no saturante*, un nodo queda inactivo puesto que si tuviese aún exceso, este lo hubiera *pusheado*. De modo que la energía disminuye en al menos una unidad. En cambio un push saturante aumenta la energía dado que puede llegar a tener aún exceso por enviar, pudiendo alcanzar altura $2n - 1$.

Finalmente, como solo el *push no saturante* decrementa la energía luego tenemos que debe compensar la energía aportada por la cantidad de *relabels* y *push saturantes*, luego $\#pushSaturantes * d_{max} + \#relabels_{max} = 2nm(2n - 1) + (2n - 1)(n - 2)$.

Lema 8 *El número de operaciones de push sin saturación es a lo sumo* $4n^2m$.

Finalmente se tiene que el algoritmo genérico *push-relabel* tiene una complejidad de $O(n^2m)$.

Teorema 3 *El algoritmo genérico push-relabel termina luego de* $O(n^2m)$ *operaciones básicas.*

Resulta importante remarcar que según la elección del orden en que se apliquen las operaciones básicas, el algoritmo puede alcanzar tiempos de ejecución mejores.

2.2. Mejoras en complejidad

Dado el carácter flexible del método *push-relabel*, el orden en que se aplican las operaciones básicas variando la estructura de representación elegida permite obtener algoritmos más eficientes que la versión genérica presentada anteriormente.

A continuación se detallan dos enfoques de mejoras en complejidad que se basan en considerar el conjunto de vértices activos para saber qué operación aplicar.

2.2.1. Conjunto de vértice activos - $O(n^3)$

La idea consiste en una vez elegido un vértice activo, aplicarle tantas operaciones de *push* como sea posible. La versión genérica del algoritmo en cambio recorre los ejes, pasando de un vértice a otro entre cada operación, repitiendo el procedimiento por cada paso. Esta versión del algoritmo consiste entonces en mantener un conjunto de todos los vértices activos, tomando uno por uno para poder remover todo el exceso posible mediante la red residual y reponiéndolo en el conjunto solo si aún queda exceso remanente. Su complejidad de esta forma resulta ser $O(n^3)$.

De esta mejora se presentan en este trabajo dos implementaciones llamadas *relabel-to-front* y *relabel-to-back* en donde el conjunto de vértices activos se representa por medio de una *linked-list* y un cola *FIFO* respectivamente (véase posteriormente).

2.2.2. Conjunto de vértice activos por *highest label* - $O(n^2\sqrt{m})$

Esta versión introducida por Cheriyan-Maheshwari en 1989 aplica operaciones *push-relabel* de forma análoga a la anterior, aunque se diferencia en además mantener un orden en particular en el cual se recorren los vértices activos. El vértice siguiente a procesar v se elige como aquel que maximice $d(v)$, siendo de este modo su complejidad $O(n^2\sqrt{m})$.

Pensemos que procesar el exceso de un vértice altera el exceso de todos sus vecinos que se encuentren a menor $d(v)$, los cuales si ya fueron previamente procesados deberán ser procesados nuevamente. De modo que esta versión del algoritmo busca disminuir el re-procesamiento de elementos del conjunto de vértices activos.

Las mejoras recién mencionadas se basan en el *caracter local* de la aplicación del las operaciones *push/relabel* en un solo vértice, a diferencia del siguiente método a desarrollar que analiza operaciones *push* aplicables a un camino unidireccional de nodos. La ventaja de esta idea es que se puede acotar el costo de esta operación en función de la estructura de datos de representación.

A continuación se detalla una de las versiones más rápidas del algoritmo *push-relabel* la cual emplea árboles dinámicos, estructura introducida por Sleator-Tarjan en 1985 [7].

2.2.3. Corte de Árboles Dinámicos

Con el objetivo de reducir la cota de complejidad que aportan las operaciones de *push no saturante*, se introduce como estructura el árbol dinámico.

- **Estructura: Árbol Dinámico**

Esta estructura de datos permite manetener un conjunto de vértices disjuntos, dado que inicialmente todos los vértices son la raíz de un árbol y no tienen hijos. Los ejes del árbol son dirigidos hacia la raíz, de hijo a padre. Se denota $p(v)$ al padre de un vértice v. A cada vértice se le asocia un valor real $g(v)$, que en principio adopta el valor ∞.

Se describen a continuación las operaciones que tiene asociada la estructura para su utilización en este método:

- *find_ root(v)*: Encuentra y retorna el valor de la raíz del árbol que contiene al vértice v.
- *find_ size(v)*: Retorna la cantidad total de vértices del árbol en el que se encuentra v.
- *find_ value(v)*: Retorna el valor $g(v)$.
- *find_ min(v)*: Encuentra y retorna el ancestro w de v con mínimo valor $g(w)$. En caso de empate se elige el vértice w más cercano a la raíz.
- *add_ value(v, x)*: Le suma el valor real x a $g(w)$ para cada ancestro w de v.(se adopta la convención $\infty + (-\infty) = 0$).
- *link(v, w)*: Combina los árboles que contienen al vértice v y w transformando a w en padre de v. Esta operación no hace nada si, v y w están en el mismo árbol o si v no es raíz de su árbol.
- *cut(v)*: Separa el árbol que contiene a v en dos árboles, borrando el eje entre v y su padre. Esta operación no hace nada si v es la raíz.

- **La red admisible y la cota de complejidad**

Los ejes presentes en el árbol son *ejes admisibles*, por definición un eje admisible cumple que $d(v) = d(w) + 1 \wedge c_f(v, w) > 0$ con $p(v) = w$. Se destaca aquí que los vértices hijos tienen un valor de *labeling* mayor que el de su padre. Por lo tanto la raíz es el vértice de menor labeling en un árbol dado. Cada vértice contiene el valor $g(v) = c_f(v, p(v))$ si v tiene padre, y el valor ∞ si v es la raíz del árbol. Se asume además que el valor del exceso se encuentra contenido en el vértice.
Pero el árbol no representa a la red admisible en su totalidad, algunos ejes admisibles podrían no estar representados en el mismo. Esto se debe a que se limita el máximo tamaño del árbol a un valor k.
Esta cota a la cantidad de vértices en un árbol es esencial para la complejidad del algorítmo, y es en torno a esto que se adaptan todas las operaciones del método propuesto.

Una secuencia de l operaciones sobre un árbol, comenzando por un bosque de árboles de único vértice raíz, tienen una cota temporal $O(l \cdot logk)$.

Por lo dicho para mantener la cota de complejidad es necesario que los árboles no tengan más de k vértices, por lo que no es posible unir dos árboles mediante un eje admisible que juntos sumen más de k vértices. Esto hace que sea necesario mantener en una estructura auxiliar el valor de flujo de aquellos ejes admisibles que no se encuentran en el árbol y que por lo tanto resultan ïnvisibles". Para lo mismo proponen utilizar la lista de vecinos de la red $G = (V, E)$, guardando allí no solo el vecino sino también el flujo, entre el nodo en cuestión y su vecino.

- **El procedimiento *send(v)*, el corazón del algoritmo**

Utilizando las operaciones adecuadas es posible empujar flujo, desde un vértice hacia sus ascendentes, ya sea delimitado en algún eje por un *push* saturante (cuello de botella del camino) o moviendo el exceso a lo largo de todo el camino hasta la raíz del árbol.

El procedimiento *send(v)* aplica solamente para vértices activos que no son raíz, empuja su exceso hacia la raíz de su árbol, cortando los ejes que fueron saturados por el *push*, repitiendo estos pasos hasta que $e(v) = 0$ o v sea la raíz. El corte se realiza en aquellos ejes que al estar saturados tienen un valor $g(v) = 0$ (valor mínimo posible), recordar que por lo definido en la función *find_min(v)*, si hay varios ejes a lo largo del camino con el mismo valor primeramente se accederá al que está más cerca de la raíz. De este modo, los cortes se aplican de arriba hacia abajo en el árbol.

A continuación se define el procedimiento *send(v)*:

```
Send(v):
    Mientras find_root(v) ≠ v ∧ e(v) > 0
        Enviar δ = min(e(v), find_value(find_min(v))) unidades de flujo por el camino en el
        arbol desde v aplicando add_value(v, −δ)
        Mientras find_value(find_min(v)) = 0 hacer
            u = find_min(v) y aplicar cut(u) seguido de add_value(u, ∞)
```

- **Algoritmo *push-relabel* con árbol dinámico**

En esencia el algoritmo funciona de manera similar al de ordenamiento *First-In First-Out*: se mantiene una cola Q de nodos activos que son procesados para descargar su exceso hasta que la cola esté vacía. La variación en este caso es la manera en la que se realizan las acciones de *push* y *relabel*, mediante la operación *Tree-Push-Relabel*. Esta operacíon aplica a un **vértice activo** v que es **raíz** de un árbol dinámico.

Dado un eje vw pueden ocurrir dos casos:

1. El eje vw es admisible y:

   I . Los árboles que contienen a v y a w suman a lo sumo un total de k vértices, la operación *tree_push_relabel* une estos dos árboles transformando a w en el padre de v, y luego efectuando un *send* desde v.

   II . Los árboles que contienen a v y a w suman más de k vértices. Y he aquí una genialidad del método, en este caso la operación *Tree-Push-Relabel* hace un *push* tradicional sobre la *arista invisible*, a modo de "arrojar" el exceso de un árbol hacia el otro, y luego continúa aplicando un *send* pero ahora desde el vértice w en el otro árbol.

2. El eje vw no es admisible, pero $c_f(v, w) > 0$. En este caso se actualiza el vecino y se aplica la operación de *relabel* si fuera necesario. Si se hace un *relabel* a v, se aplica un corte a todos los ejes entrantes a v, dado que ahora estos ejes ya no serán admisibles.

- *Correctitud*

Una cuestión importante a tener en cuenta es que la raíz de un árbol dinámico tal y como fue definido no tiene ejes admisibles salientes. Como se mencionó anteriormente dado un vértice v raíz, $d(v)$ tiene el mínimo valor de entre los vértices de un mismo árbol, y por lo tanto no pueden existir ejes admisibles entre v y otros vértices de su propio árbol. Esto es significativo para la correctitud del algoritmo; dado que si se tiene como precondición para la operación *Tree-Push-Relabel* que v es raíz, esto asegura que siempre que se estén uniendo dos vértices por medio de un eje admisible con la operación $link(v, w)$, v y w pertenezcan a árboles distintos. Un segundo elemento necesario para la correctitud del algoritmo, es que se asegura que los nodos activos son siempre nodos raíz.

Se expone aquí la operación *tree_push_relabel* descripta:

```
#precondición: v es un nodo raiz, activo
Tree-Push-Relabel(v):
    Sea w el vecino actual de v
    Si d(v) = d(w) + 1 ∧ c_f(v, w) > 0
        Si find_size(v) + find_size(w) ⩽ k
            transformar a w en el padre de v efectuando
```

```
            add_value(v, −∞), add_value(v, c_f(v, w)), link(v, w)
            empujar el exceso de v a w aplicando send(v)
        sino
            hacer un push(v, w) para mover el exceso de v a w
            aplicar send(w)
    sino
        si w no es el ultimo vecino en la lista de v
            avanzar al proximo vecino en la lista
        sino
            poner como actual al primer vecino en la lista de v
            Aplicar cut(u), add_value(u, ∞) a cada hijo u de v
            aplicar relabel(v)
```

- *Complejidad*

La cota para la cantidad de veces que un vértice es agregado a la cola Q es $O(nm + n^3/k)$ [1]. Con un costo de $O(logk)$ por cada operación en el árbol, da un tiempo total $O((nm+n^3/k)logk)$ para la ejecución del algoritmo, que al elegir el valor de $k = n^2/m$ se minimiza a $O(nmlog(n^2/m))$.

La siguiente mejora en complejidad a tratar se basa ya no en establecer una cota a las operaciones de *push* sino más bien a la capacidad de los ejes de la red.

2.2.4. Capacidades acotadas o *excess scaling-* $O(nm + n^2logU)$

Introducido por Ahuja-Orlin en 1987 [5], esta versión propone acotar la complejidad del algoritmo en base al valor U, cota de las capacidades de la red las cuales deben ser enteras para que valga la cota.

La idea que respalda esta cota de complejidad surge de definir un parámetro $\Delta = 2^{[logU]}$ inicialmente, el cual representa una cota superior para el exceso de los vértices activos. La ejecución del algoritmo se realiza por fases, donde después de cada una Δ se ve disminuido, finalizando la ejecución cuando $\Delta < 1$, es decir, cuando no quedan vértices por procesar. De esta forma la terminación ocurre al cabo de $logU + 1$ fases.

Para poder mantener el invariante de que ningún exceso supere la cota establecida, se debe limitar la cantidad de preflujo que envía una operación *push*. De allí que el procesamiento de los vértices elija aquel con mayor cantidad de exceso y menor *label*, así se descarga primero el flujo de los vértices más cercanos al sumidero. Notemos que si se eligiera un vértice con máximo exceso pero no de menor *label* posible, este podría potencialmente mandarle flujo a un vértice que tenga igual cantidad de exceso, pudiendo superar la cota.

Esta idea fue re-elaborado en colaboración con Tarjan [6] ese mismo año, proponiendo una versión más sofisticada con complejidad $O(nm + n^2\sqrt{logU})$ la cual combina mantener el conjunto de vértices en un *stack* junto con una idea previamente usada para el método de bloqueo de flujo (véase Introducción).

Las mejoras hasta aquí desarroladas trabajan directamente con la red residual, sin crear ejes ni vértices. La siguiente mejora se basa en tranformar la red de trabajo agregando ejes y nodos de forma tal de generar una *estructura* global más compacta.

2.2.5. Red Compacta

La primer idea que propuso modifcar la red residual fue propuesta por Cheriyan-Hagerup en 1990 [8] en el que definen una extensión del algoritmo genérico del método *push-relabel* llamado *algoritmo genérico incremental*, agregando la operación *add edge*. Este algoritmo manipula el preflujo en la subred residual, agregando gradualmente los ejes remanentes.

La ejecución del algoritmo original que proponen aplica operaciones de *push-relabel* y *add-edge* hasta que ya no haya ninguna aplicable. Análogamente a las mejoras previamente desarrolladas, el orden en que se aplican las operaciones permite alcanzar mejoras en complejidad del algoritmo original.

La primera propuesta desarrollada por Cheriyan fue designar a cada nodo un *current edge* o eje actual, siendo este el único mediante el cual cada vértice puede mandar flujo en la red. Su modificación solo se realiza entonces cuando ya deja de ser elegible el eje. No obstante, se obtuvieron mejores resultados haciendo que esta elección sea aleatoria, brindándole un carácter no determinístico al algoritmo. La novedad del trabajo, además de la incorporación de la nueva operación, fue que se propuso que la elección

del *current edge* de cada nodo quede determinada por una estrategia de juego, idea distinta al enfoque desarrolado hasta ese entonces en el método *push-relabel*.

Fueron King y Rao quienes en 1992 propusieron una estrategia de juego completamente determinística [9] mediante la cual lograron desarrollar un algoritmo *push-relabel* de complejidad $O(mn(log_{m/nlogn}n))$. Para cuando $m/n = \theta(n^{\epsilon})$ para toda constante positiva ϵ, la complejidad se reduce al costo lineal $O(mn)$.

Finalmente, al cabo de varias metodologías y estrategias de juego desarrolladas, en 2013 Orlin [10] logró la implementación de un algoritmo lineal para cualquier valor de m y n, siendo este el algoritmo con mejor complejidad conocido hasta ahora. Existen casos especiales para los cuales se conocen complejidades aún mejores como lo es cuando $m = O(n)$ (grafos ralos), alcanzando ser $O(n^2/logn)$.

La modificación de Orlin propone la creación de una red compacta, en la cual considera inicialmente los ejes con capacidad media aproximada y descarta todos los ejes restantes. Para preservar la estructura de la red decide construir *pseudoejes* los cuales vienen a reemplazar a varios ejes con capacidad chica, quedando completamente eliminados aquellos ejes con capacidad muy alta. Además, elimina aquellos vértices para los cuales hay una abundante cantidad de ejes incidentes,

2.3. Versiones Implementadas

A continuación se presentan dos versiones del método *push-relabel* en base a la mejora que mantiene un conjunto de vértices activos, siendo su complejidad $O(V^3)$. Los algoritmos implementados que emplean una lista enlazada y una cola *First-In First-Out* para el conjunto respectivamente son: *relabel-to-front*[2] y *relabel-to-back*. En ambos casos los ejes de la red $G = (V, E)$ se organizan en una lista de adyacencias. De modo que, el vértice w aparece en la lista de v si (v, w) o $(w, v) \in E$.

2.3.1. Algoritmo *relabel-to-front*

El algoritmo *relabel-to-front* aquí presentado se implementa en base a una estructura de lista L que contiene todos los vértices de la red, a excepción de s y t.

El algoritmo realiza un recorrido de la lista L, iniciando desde el primer elemento hasta encontrar un vértice v con exceso para luego realizar una «descarga» o *discharge* del flujo. Por medio de esta acción ejecuta operaciones *push/relabel* hasta que v no contenga más exceso. Cuando el vértice ha sido re-etiquetado al menos una vez al ejecutar la función *discharge* se lo pone al frente de la lista, re-comenzando desde allí el recorrido de la lista.

Resulta importante mencionar que en lugar de tener la función de distancia propia del algoritmo genérico del método *push-relabel*, en esta implementación se trabaja con la función de altura la cual no es más que una interpretación en particular que preserva las características previamente desarrolladas para la función de distancia. Para mantener la fidelidad a la bibliografía [2], se mantiene la nomenclatura de las funciones y operaciones.

- **Ejes admisibles**

Un concepto primordial en la correctitud y análisis de este algorítmo es la noción de *ejes admisibiles*, estos son aquellos ejes de la red residual a través de los cuales se puede enviar flujo. Sea $G = (V, E)$ una red de flujo con fuente s y sumidero t, f un preflujo en G, y h la función de altura, luego (v, w) es un eje admisible si $c_f(v, w) > 0$ y $h(v) = h(w) + 1$. De lo contrario (v, w) es inadmisible. La red admisible es $G_{f,h} = (V, E_{f,h})$, donde $E_{f,h}$ es el conjunto de ejes admisibles.

Lema 9 *Si $G = (V, E)$ es una red de flujo, f un preflujo en G, y h la función de altura en G, luego* ***la red admisible*** *$G_{f,h} = (V, E_{f,h})$* ***es acíclica****.*

- **Pseudocódigo *relabel-to-front***

En primer lugar, se inicializa el preflujo de la red y se agregan todos vértices a la lista L, excepto la fuente y el sumidero los cuales no son procesados. Se considera por cada vértice un *vecino actual* con el que se recorre el vecindario de cada nodo en búsqueda del eje admisible por el que se pueda o bien mandar el exceso hallado o bien actualizar el label del nodo, dichas operaciones en conjunto se denominan *discharge*.

En caso de que un *discharge* haya producido *relabel*, el vértice en cuestión se mueve al frente de la lista L, re-comenzando desde allí el recorrido del algoritmo. De esta forma la lista L no se recorre una única vez, además sus elementos van adquieriendo a lo largo de la ejecución una posición distinta a la inicial.

A continuación se presenta el pseudocódigo del algoritmo, en el Apéndice A se presenta su implementación correspondiente.

```
Relabel-to-Front(G,s,t)
    Inicializar el preflujo de G
    Crear la lista L con v ∈ L ∀v ∈ V(G) − {s,t}
    Por cada nodo v definir actual(v,N) = N[v][0], primer vecino en su lista de adyacencia
    Recorrer L desde el inicio hasta finalizar, por cada nodo v hacer:
        discharge(v)
        Si hubo un relabel de v, agregarlo al inicio y reiniciar el ciclo
```

El invariante que se mantiene durante la ejecución del algoritmo es que todos los vértices ya recorridos de la lista no tienen exceso, por cada uno de los nodos desde el actual se cumple además que no presenta ejes admisibles con sus antecesores en la lista.

Dicho invariante garantiza que si no hay re-etiquetado al descargar a v, el exceso de v (en caso de tenerlo) no se habrá mandado a ninguno de sus antecesores en la lista L. Es así que v mantiene su posición en la lista, ahora con exceso nulo, y que se puede seguir iterando al próximo vértice.

El algoritmo termina cuando se ha recorrido toda la lista L. El invariante asegura que todos los vértices $v \in L$ tienen exceso nulo, obteniéndose el máximo flujo en la red.

- ***Descarga* de un vértice con exceso**

Un vértice v con exceso positivo es descargado del exceso de flujo mediante la operación *push* a través de ejes admisibles hacia sus vértices vecinos, si no hubiera ejes admisibles es necesario re-etiquetar a v para que sus ejes salientes se hagan admisibles, recorriendo nuevamente los vecinos de v. El algoritmo termina cuando el exceso de v es nulo. Este comportamiento es el que se describe en el siguiente pseudocódigo.

```
Discharge(v)
    Mientras v tenga exceso positivo, considerar w = actual(v,N)
        Si no quedan vecinos en el vecindario, hacer relabel(v) y actual(v,N) = N[v][0]
        De lo contrario, si c_f(v,w) > 0 ∧ h(v) = h(w) + 1 hacer push(v,w)
        Sino continuar con el siguiente vecino: actual(v,N) = siguiente(actual(v,N))
```

Notemos que al salir de la ejecución el *vecino actual* del nodo al cual se aplicó *discharge*, queda inmodificado hasta la próxima vez que se llame a *discharge* para ese nodo.

- ***Complejidad***

Dado que el algoritmo *relabel-to-front* llama a la función *discharge*, analicemos primero la complejidad de esta operación. Cada iteración ejecutada en la descarga toma una de las siguientes acciones:

- *relabel*: el algoritmo *push-relabel* genérico da una cota de $O(V)$ a la cantidad de re-etiquetados ejecutados para un mismo vértice, y por lo tanto una cota $O(V^2)$ para todos los vértices en total (no si considera s ni t). La cota temporal para realizar los $O(V^2)$ reetiquetados esta dada por $O(VE)$.
- *recorrer vecinos:* Se actualiza $actual(v, N)$ tantas veces como cantidad de vecinos tenga el vértice v, lo que da una cota $O(grado(v))$ por cada reetiquetado a v, dado que a un vértice se lo reetiqueta a lo sumo $|V|$ veces, la cota es $O(V \cdot grado(v))$. Como $grado(v) \leqslant |V| - 1$, recorrer vecinos tiene una cota $O(V^2)$.
- *push*: Se realiza un total de $O(VE)$ *push* con saturación. Mientras que para el caso de *push* no saturante, el exceso ha sido reducido a cero, saliendo de la función de descarga.

Teniendo en cuenta el análisis de complejidad expuesto, el tiempo de ejecución de la descarga esta acotada por $O(VE)$. Resta ahora analizar la complejidad temporal del algoritmo *push-to-front*.

Cuando se re-etiqueta un vértice este pasa a estar al comienzo de la lista L de longitud $|V| - 2$, recomenzando el recorrido de la lista. Podemos pensar que una fase es lo que sucede entre dos re-etiquetados sucesivos (aunque podría ocurrir que haya más de un re-etiquetado dentro de la función de descarga para liberar el total del exceso, dentro de la misma fase). La cota de re-etiquetados es la dada anteriormente $O(V^2)$, dando así mismo una cota para la cantidad máxima posible de fases.

En cada fase se recorre a lo sumo la totalidad de la lista L, haciendo por lo tanto $O(V)$ llamados a la función de descarga en una sola fase. Esto significa que la función de descarga es llamada un total de $O(V^3)$ veces, que es también la cota de la cantidad de *push* sin saturación efectuados en la descarga. La complejidad total del algoritmo *relabel-to-front* resulta así $O(V^3 + VE) = O(V^3)$.

2.3.2. Algoritmo *relabel-to-back*

En esta versión del algorítmo *push-relabel* los vértices con exceso se procesan en el orden FIFO (*First-In First-Out*). Para ello se utiliza como estructura una cola Q donde se irán agregando únicamente aquellos vértices con exceso, es decir, los nodos activos. Inicialmente se agregan a la cola los vecinos de la fuente s, ya que al inicializar el preflujo en la red se hace un *push* saturante de la fuente s a sus vecinos, pasando estos de inactivos a activos. Posteriormente se irán agregando nuevos nodos activos en el orden en que se fueron activando, siendo este mismo el orden en que se procesarán.

- **Procesamiento de un nodo activo**

Procesar un nodo activo involucra una serie de acciones: sea un nodo activo v en el tope de la cola Q, se lo desencola y luego se hace un *push* a sus vecinos admisibles, estos vecinos que ahora tienen exceso y por ende son activos, son encolados en Q en orden de activación. Una vez recorridos todos los vecinos, si v aún es activo se lo re-etiqueta y encola nuevamente en Q para su procesamiento en la siguiente fase junto con sus vecinos activados. En este caso todos los *push* habrán sido saturantes. Sin embargo, si hubiere un *push* sin saturación o en su defecto un *push* saturado que anule el exceso en v, este se torna inactivo y luego no es necesario reetiquetarlo ni encolarlo en Q. Cabe aclarar el concepto de *vecinos admisibles*, estos son aquellos vecinos que están unidos a un nodo por *ejes admisibles*.

- **Fases del algoritmo**

En primera instancia se encolan en Q los vecinos de s, a estos se los denomina nodos de la fase 1. Todos los nodos que ingresan en la cola al procesar estos nodos, son los nodos de la fase 2. Generalizando, procesar los nodos de la fase i, hace que se agreguen nodos en la cola y estos serán los nodos de la fase $i+1$. Los nodos una vez procesados no pertenecen a una única fase, podrían estar incluidos en la fase $i+1$ nodos que estaban en la fase i, dado que se vuelven a encolar si aún están activos.

- **Pseudocódigo *relabel-to-back***

Luego de realizado el análisis de la estructura y comportamiento del algoritmo se presenta aquí su pseudocódigo (véase en mayor detalle la implementación en el Apéndice B). Posteriormente se analiza su complejidad.

```
Relabel-to-Back(G,s,t)
    Inicializar el preflujo en G
    Para todo vecino de s, agregarlo a la cola Q
    ∀v ∈ V(G) − {s}, definir
        h(v) = #ejes en el camino mas corto de v a t

    Mientras que Q ≠ ∅, sacar el nodo v tope y hacer:
        #descargar su exceso a todos sus vecinos admisibles
        Mientras e(v) > 0 ∧ ∃w vecino admisible:
            hacer push(v,w) y agregar a w a Q si w ≠ s,t ∧ w ∉ Q
        Si e(v) > 0 hacer relabel(v) y agregarlo a Q
```

Complejidad: Durante la ejecución del algoritmo la cantidad de máxima de fases es $4n^2$ [1], si consideramos que hay a lo sumo n nodos por fase, la complejidad del algoritmo resulta $O(n(2n^2+n)) = O(n^3)$.

3. Conclusiones

En este trabajo se ha realizado un *review* de la historia del método *push-relabel* como forma de resolución del problema de flujo máximo en grafos. Como bien se ha detallado, surge de flexibilizar la Ley de Conservación de Flujo trabajando con un preflujo a lo largo de la ejecución del algoritmo.

Tanto este método como el de camino de aumento visto en la materia, buscan que al finalizar el algoritmo no haya un camino de aumento entre la fuente s y el sumidero t, y se tenga un flujo válido. Se puede ver cierta simetría entre ambos al ver que el método de camino de aumento trabaja con un flujo válido distribuyéndolo en los ejes de acuerdo a las leyes, mientras que *push-relabel* mantiene como

invariante que no haya un camino de aumento hasta la finalización del algoritmo donde el preflujo se torna finalmente un flujo válido.

Además, la definición de operaciones locales *push/relabel* le otorgan una gran flexibilidad al algoritmo que permitieron que se puedan diseñar diversas mejoras en complejidad basadas en distintas ideas. El primer caso fue a través del orden de las operaciones básicas, seguido de establecer una cota para el costo de las operaciones según la estructura de representación usada. Otra idea fue incluso más abarcativa y propuso la extensión de las operaciones básicas, siendo este el origen del mejor algoritmo diseñado dentro del método *push-relabel* mediante construcción de una red auxiliar más compacta que la residual (eliminando nodos y redefiniendo ejes).

A modo de conclusión, se deja como reflexión que los conocimientos adquiridos en la materia permitieron que se pueda abordar la bibliografía sobre el método en estudio, pudiendo comprender la naturaleza de las propiedades y fundamentos en los que se basa. Conocer la historia del algoritmo brindó noción de manera ilustrativa de las distintas estrategias que se desarrollan en orden de poder bajar la complejidad de un algoritmo, así como también la velocidad en la que se realiza el avance de la investigación del mismo.

Por último, las implementaciones realizadas -las cuales presentan un mejora en complejidad del algoritmo genérico *push-relabel*-, permitió visualizar y entender mejor los puntos clave de la metodología. Como último comentario, se cree que tanto las versiones implementadas como la mejora conjunto de vértices activos por *highest label* podrían ser un buen trabajo de laboratorio de la materia.

Referencias

[1] Goldberg, A.V., Tarjan, R.E. (1988): *A new approach to the maximum flow problem.* J. Assoc. Comput. Mach. 35, 921-940

[2] Cormen T.H.,Leiserson C.E., Rivest R.L., Stein C. *Introduction to algorithms* Third edition. The MIT Press 2009. Chapter 26 Maximum Flow.

[3] Ahuja R.k, Magnanti T.L, Orlin J.B. *Network Flows. Theory, Algorithms and Applications.* Prentice Hall 1993. Chapter 7, section 7.7 FIFO preflow-push algorithm.

[4] Jungnickel Dieter. *Graphs, Networks and Algorithms* Fourth Edition, Springer 2013. Chapter 6, section 6.6:207-209.

[5] Ahuja, R.K., Orlin, J.B. (1987): *A fast and simple algorithm for the maximum flow problem.* Sloan Working Paper 1905-87, Sloan School of Management, M.I.T

[6] Ahuja, R.K., Orlin, J.B., Tarjan, R.E. (1987): *Improved time bounds for the maximum flow problem.* Technical Report CS-TR-118-87, Department of Computer Science, Princeton University (SIAM J. Comput., to appear)

[7] Sleator, D. D., y Tarjan, R. E. *Self-adjusting binary search trees.* J. ACM 32,3 (July 1985), 652-686.

[8] J. Cheriyan, T. Hagerup, and K. Mehlhorn, *Can a maximum flow be computed on o(nm) time?*, ICALP, 1990, pp. 235–248.

[9] V. King, S. Rao, and R. Tarjan, *A faster deterministic maximum flow algorithm*, Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), SODA '92, Society for Industrial and Applied Mathematics, 1992, pp. 157–164.

[10] J. B. Orlin, *Max flows in O(nm) time, or better*, Proceedings of the 45th Annual ACM Symposium on Theory of Computing (Palo Alto, CA, USA), STOC '13, ACM, 2013, pp. 765–774.

[11] Teórica AEDIII 1er cuatrimestre 2018 FCEN, UBA. *Flujo en redes - Problema de flujo máximo.* https://campus.exactas.uba.ar/course/view.php?id=992section=4

4. Apéndice

4.1. Implementación del algoritmo *relabel-to-front*

La implementación aquí presentada se basa en el pseudocódigo proporcionado en [2].

- **Lista de vecinos**

Los ejes de la red $G = (V, E)$ se organizan en una lista de adyacencias N. De este modo, el vértice w aparece en la lista de $N[v]$ si (v, w) o $(w, v) \in E$. La operación $head(N[v])$ retorna el primer vértice en la lista $N[v]$, y $w.next\_neighbour$ apunta al siguiente vecino de v en $N[v]$. Este puntero es nulo si w era el último vértice en la lista de vecinos de v. El atributo $v.current$ apunta al vecino w de v que se esta considerando actualmente.

```
Relabel-to-Front(G,s,t)
    Init-Preflow(G,s)
    for each v ∈ G.V − {s,t}
        push_back(L,v)
    for each v ∈ G.V − {s,t}
        v.current = N[v].head()
    v = L.head()
    while v ≠ NULL
        old-height = v.h
        Discharge(v)
        if v.h > old-height
            mover a v al frente de la lista L
        v = v.next

Discharge(v)
    while v.e > 0
        w = v.current
        if w == NULL
            Relabel(v)
            v.current = N[v].head()
        else if c_f(v,w) > 0 ∧ v.h == w.h + 1
            Push(v,w)
        else v.current = w.next-neighbour
```

4.2. Implementación del algoritmo *relabel-to-back*

Aqui se presenta la implementación del algoritmo en base al pseudocódigo brindado en el desarrollo.

La inicialización de las alturas se realiza en base a la aplicación de BFS. La estrucutra Q se trata de una cola que posee los vértices a ser explorados. Para poder analizar en $O(1)$ la presencia de un vértice en la cola, se empleó un diccionario lineal cuya indexación permitía conocer la pertenencia o no.

```
Relabel-to-Back(G,s,t)
    Init-Preflow(G,s)
    #Se agregan vecinos de s a la cola
    for each (v,w) ∈ E
        if v == s ∧ w ≠ t
            Q.add(w)
    #Inicialización de alturas
     for each v ∈ V\{s}
        h(v) = #ejes en el camino mas corto de v a t
    #ciclo principal
    while Q ≠ ∅
        v = Q.pop()
        #v descarga su exceso a todos sus vecinos admisibles
        while e(v) > 0 ∧ ∃(v,w) ∈ E_f ∧ v.h == w.h + 1
            push(v,w)
```

```
        if w ≠ s, t ∧ w ∉ Q
            Q.add(w)
    if e(v) > 0   #si aun hay exceso en v se lo agrega nuevamente a Q
        relabel(v)
        Q.add(v)
```