APLICACIÓN DE LAS GPU'S EN LA SOLUCIÓN DEL PROBLEMA DE FLUJO MÁXIMO

SERGIO GUTIÉRREZ MOTA

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Programación y Tecnología Software

Convocatoria: Septiembre
Calificación: Sobresaliente

10 de septiembre de 2012

Director:

Antonio Gavilanes Franco

Autorización de difusión

Autor
Sergio Gutiérrez Mota

10 de septiembre de 2012

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Aplicación de las GPU's en la solución del problema del flujo máximo", realizado durante el curso académico 2011-2012 bajo la dirección de Antonio Gavilanes Franco en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

En los últimos años, el ámbito de la computación gráfica ha sido uno de los principales objetos de estudio gracias al avance tecnológico llevado a cabo en los dispositivos dedicados a ello. Las GPUs modernas han visto potenciada su utilidad con la llegada de plataformas como CUDA que permiten escribir programas de propósito general en este tipo de procesadores paralelos. Entre los problemas a los que más se ha recurrido están todos aquellos relacionados con grafos, debido a su utilidad en numerosos campos de la vida real.

Este trabajo se centra en la resolución del problema de hallar el flujo máximo de un grafo utilizando CUDA. Para ello, se estudian diferentes algoritmos existentes en la actualidad y se implementa una versión paralela de uno de ellos, el algoritmo de *push-relabel*.

Se han realizado pruebas para medir el rendimiento de nuestra implementación sobre tres tipos diferentes de grafos, *ADG*, *RMFGEN* y *Washington-RLG* que son los típicamente utilizados para los problemas de flujo máximo. Las pruebas muestran cómo nuestra implementación es, en media, 20 veces más rápida que la implementación secuencial.

Palabras clave

Flujo máximo, algoritmo push-relabel, CUDA

Abstract

In the last few years, graphics computation has been one of the most important research fields thanks to the advances in GPU hardware. The evolution of these devices has given researchers access to low cost hardware to solve general-purpose problems. Frameworks like CUDA have been developed in order to help programmers to write code for the GPU exposing the device as an array of SIMD multi-core processors. Recently, researchers have focused on solving graph problems in this new platform due to its wide range of applications.

This document focuses in the resolution of the maximum flow problem using CUDA. We present some of the faster algorithms to date and implement the parallel version of one of them, the *push-relabel* algorithm.

We use three of the most common type of graphs for the maximum flow problem in our evaluations, *ADG's*, *RMFGEN's* and *Washington-RLG's*. Experimental results show that our implementation achieves 20x speedup when compared with the sequential version of the algorithm.

Keywords

Maximum flow, push-relabel algorithm, CUDA

Índice general

Capítulo 1

Introducción

Las redes de flujo son un modelo que permite representar sistemas tales como mapas de carreteras, redes de tuberías o conexiones de red, desde un punto de vista abstracto. Todos estos sistemas comparten una serie de características, a saber, se pueden asimilar a grafos cuyos arcos tienen una capacidad máxima y por los cuales transitan elementos (coches, líquidos, datos, etc.).

Sobre las redes de flujo se pueden plantear numerosos problemas, entre ellos el de la búsqueda del flujo máximo que consiste en encontrar la máxima cantidad de elementos que se pueden enviar entre dos vértices de la red.

1.1. Red de flujo

Una red de flujo es un grafo dirigido $G = (V, E)$ en el cual cada arco $(u, v) \in E$ tiene una capacidad no negativa $c(u, v) \geq 0$. Se asume que si $(u, v) \notin E$ entonces $c(u, v) = 0$. En toda red de flujo se distinguen dos vértices especiales, la **fuente s** y el **destino o sumidero t**. Por comodidad, se suele asumir que todo vértice se encuentra en alguna ruta de s a t.

En la figura 1.1 se muestra una red de flujo donde el valor de cada arco representa su capacidad.

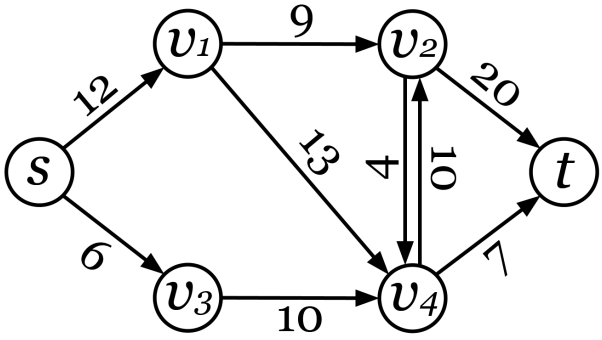


Figura 1.1: Representación de una red de flujo

Dada una red de flujo, un flujo para ella es una función $f : V \times V \to \mathbb{R}$ que satisface las siguientes propiedades:

- Restricción de capacidad: $\forall u, v \in V : f(u, v) \leq c(u, v)$
- Simetría: $\forall u, v \in V : f(u, v) = -f(v, u)$
- Conservación de flujo: $\forall u \in V - \{s, t\} : f(u, V) =_{def} \sum_{v \in V} f(u, v) = 0$

La cantidad $f(u, v)$, que puede ser positiva, negativa o nula, es llamada flujo desde el vértice u al vértice v. El valor de un flujo f queda entonces definido como:

$$|f| =_{def} f(s, V)$$

es decir, el valor de un flujo es la cantidad total de flujo que fluye desde la fuente.

Examinemos, desde un punto de vista intuitivo, las condiciones que debe satisfacer un flujo. La restricción de capacidad impone un límite a la red en cuanto a la cantidad de flujo que se puede enviar por cada arco. La simetría es una propiedad añadida para facilitar la notación y nos dice que si circula un flujo f desde el vértice u hasta el vértice v, entonces ese mismo flujo, pero con signo negativo, puede imaginarse que circula en sentido inverso (de v a u). Por último, la conservación de flujo nos dice que, para cualquier vértice, sin contar s ni t, la suma del flujo de salida de dicho vértice es nula. Si combinamos esta propiedad con la de la simetría podemos llegar a la conclusión de que para cualquier vértice, de nuevo sin contar s ni t, se cumple que entra y sale la misma cantidad de flujo, esto es:

$$\forall u \in V - \{s, t\} : f(u, V) = f(V, u) = 0$$

1.1.1. Ejemplo

En la figura 1.2 se muestra un diagrama de la misma red de flujo vista antes, pero con un flujo concreto. Cada vértice, excepto la fuente, denotada por s, y el destino, denotado por t, viene identificado por un número. Además, cada arco lleva asociado un par de valores que representan, respectivamente, flujo/capacidad.

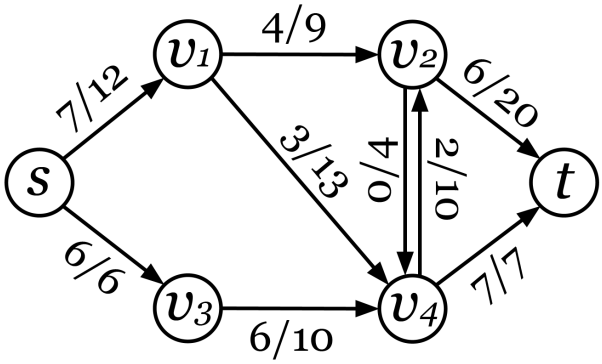
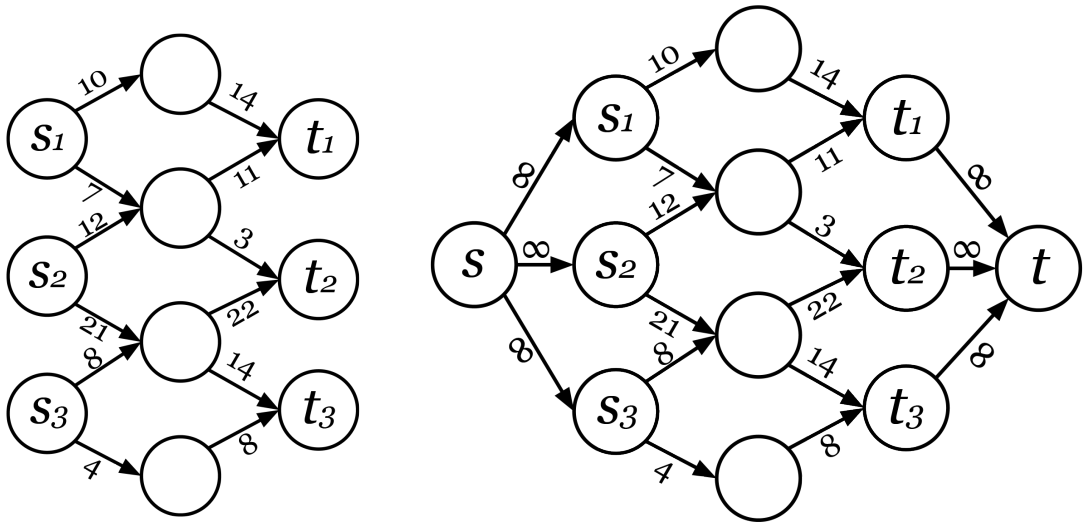


Figura 1.2: Diagrama de una red de flujo con un flujo arbitrario

1.2. Flujo máximo

El problema del flujo máximo consiste en hallar la máxima cantidad de flujo que se puede enviar, en una red de flujo, desde el vértice fuente s al vértice destino t.

Existe la posibilidad de contemplar más de un vértice fuente y más de un vértice destino, pero como se muestra en la figura 1.3, es posible encontrar una red de flujo con una sola fuente y un solo destino equivalente a la red original por lo que a partir de ahora nos centraremos en el caso más sencillo.



(a) (b)

Figura 1.3: Para transformar una red de flujo con múltiples vértices fuente y destino a una con un solo vértice fuente y destino, es suficiente con añadir un vértice fuente, conectado con las anteriores fuentes mediante arcos de capacidad infinita y añadir un vértice destino al que se le hace llegar los antiguos vértices destino, también mediante arcos de capacidad infinita.

La figura 1.4 muestra la red de flujo del ejemplo, con una posible solución al problema del flujo máximo. Obsérvese que, mientras el flujo del ejemplo tenía un valor de 13, el de la figura 1.4 tiene un valor de 18. En las dos siguientes secciones se detallan los dos métodos más populares para resolver este problema.

1.3. Método de Ford-Fulkerson

En 1956, L. R. Ford y D. R. Fulkerson [4] propusieron una forma de solucionar, de manera metódica, el problema del flujo máximo. El método se basa en tres ideas fundamentales: redes residuales, caminos aumentados y el teorema del corte mínimo. Éste último es la corrección

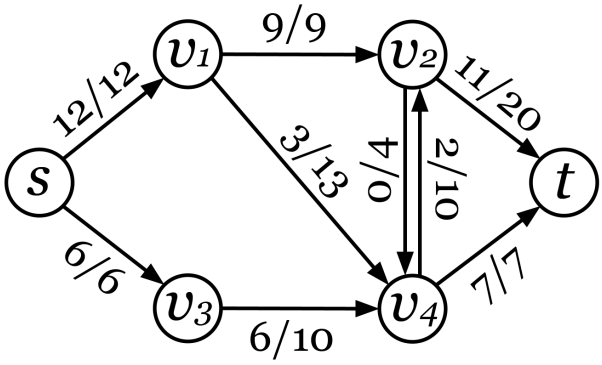


Figura 1.4: Flujo máximo para la red de flujo de ejemplo

del método que propusieron y no lo vamos a tratar en este trabajo. En las siguientes secciones se detallan los conceptos necesarios para comprender el método de Ford-Fulkerson.

1.3.1. Redes residuales

Supongamos que se tiene una red de flujo $G = (V, E)$ con vértice fuente s y vértice destino t. Dado un flujo f y un par de vértices u y v de la red, se define como **capacidad residual** del arco (u, v) a la cantidad de flujo adicional que se puede enviar desde u a v antes de sobrepasar la capacidad $c(u, v)$. Más formalmente:

$$c_f(u, v) =_{def} c(u, v) - f(u, v)$$

donde $c_f(u, v)$ es la capacidad residual del arco (u, v).

Dada entonces una red de flujo $G = (V, E)$ y un flujo f, la **red residual** de G inducida por f se define como $G_f = (V, E_f)$ donde

$$E_f =_{def} \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

o lo que es lo mismo, G_f es un grafo con los mismos vértices que el original, pero cuyos arcos son **arcos residuales**, es decir, aquellos que tienen una capacidad residual positiva. Desde un punto de vista intuitivo, los arcos residuales representan la cantidad máxima de flujo que podemos enviar desde un vértice a otro sin superar la capacidad del arco.

En la figura 1.5 se muestra la red residual de la red de flujo usada como ejemplo en la figura 1.2, con un flujo arbitrario. Como se puede observar, si se tiene en cuenta el grafo original, se han añadido o eliminado arcos. Esto es debido a que lo que se muestra en la figura son arcos residuales. Tomando como ejemplo el arco (v_1, v_2), que tiene una capacidad de 9, se puede observar que aún es posible enviar 5 unidades de flujo desde el vértice v_1 hacia el vértice v_2. Si el arco tiene una capacidad de 9, y ya sólo podemos enviar 5 unidades de flujo, significa que hemos enviado 4 unidades de flujo por el arco original. El arco de sentido inverso representa que se pueden devolver las 4 unidades de flujo que se han enviado. Formalmente el arco (v_2, v_1) tiene capacidad 0 y el valor de su capacidad residual

es $c_f(v_2, v_1) = c(v_2, v_1) - f(v_2, v_1) = c(v_2, v_1) + f(v_1, v_2) = 0 + 4 = 4$. Por eso la red residual contiene no uno, sino dos arcos entre v_1 y v_2. Uno con la capacidad residual que podemos mandar todavía de v_1 a v_2 y otro de v_2 a v_1 con la capacidad residual de lo que ya se ha enviado.

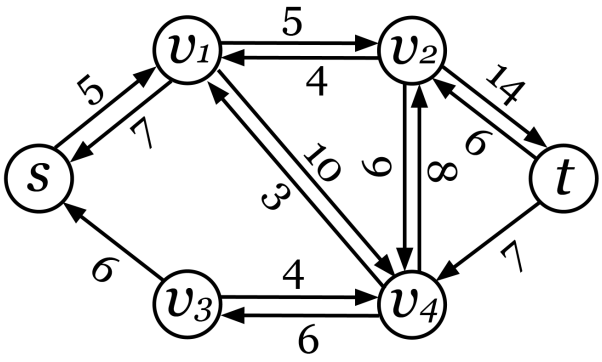


Figura 1.5: Red residual del grafo de la figura 1.2

1.3.2. Caminos aumentados

Dada una red de flujo $G = (V, E)$ y un flujo f, un **camino aumentado** p es un camino simple desde s a t en la red residual G_f. Por la definición de red residual, cada arco (u, v) de un camino aumentado admite una cantidad positiva de flujo adicional desde u a v.

En la figura 1.6 se muestra un posible camino aumentado del grafo del ejemplo de la figura 1.5. Como se puede observar, se puede aumentar el flujo enviado de s a t a través del camino aumentado en un máximo de 5 sin violar la restricción de capacidad de ninguno de los arcos del camino. A la cantidad máxima de flujo adicional que se puede enviar a través de un camino aumentado p se le llama **capacidad residual** de p y formalmente se define como:

$$c_f(p) =_{def} min\{c_f(u, v) : (u, v) \text{ está en } p\}.$$

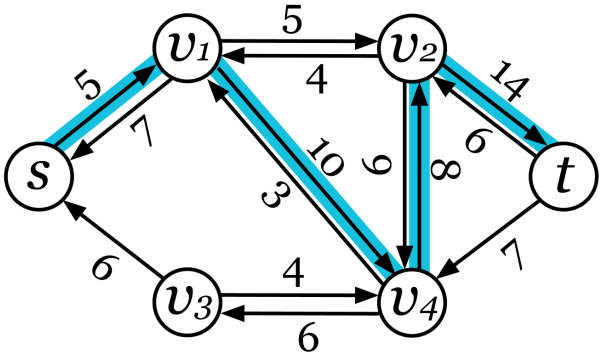


Figura 1.6: Camino aumentado sobre una red residual

1.3.3. Método de Ford-Fulkerson

La propuesta original presentada por Ford y Fulkerson contiene la forma de obtener el flujo máximo de una red de flujo mediante el método presentado en el algoritmo 1.1.

```
function FORD-FULKERSON(G, s, t)
    inicializar flujo f a 0
    while exista un camino aumentado p en G do
        aumentar el flujo f a través de p
    end while
    return f
end function
```

Algoritmo 1.1: Método de Ford-Fulkerson

Dado que no se concreta la forma en que se deben encontrar los caminos aumentados ni se especifica la cantidad de flujo que se debe enviar por cada uno, la propuesta es considerada comúnmente como un esquema algorítmico y no como un algoritmo concreto. En la siguiente sección se estudia una posible implementación del esquema y se visualiza su desarrollo mediante un ejemplo.

1.3.4. Algoritmo de Edmonds-Karp

El algoritmo de Edmonds-Karp [3] no es más que una concreción del método de Ford-Fulkerson en el que la búsqueda de los caminos aumentados se realiza mediante una búsqueda primero en anchura (*breath-first search*, BFS) y para el que, por cada iteración del método, se ha decidido aumentar el flujo, a través del camino aumentado, en la capacidad residual, $c_f(p)$, de este.

```
function EDMONDS-KARP(G, s, t)
    for all (u, v) ∈ E[G] do
        f[u, v] ← 0
        f[v, u] ← 0
    end for
    while se encuentre un camino p de s a t en G_f mediante BFS do
        c_f(p) ← min{c_f(u, v) : (u, v) está en p}
        for all (u, v) en p do
            f[u, v] ← f[u, v] + c_f(p)
            f[v, u] ← −f[u, v]
        end for
    end while
end function
```

Algoritmo 1.2: Algoritmo de Edmonds-Karp

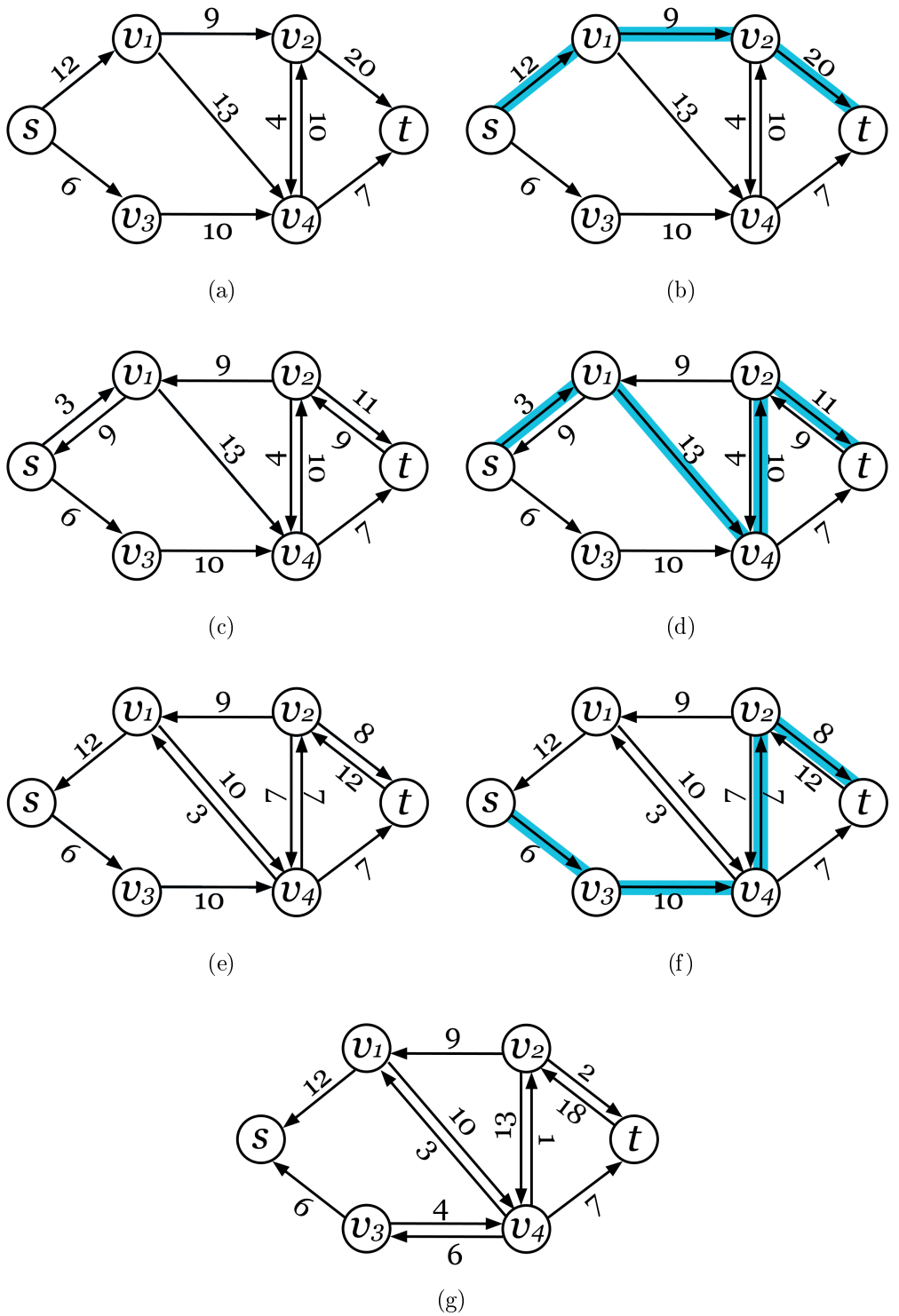


Figura 1.7: Ejecución del algoritmo de Edmonds-Karp sobre el grafo del ejemplo. (a)-(f) La columna de la izquierda muestra el grafo residual y la columna de la derecha los caminos aumentados encontrados y sobre los cuales se envía el flujo. (g) Flujo máximo encontrado.

En la figura 1.7 se puede observar la evolución del algoritmo en una red de flujo de ejemplo.

Puede demostrarse que el número de aumentos en el flujo realizados por este algoritmo hasta hallar el flujo máximo es $O(|V|\,|E|)$. Si además consideramos que encontrar un camino aumentado tiene un coste $O(|E|)$ utilizando BFS, entonces podemos concluir que el tiempo de ejecución del algoritmo de Edmonds-Karp es $O(|V|\,|E|^2)$ [2].

1.4. Algoritmo de *push-relabel*

Este algoritmo, presentado por Goldberg y Tarjan en 1987-1988 [5], representa una alternativa más eficiente al método de Ford-Fulkerson para resolver el problema de flujo máximo.

En lugar de tratar el grafo como un todo, este algoritmo trabaja con cada vértice y sus vecinos de una forma local. Durante la ejecución, puede no cumplirse la propiedad de la conservación de flujo sino una versión relajada:

$$\forall u \in V - \{s\} : f(V, u) \geq 0$$

o lo que es lo mismo, la cantidad de flujo en cada vértice (excepto el fuente) puede ser positiva. Recuérdese que los flujos cumplen $f(V, u) = 0$ para todo $u \in V - \{s, t\}$. Debido a este cambio, al flujo que se maneja a lo largo de la ejecución de este algoritmo se le llama **preflujo**, que es una función $f : V \times V \to \mathbb{R}$ que además de esta nueva propiedad, cumple la restricción de capacidad y la simetría, vistas anteriormente para los flujos. A la cantidad de flujo de un vértice u se le llama ahora **exceso de flujo** de u. Es decir, formalmente, $e(u) =_{def} f(V, u)$. Aunque el exceso de flujo puede ser negativo, nosotros hablaremos de exceso de flujo de u para indicar que $e(u) > 0$. Cuando no sea así, diremos explícitamente que el exceso es negativo.

Para visualizar el funcionamiento del algoritmo de *push-relabel* de una forma intuitiva podemos imaginarnos la red de flujo como un sistema de cañerías, donde los arcos son tuberías y los vértices son sus intersecciones. Dado que ahora los vértices pueden tener un exceso de flujo podemos imaginarnos que cada vértice dispone de un depósito lo suficientemente grande para almacenar su exceso de flujo, si lo tuviera. Además, cada vértice, su depósito y todas las cañerías que se conectan con él, están en una plataforma cuya altura aumenta a lo largo de la ejecución del algoritmo. La altura de cada vértice determina hacia dónde se puede mandar flujo: sólo hacia plataformas más bajas.

Al principio del algoritmo, el vértice fuente está a una altura $|V|$ y el resto se sitúa a altura 0. El algoritmo empieza enviando todo el flujo que pueda desde el vértice fuente, para entrar a continuación en un proceso iterativo que se encarga de levantar las plataformas (*relabel*) y seguir *empujando* el exceso de flujo hacia vértices más cercanos al vértice destino (*push*). Al finalizar el algoritmo, no sólo habremos transformado el preflujo en un flujo legal sino que este flujo será además máximo.

A continuación se pasa a detallar las dos operaciones más importantes del algoritmo.

1.4.1. Operación *push*

La operación de empujado o *push* se puede aplicar a un vértice u siempre y cuando se cumplan las siguientes condiciones:

- $e(u) > 0$: u es un vértice con exceso de flujo
- $c_f(u, v) > 0$: es posible enviar parte del exceso a un vértice vecino v
- $h(u) > h(v)$: la altura del vértice u es mayor que la de su vecino v

La cantidad de exceso que se empuja es el mínimo entre el exceso de flujo que tiene el vértice u y la cantidad máxima que se puede enviar por el arco que lo conecta con v. En el algoritmo 1.3 se puede ver la implementación de esta operación en pseudocódigo. Como se puede observar, una vez comprobados los requisitos, se recoge en una variable auxiliar d_f la cantidad de flujo que se va a enviar y se pasa a actualizar tanto los excesos de flujo de ambos vértices como el flujo de los arcos que los unen.

Requiere: $e(u) > 0$, $c_f(u, v) > 0$ y $h(u) > h(v)$
  function PUSH(u, v)
    $d_f \leftarrow \min(e(u), c_f(u, v))$
    $f(u, v) \leftarrow f(u, v) + d_f$
    $f(v, u) \leftarrow -f(u, v)$
    $e(u) \leftarrow e(u) - d_f$
    $e(v) \leftarrow e(v) + d_f$
  end function

Algoritmo 1.3: Operación *push*

1.4.2. Operación *relabel*

La operación de reetiquetado o *relabel* se encarga de actualizar la altura de los vértices cuando estos tienen exceso de flujo y no pueden enviárselo a ninguno de sus vecinos por encontrarse a menos altura. Para aplicar correctamente esta operación, han de cumplirse las siguientes condiciones:

- $e(u) > 0$: u es un vértice con exceso de flujo
- $\forall v \in \{v|(u, v) \in E_f\}\,(h(u) \leq h(v))$: la altura del vértice u es menor o igual que la de todos sus vecinos v

Si se cumplen estos requisitos entonces se puede reetiquetar el vértice u para situarlo a una altura mayor que alguno de sus vecinos; en concreto, se sube lo justo para estar por encima de, al menos, un vecino. En el algoritmo 1.4 se presenta el pseudocódigo de esta operación.

Requiere: $e(u) > 0$ y $\forall v \in \{v|(u,v) \in E_f\} : h(u) \leq h(v)$
  function RELABEL(u)
    $h(u) \leftarrow 1 + \min(h(v) : (u,v) \in E_f)$
  end function

Algoritmo 1.4: Operación *relabel*

1.4.3. Algoritmo general

El algoritmo de *push-relabel* se presenta en el algoritmo 1.5. Primero se inicializan los valores de las alturas, excesos y flujos para después enviar todo el flujo posible desde el vértice fuente a sus vecinos, como se muestra en el algoritmo 1.6. El algoritmo entra entonces en una sección en la que intentará aplicar alguna de las dos operaciones vistas antes, a todos los vértices de la red (exceptuando el fuente y el destino). Cuando se sale del bucle principal, el flujo obtenido es máximo.

  function PUSH-RELABEL(G)
    INICIALIZA-PREFLUJO(G, s)
    while se pueda aplicar una operación *push* o *relabel* **do**
      aplicar una operación *push* o *relabel*
    end while
  end function

Algoritmo 1.5: Algoritmo de *push-relabel*

1.4.4. Análisis del algoritmo de *push-relabel*

Como se detalla en esta sección, el tiempo de ejecución del algoritmo de *push-relabel* es $O(|V|^2 |E|)$ que mejora el tiempo obtenido con el algoritmo de Edmonds-Karp. Para determinar el tiempo de ejecución del algoritmo vamos a calcular una cota superior del número de operaciones que se aplican, en concreto, hay que hallar una cota superior en el número de operaciones *relabel* y en el número de operaciones *push*.

Cota superior en el número de operaciones *relabel*

Vamos a trazar un límite en el número de operaciones *relabel* que se aplican a lo largo del algoritmo. Para ello, es necesario razonar sobre la altura máxima a la que se puede levantar un vértice.

En primer lugar, demostremos que, dados $u, v \in V$:

$$\text{(I) } (u,v) \in E_f \Rightarrow h(u) \leq h(v) + 1$$

Formalmente se demuestra por inducción sobre el número k de operaciones *push* o *relabel* realizadas.

```
function INICIALIZA-PREFLUJO(G, s)
    for all u ∈ V(G) do
        h(u) ← 0
        e(u) ← 0
    end for
    for all (u, v) ∈ E(G) do
        f(u, v) ← 0
        f(v, u) ← 0
    end for
    h(s) ← |V(G)|
    for all u ∈ Adj(s) do
        f(s, u) ← c(s, u)
        f(u, s) ← −c(s, u)
        e(u) ← c(s, u)
        e(s) ← e(s) − c(s, u)
    end for
end function
```

Algoritmo 1.6: Inicialización del preflujo

El caso $k = 0$ es sencillo, pues todos los vértices, salvo la fuente, tienen altura 0. Y la fuente sólo aparece en arcos (u, s) en la red residual. Para el paso inductivo distinguimos dos casos:

a) se ha realizado una operación *relabel* al vértice w. Si $w \neq u, v$, entonces (I) es inmediato. Si $w = v$, basta observar que la altura de un vértice nunca decrece durante la ejecución del algoritmo. Si $w = u$, la prueba es sencilla, por definición de la altura que pasa a tener u

b) se ha realizado una operación *push* al arco e. Si $e \neq (u, v)$, es inmediato. Si no, $h(u) > h(v)$. Si aparece (v, u) en la red residual, se tiene $h(v) < h(u)+1$. Por otra parte, si (u, v) permanece en la red residual entonces, por hipótesis de inducción, $h(u) \leq h(v)+1$, antes de aplicar *push*. Pero $h(u) > h(v)$, para poder aplicarlo, luego $h(u) = h(v) + 1$.

Probado esto, si p es un camino en la red residual entre los vértices u y v, entonces, aplicando repetidas veces (I), $h(u) \leq h(v)+l$, siendo l la longitud de p. Como dicha longitud puede ser, como mucho $|V| - 1$, resulta que $h(u) \leq h(v) + |V| - 1$.

Tomemos ahora un vértice u con exceso de flujo. Como el único vértice que genera flujo es el vértice fuente, podemos asumir, de forma intuitiva, que existe un camino desde u al vértice s en la red residual. Piénsese que el camino de arcos inversos de éste es el que ha hecho posible mandar el flujo desde s al vértice u. Por tanto, por lo que acabamos de demostrar, resulta que:

$$\text{(II) } e(u) > 0 \Rightarrow h(u) \leq 2\,|V| - 1$$

Ahora bien, en la red residual son $|V| - 2$ los vértices a los que se puede aplicar una operación *relabel*, luego el número máximo de veces que se aplica esta operación a lo largo de la ejecución del algoritmo es $(|V| - 2)(2|V| - 1) \leq 2|V|^2$.

Cota superior en el número de operaciones *push*

Vamos a hallar una cota del número de operaciones *push* que se llegan a aplicar en la ejecución del algoritmo. Distinguimos dos tipos de estas operaciones: las saturadoras (aquellas que saturan el arco por el cual se transfiere el flujo) y las no saturadoras.

Empezamos con las primeras. Sea (u, v) un arco sobre el que se ha realizado un *push* saturador, por lo tanto se cumple que $h(u) > h(v)$. El arco desaparece de la red residual y aparece el inverso, es decir, (v, u). Para llegar a hacer otra operación *push* saturadora desde v a u es necesario aplicar antes una operación *relabel*, para lo cual la altura de v debe aumentar en, al menos, 2. Ahora bien, la altura de un vértice distinto al fuente, empieza en 0 y, según vimos en (II), nunca excede $2|V| - 1$, luego el número de veces que un vértice puede aumentar su altura en 2 es menor que $|V|$, por lo que el número de operaciones *push* saturadoras entre u y v es menor que $2|V|$ y por lo tanto $|V||E|$ es una cota del número de operaciones *push* saturadoras.

Para obtener una cota del número de operaciones *push* no saturadoras definimos la **energía** de un preflujo f como:

$$\Phi(f) =_{def} \sum_{v:e(v)>0} h(v)$$

Como el vértice fuente no tiene exceso de flujo y el resto de los vértices empiezan teniendo altura 0, la energía del preflujo inicial es 0 y va variando según se aplican operaciones *relabel* y *push*, saturadoras o no saturadoras.

Un reetiquetado aumenta la energía del preflujo, pues el rango de la suma no varía y las alturas nunca decrecen, y lo hace en una cantidad menor que $2|V|$, según vimos en (II).

Un *push* saturador de u a v no cambia las alturas de los vértices implicados, pero puede hacer que v pase a tener exceso de flujo, con lo que la energía, de aumentar, lo hace en $2|V| - 1$, como máximo, de nuevo por (II).

Consideremos un *push* no saturador sobre el arco (u, v). Por definición de esta operación se cumple $e(u) > 0$ antes de realizarla, y $e(u) = 0$, después. Por tanto, la energía del preflujo disminuye en $h(u)$. Además, tras la operación, v pasa a tener exceso de flujo, si no lo tenía ya. Luego la energía del preflujo aumenta en, a lo más, $h(v)$. Pero, para poder aplicar el *push* se tenía $h(u) > h(v)$, luego un *push* no saturador disminuye la energía del preflujo en al menos, $1 (\leq h(u) - h(v))$.

Antes de dar un paso cualquiera del algoritmo, algún vértice tiene sobreflujo luego la energía del preflujo es > 0. Llamemos r al número de operaciones *relabel* realizadas, ps al

de *push* saturadoras y pn al de *push* no saturadoras. Entonces:

$$\begin{aligned} 0 < \Phi(f) &\leq 2\,|V|\,r + 2\,|V|\,ps - pn \\ &= 2\,|V| \cdot 2\,|V|^2 + 2\,|V| \cdot 2\,|V|\,|E| - pn \\ &= 4\,|V|^2\,(|V| + |E|) - pn \end{aligned}$$

luego

$$pn < 4\,|V|^2\,(|V| + |E|)$$

En conclusión, el número de operaciones *relabel* y *push* que realiza el algoritmo es del orden $O(|V|^2\,|E|)$.

1.5. Ejemplo de evolución del algoritmo

En las figuras 1.9, 1.10 y 1.11 se muestra la evolución del algoritmo de *push-relabel* cuando se aplica al grafo del ejemplo de la figura 1.2. Como se puede observar en la figura 1.8, los vértices se simbolizan mediante discos que representan su altura. Un solo disco es equivalente a decir que el vértice tiene altura 0 y cada disco adicional suma 1 a la altura total del vértice. Además de la altura, se ha añadido una etiqueta al lado de cada vértice que contiene su identificador y su exceso de flujo.

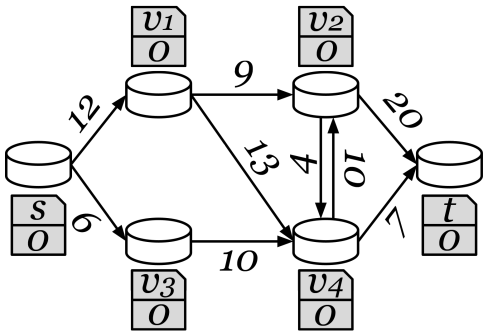


Figura 1.8: Grafo del ejemplo de la figura 1.2 con la nueva representación

A lo largo de la ejecución del algoritmo se asume secuencialidad en el tratamiento de los vértices activos (aquellos con exceso de flujo, identificados en el diagrama por una etiqueta de color naranja) de forma que estos se tratan en el mismo orden en el que se convierten en activos. Así mismo, se ha optado por aplicar todas las operaciones *push* posibles al vértice activo actual hasta que deje de tener exceso de flujo. Para resaltar cuál es el vértice activo actual, se ha bordeado su etiqueta de color rojo. Por último, un vértice coloreado de azul representa que se va a aplicar la operación *relabel* sobre dicho vértice. De forma similar, un arco de color azul representa que se va a aplicar la operación *push* sobre dicho arco.

Es importante observar que, dentro del bucle principal del algoritmo, no se repite nunca un mismo estado de la red residual en dos iteraciones diferentes; dicho de otra forma, en cada iteración del bucle principal se modifica alguna de las características de la red residual (altura, flujo o exceso) de forma que no se repite ninguna configuración de ninguna red de flujo anterior. Esta característica del algoritmo puede esbozarse mediante un ejemplo sencillo. Sean u y v dos vértices de altura 5 y 4 respectivamente. Se envía, mediante una operación *push*, 10 de flujo por el arco (u, v). Debido a esta transferencia de flujo, el vértice v tiene ahora, como mínimo, un exceso de 10, convirtiéndose en un vértice activo de la red. Debido a la altura de los vértices, el vértice v no puede devolver su exceso de flujo al vértice u sin aplicar antes una operación *relabel* sobre v. Tanto si aplicamos una operación *push* válida, como si aplicamos una operación *relabel* (que sólo puede aumentar la altura de los vértices), la red resultante de la siguiente iteración da lugar a una configuración completamente nueva. Esta propiedad del algoritmo es muy importante puesto que nos asegura que no se pueden formar ciclos indefinidos y que el algoritmo termina en cualquier caso.

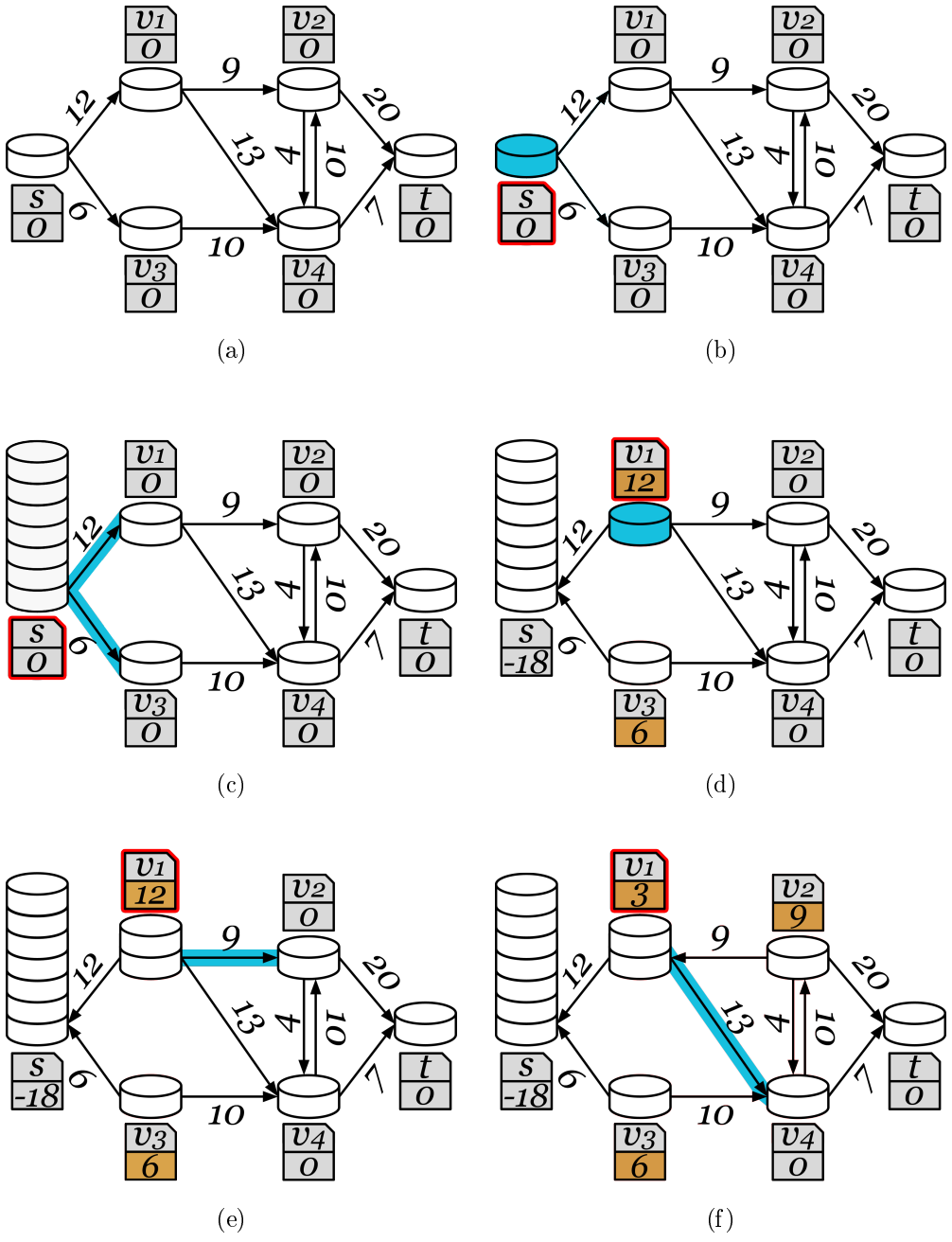


Figura 1.9: (a) Representación del grafo del ejemplo mostrando la altura y excesos de los vértices. (b)-(c) Inicialización del algoritmo, se sitúa el vértice fuente a altura $|V|$ y se envía todo el flujo posible desde él. (d)-(f) Se ejecutan las operaciones *push* o *relabel* a medida que sean aplicables.

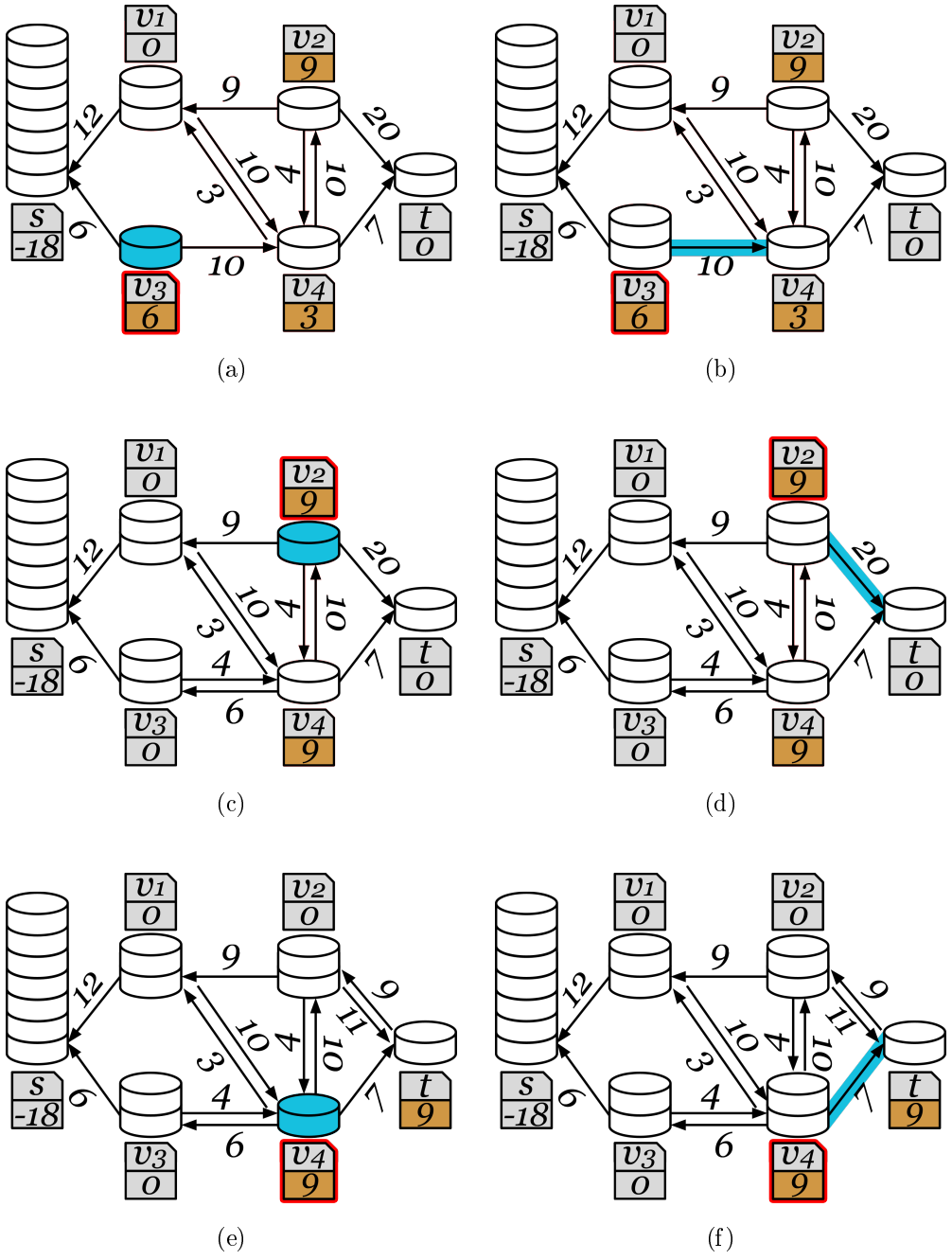


Figura 1.10: (a)-(f) Se continúan ejecutando las operaciones *push* o *relabel* a medida que son aplicables.

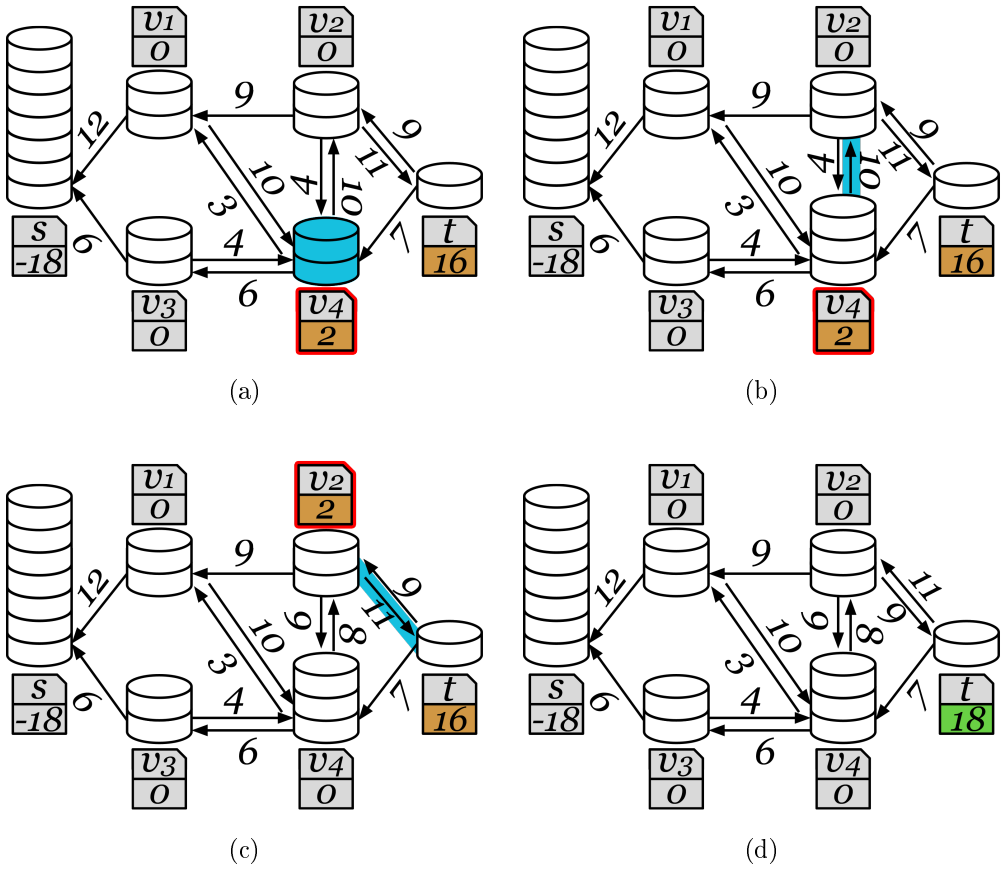


Figura 1.11: (a)-(c) Últimas aplicaciones de las operaciones *push* y *relabel*. (d) Red de flujo final en la que el exceso de flujo del vértice destino es la cantidad de flujo máximo que se puede enviar desde el vértice fuente. Llegados a este punto deja de haber vértices activos distintos al sumidero y por lo tanto el algoritmo ha terminado.

Capítulo 2

Tipos de grafos utilizados

A lo largo del desarrollo de nuestra implementación del algoritmo de *push-relabel*, hemos usado una serie de grafos de prueba para comprobar resultados y medir tiempos de ejecución. Se han usado algunas de las especificaciones de los grafos utilizados en el primer certamen de implementación DIMACS [13]. Los grafos elegidos han sido diseñados para su uso en problemas de redes, entre ellos, el problema del flujo máximo.

2.1. Grafos densos acíclicos

Este tipo de grafos se construyen conectando cada vértice con el resto, pero sin formar ciclos. Si le asignamos un índice a cada vértice, un grafo denso acíclico está formado por un conjunto de vértices $V = \{v_0, v_1, v_2, ...\}$ y un conjunto de arcos $E = \{(v_i, v_j) \in V \times V : i < j\}$. Las capacidades de los arcos se eligen de forma aleatoria. Se asume que el vértice v_0 es la fuente y el vértice v_n es el destino, siendo $n = |V| - 1$.

La figura 2.1 muestra un diagrama con la estructura de un grafo denso acíclico. Se ha distinguido el color de los arcos según el vértice de salida para resaltar el hecho de que cada vértice está conectado con todos los siguientes.

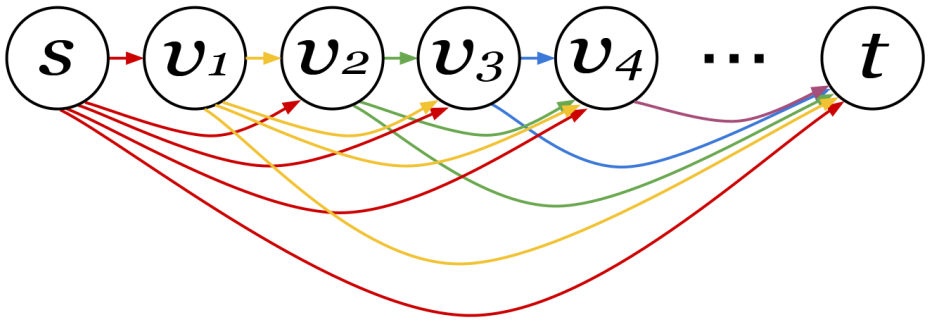


Figura 2.1: Estructura de los grafos densos acíclicos

2.2. Grafos RMFGEN

Los grafos *RMFGEN* fueron originalmente desarrollados por Goldfarb y Grigoriadis [6]. Están compuestos de un número determinado de rejillas cuadradas. Cada rejilla está formada por vértices conectados con sus vecinos en las cuatro direcciones posibles: norte, sur, este y oeste. Además, los vértices de cada rejilla están conectados biyectivamente con los vértices de la siguiente rejilla, usando para ello permutaciones de dichos vértices. El vértice fuente está situado en la esquina noroeste de la primera rejilla y el vértice destino, en la esquina sudeste de la última rejilla.

Tanto el número de vértices de un lado de las rejillas, como el número de rejillas del grafo, pueden modificarse variando los parámetros llamados a y b respectivamente. Es también posible modificar el rango de las capacidades de los arcos con los parámetros c_1 y c_2. La capacidad de los arcos que viajan de una rejilla a otra es elegida de forma aleatoria en el rango de enteros determinado por el intervalo (c_1, c_2). La capacidad de los arcos que conectan los vértices de una misma rejilla vale siempre $c_2 * a * a$.

En la figura 2.2 se puede observar un grafo *RMFGEN* de parámetros $a = 3$ y $b = 2$. Se ha suprimido la capacidad de los arcos a fin de simplificar la representación.

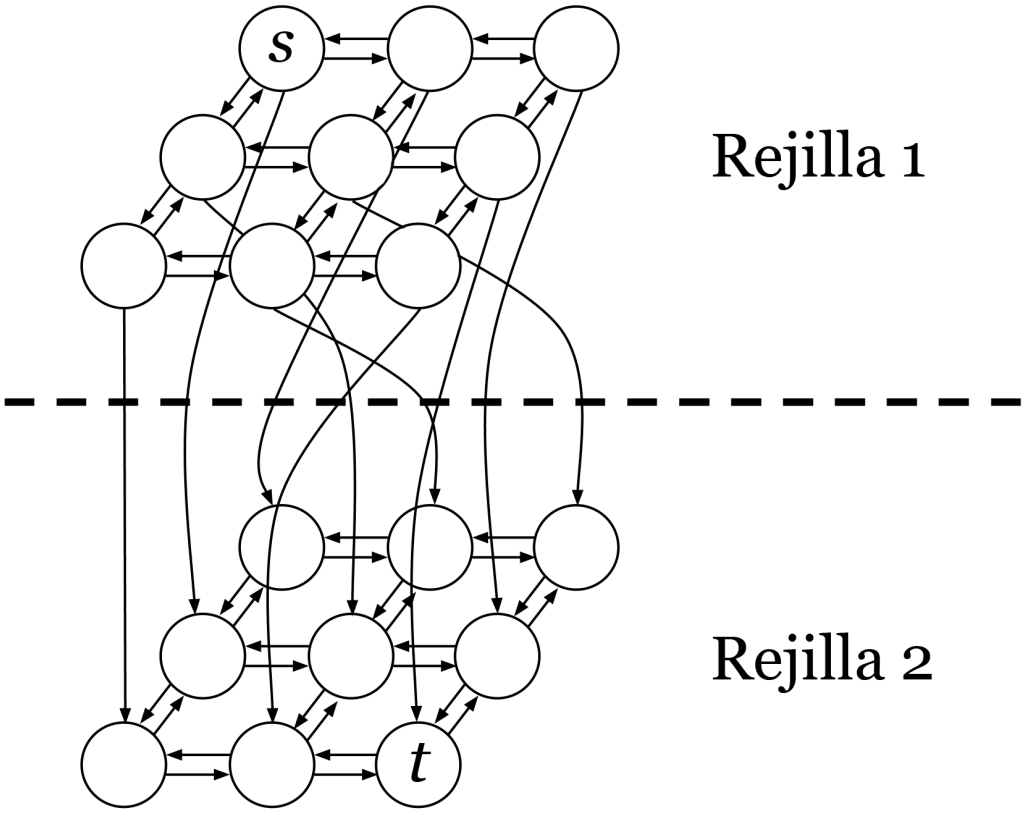


Figura 2.2: Estructura de un grafo *RMFGEN* de parámetros $a = 3$ y $b = 2$

2.3. Grafos Washington-RLG

La herramienta *Washington* [13], desarrollada por Richard Anderson y sus estudiantes, permite la generación de multitud de tipos de grafos. Entre ellos se incluye la generación de *Washington Random Level Graphs* o grafos Washington-RLG.

Los grafos *Washington-RLG* están formados por una rejilla rectangular de vértices de w filas y l columnas. Cada vértice de una fila tiene tres arcos salientes a vértices aleatorios de la siguiente columna. El vértice fuente y destino están situados fuera de la rejilla. La fuente está conectada a todos los vértices de la primera columna y de igual forma, el destino es accesible desde los últimos vértices de cada fila. La capacidad de todos los arcos toma valores aleatorios en el rango de enteros determinado por el intervalo $(1, cap)$, donde cap es uno de los parámetros de la herramienta.

La figura 2.3 muestra un grafo *Washington-RLG* de parámetros $w = 4$, $l = 3$. Las capacidades se han omitido por claridad.

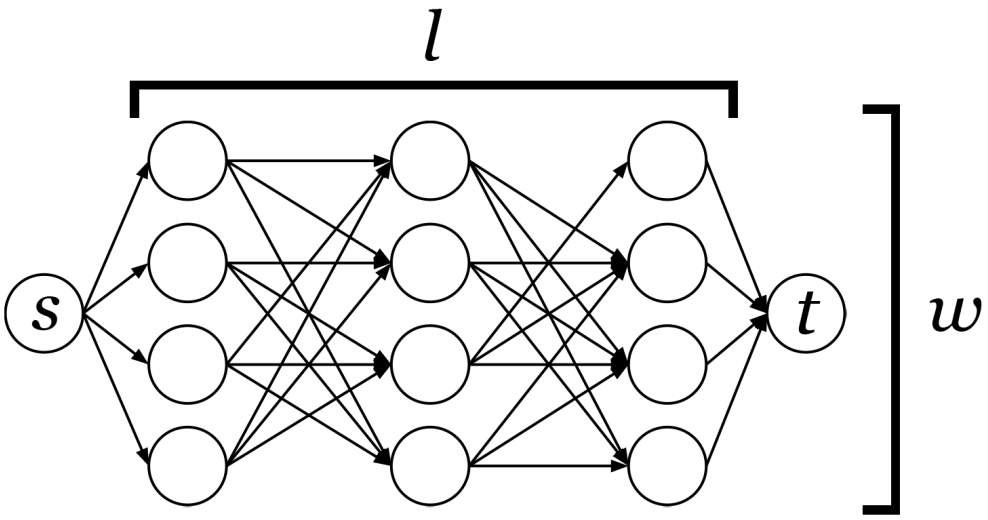


Figura 2.3: Estructura de un grafo *Washington-RLG* de parámetros $w = 4$ y $l = 3$

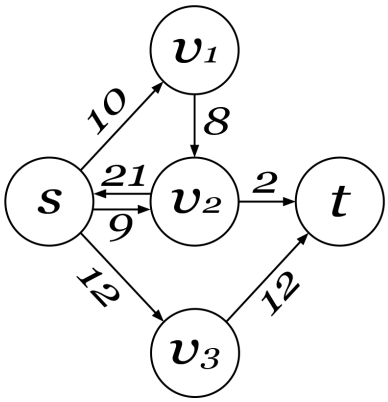
2.4. Representación en fichero

Para almacenar los grafos de prueba utilizados se han usado dos tipos de representación diferentes. Los grafos densos acíclicos se han guardado mediante una matriz de adyacencia. Se ha optado por esta representación debido a que este tipo de grafos tienen un número de vértices reducido, por lo que el tamaño de la matriz se mantiene siempre en tamaños manejables. Además, este tipo de representación facilita mucho la creación de la estructura interna de los grafos que usa nuestra implementación.

En la figura 2.4 se muestra un grafo sencillo representado mediante su matriz de adyacencia tal y como se presenta en el fichero. El primer valor corresponde al número de vértices del grafo. Se asume que el vértice fuente es el primero y el destino, el último.

5				
0	10	9	12	0
0	0	8	0	0
21	0	0	0	2
0	0	0	0	12
0	0	0	0	0

(a) Fichero en formato de matriz de adyacencia



(b) Grafo de ejemplo

Figura 2.4: Representación mediante el formato de matriz de adyacencia

Para los grafos con mayor número de vértices, como los grafos *RMFGEN* y *Washington-RLG*, se ha optado por la representación en formato *DIMACS*. Los ficheros con este formato se dividen en líneas de texto. Cada línea comienza con un carácter diferente dependiendo del tipo de información que contiene. Los diferentes tipos de líneas se detallan a continuación:

- **Comentario**: Los comentarios muestran información en lenguaje natural sobre el grafo y pueden ser ignorados por los programas. Las líneas de comentario comienzan con el carácter *c*.

  c Este es un ejemplo de comentario en formato DIMACS

- **Problema**: Sólo hay una línea de problema en todo el fichero y debe aparecer al principio, antes de mostrar cualquier información sobre los arcos o los vértices. Para las redes de flujo máximo (identificadas por las letras *max*), las líneas de problema dan la información del número de vértices y arcos del grafo. Se usa la letra *p* para marcar el inicio de una línea de problema.

  p max NÚMERO_VÉRTICES NÚMERO_ARCOS

- **Vértice**: Sólo deben existir dos líneas de vértice y su objetivo es informar de cuál es el vértice fuente (mediante el carácter *s*) y cuál el vértice destino (mediante el carácter *t*). Las líneas de vértice se identifican con el carácter *n* al principio.

n ID s/t

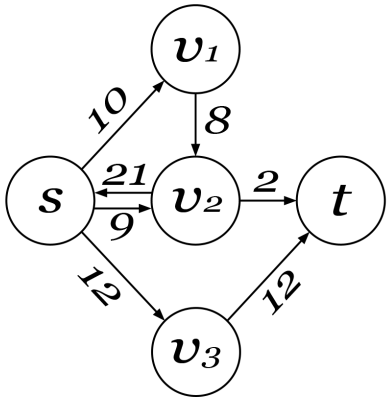
- **Arco**: Existe una línea de arco por cada uno de los arcos de la red e indican el vértice inicial, el vértice final y la capacidad del arco. La letra *a* se usa para identificar este tipo de líneas.

a INICIAL FINAL CAPACIDAD

En la figura 2.5 se puede observar el mismo grafo del ejemplo anterior, con su representación en formato DIMACS.

```
p  max  5  7
n  1  s
n  5  t
a  1  2  10
a  1  3  9
a  1  4  12
a  2  3  8
a  3  1  21
a  3  5  2
a  4  5  12
```

(a) Fichero en formato DIMACS



(b) Grafo de ejemplo

Figura 2.5: Representación mediante el formato DIMACS

Capítulo 3

Construcción de grafos de prueba

Se han utilizado los grafos presentados en el capítulo anterior para medir tiempos de ejecución del algoritmo. En las siguientes secciones se muestran los resultados obtenidos sobre su construcción y una valoración de los mismos, que será relevante cuando analicemos los tiempos de ejecución de nuestra implementación del algoritmo *push-relabel*. En la sección 3.1 se introduce la representación interna utilizada para los grafos. En la sección 3.2 se explican con detalle los grafos de prueba utilizados. Por último, en la sección 3.3 se evalúan los tiempos de carga, desde fichero, de los grafos de prueba.

3.1. Representación de grafos

Existen numerosas estructuras de datos para almacenar grafos. Las tablas *hash* han demostrado ser muy eficientes en CPU [12]. Sin embargo, en las GPUs, la memoria está enfocada al trabajo con gráficos, por lo que las estructuras lineales como arrays y matrices se ven favorecidas por los mecanismos de optimización del sistema.

En nuestra representación hemos optado por usar una lista de adyacencia compacta, como la usada en otras publicaciones [7] [15]. Esta estructura está compuesta por dos listas, una de vértices y otra de arcos, como se muestra en la figura 3.1. La primera contiene los índices a los primeros arcos salientes de cada vértice. La segunda lista contiene los índices de los vértices a los que apunta cada arco.

Los índices utilizados en ambas listas nos sirven también para acceder a la información propia de cada vértice y arco. Además de la estructura del grafo, se almacena también la altura y el exceso de cada vértice, así como la capacidad y la capacidad residual de cada arco. Por último se crea una lista que almacena, por cada arco, el índice a su arco inverso; es decir, dado el arco que viaja del vértice u al vértice v, se guarda la posición del arco que viaja del vértice v al vértice u.

En la tabla 3.1 se muestran las variables que componen un grafo. Además, se muestra cuáles de ellas pertenecen a propiedades de vértices y cuáles a propiedades de arcos.

Hemos elegido este formato por sus ventajas a la hora de resolver el problema de flujo máximo. Con esta representación, la asignación de los diferentes hilos de ejecución es

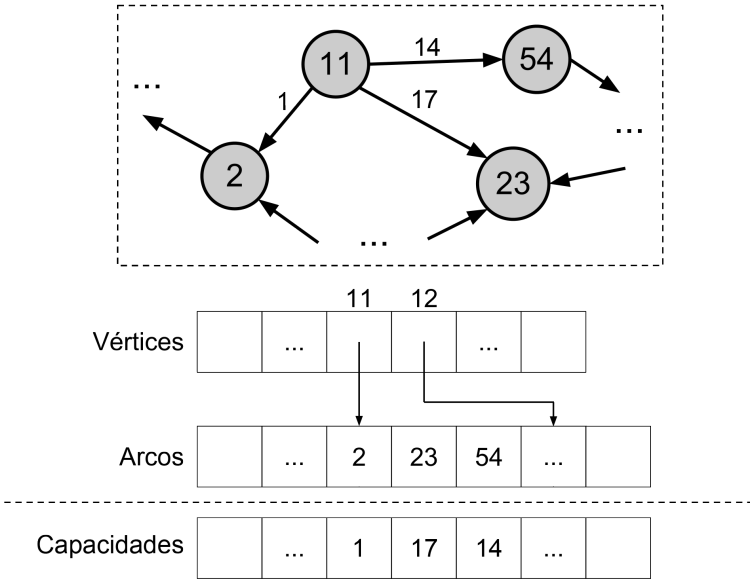


Figura 3.1: Estructura de datos utilizada para representar los grafos en CUDA

	Variable	Uso
Vértices	numv	Número de vértices que contiene el grafo
	v[ ]	Lista de vértices que contiene los índices al primer arco saliente
	h[ ]	Lista con la altura de cada vértice
	e[ ]	Lista con el exceso de cada vértice
Arcos	nume	Número de arcos que contiene el grafo
	target[ ]	Lista de arcos que contiene los índices de los vértices a los que apuntan
	c[ ]	Lista de capacidades de cada arco
	cr[ ]	Lista de capacidades residuales de cada arco
	einv[ ]	Lista de índices a los arcos inversos de cada arco

Tabla: 3.1: Estructura de datos utilizada para representar grafos residuales

directa. Además, dado que cada hilo necesita recorrer todos los arcos salientes del vértice que tenga asociado, y éstos están en posiciones de memoria contiguas, CUDA puede reunir todos estos accesos en una sola llamada para traer un bloque completo de datos y mejorar así el rendimiento global del algoritmo. Sin embargo, esta representación presenta varios inconvenientes. El primero es que el resto de accesos al grafo, como el acceso a los arcos inversos o a las alturas y excesos de los vértices vecinos, son a direcciones de memoria que, en general, no son contiguas. El segundo gran inconveniente es la necesidad de almacenar, de manera explícita, la información sobre los arcos inversos. Como veremos en los resultados de la sección 3.3, la estructura del grafo que se está leyendo afecta, en mayor o menor medida, al tiempo necesario para rellenar la lista de arcos inversos, que en algunos casos puede suponer hasta un 12 % del tiempo de carga total.

3.2. Grafos de prueba

Para realizar la batería de pruebas se han utilizado grafos dirigidos acíclicos (*ADG*), *RMFGEN* y *Washington-RLG* como los vistos en el capítulo anterior. En concreto, se han utilizado los grafos que se presentan en la tabla 3.2. La capacidad de los arcos se ha limitado a un valor máximo de 10000 en todos los casos. Para las pruebas, se han creado 5 grafos por cada tipo y tamaño a fin de reducir el margen de error de las mediciones.

Tipo	Parámetros	Número de vértices	Número de arcos
ADG		2000	2001000
		4000	8002000
		6000	18003000
RMFGEN (Long)	$a = 24$ $b = 192$	110592	533952
	$a = 28$ $b = 224$	175616	852208
	$a = 32$ $b = 256$	262144	1276928
RMFGEN (Wide)	$a = 36$ $b = 36$	46656	226800
	$a = 48$ $b = 48$	110592	541440
	$a = 64$ $b = 64$	262144	1290240
Washington-RLG (Long)	$w = 512$ $l = 1024$	524290	1572352
	$w = 768$ $l = 1280$	983042	2948352
	$w = 1024$ $l = 1536$	1572866	4717568
Washington-RLG (Wide)	$w = 512$ $l = 512$	262146	785920
	$w = 768$ $l = 768$	589826	1768704
	$w = 1024$ $l = 1024$	1048578	3144704

Tabla: 3.2: Grafos de prueba utilizados en las pruebas

3.3. Tiempos de carga

Los primeros resultados que evaluamos son los tiempos promedios de carga de cada grafo. Se considera cargar un grafo a la acción de leer el fichero correspondiente y transformar el contenido en la estructura interna en GPU que requiere nuestra implementación del algoritmo.

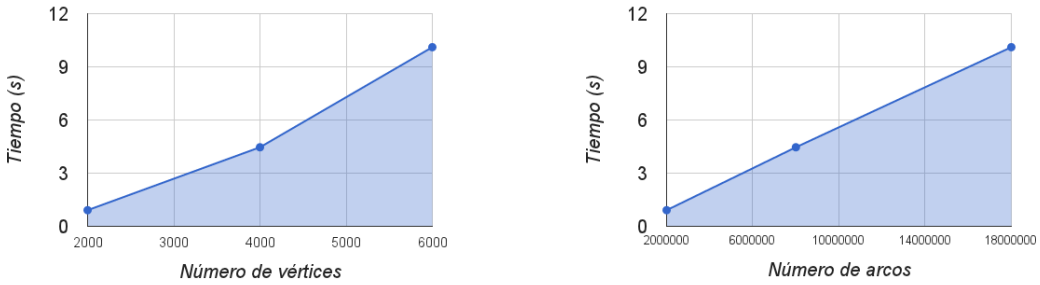
Hay que tener en cuenta que se ha intentado aprovechar el formato de cada uno de los grafos para optimizar los tiempos de carga, por lo que no se pueden extraer conclusiones definitivas si se comparan los tiempos de creación de grafos de diferente tipo. De aquí en adelante, se considera que los tiempos de carga de un grafo deben ser, como mínimo, proporcionales a su número de arcos dado que es necesario leer la capacidad de todos ellos para obtener una representación completa del grafo.

3.3.1. Tiempos de carga de matrices de adyacencia

Este tipo de representación se ha utilizado únicamente para grafos pequeños debido a la gran carga en memoria que supone almacenar la matriz completa. La principal ventaja de este formato es la facilidad de extraer toda la información necesaria, en poco tiempo.

Para hacer las pruebas, se han manejado *ADG's* con entre 2000 y 6000 vértices y entre 2 y 18 millones de arcos, como aparece en la tabla 3.2.

La figura 3.2 muestra los tiempos de carga de los *ADG's*. Como se puede observar, el tiempo de carga de un grafo es directamente proporcional a su número de arcos. La principal limitación de este tipo de grafos no viene impuesta por su tiempo de carga, sino por la representación utilizada. Como se vio en la sección 2.1, se ha usado una matriz de adyacencia para representar los *ADG's*, lo cual impone un límite en el tamaño máximo de este tipo de grafos, dado que, el tamaño de los ficheros crece de forma cuadrática respecto al número de vértices.



(a) Tiempo de carga según el número de vértices (b) Tiempo de carga según el número de arcos

Figura 3.2: Tiempos de carga para los *ADG's*

3.3.2. Tiempos de carga de grafos en formato DIMACS

Como se ha visto en la sección 2.4, el formato DIMACS solamente almacena la información pertinente de cada arco. Esta característica hace del formato un sistema eficiente para almacenar grafos dispersos. Por ello, se ha utilizado este formato para almacenar *RMFGEN's* y *Washington-RLG's*. A continuación se analizan y comparan los tiempos de carga obtenidos para cada uno.

Tiempos de carga de *RMFGEN's*

Para la valoración de los tiempos de carga de los *RMFGEN's* se ha clasificado este tipo de grafos en dos categorías diferentes según sus parámetros. Los *RMFGEN's Long* han sido creados utilizando valores altos del parámetro b. Esto significa que este tipo de grafos están compuestos por un gran número de rejillas. Los *RMFGEN's Wide* se han construído usando el mismo valor para el parámtro a y b, formando grafos de estructura cúbica.

Se han utilizado los *RMFGEN* de la tabla 3.2 que tienen entre 46656 y 262144 vértices y entre 226800 y 1290240 arcos.

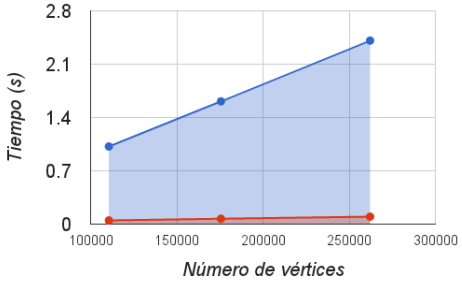
En la figura 3.3 se puede observar cómo los tiempos de carga totales –de color azul– para ambas categorías de *RMFGEN's* son similares, obteniendo tiempos de carga lineales respecto al número de arcos de los grafos. Hay que tener en cuenta que el tamaño de los ficheros que contienen este tipo de grafos crece también de forma proporcional al número de arcos del grafo. Como se puede observar, el tiempo de carga de los grafos es también proporcional a su número de vértices, esto es debido a que, por construcción, los *RMFGEN's* tienen un número de arcos directamente proporcional a su número de vértices. Por último, puede parecer que el tiempo de carga de los *RMFGEN's Wide* es menor que el de los *RMFGEN's Long*, pero hay que tener en cuenta que los dos primeros *RMFGEN's Wide* tienen menor número de vértices y arcos que los dos primeros *RMFGEN's Long*.

En las gráficas de color rojo se muestra el tiempo que se ha consumido rellenando la estructura de los arcos inversos. Gracias a la regularidad de los *RMFGEN's* es siempre posible determinar el número de arcos entrantes a un vértice dado conociendo su ubicación dentro del grafo. Es por esta característica por la que se ha conseguido minimizar los cálculos necesarios para rellenar esta información.

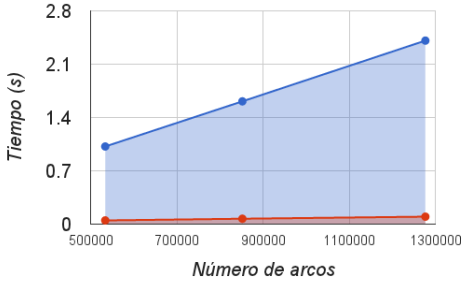
Tiempos de carga de *Washington-RLG's*

Al igual que con los *RMFGEN's*, hemos clasificado los *Washington-RLG's* en dos categorías diferentes según el valor de los parámetros utilizados para su construcción. Los *Washington-RLG's Long* se han creado usando un parámetro l alto, formando así grafos con una gran distancia entre el vértice fuente y el destino. Los *Washington-RLG's Wide* se han construído usando los mismos valores para los parámetros w y l. Este tipo de grafos tienen una estructura cuadrada.

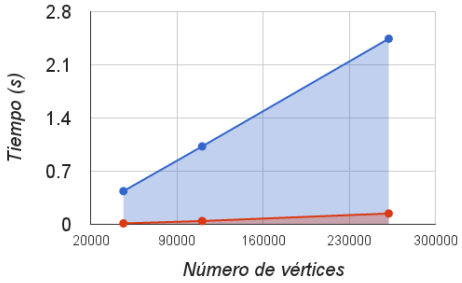
Para realizar las pruebas se han utilizado los *Washington-RLG* de la tabla 3.2 que tienen entre 262146 y 1572866 vértices y entre 785920 y 4717568 arcos.



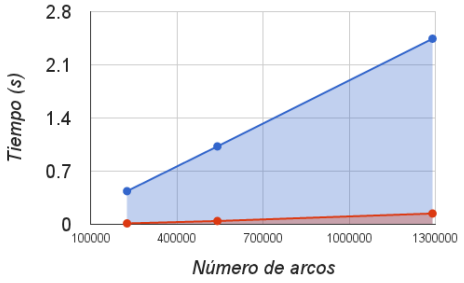
(a) Tiempo de carga de los grafos *RMFGEN Long* según el número de vértices



(b) Tiempo de carga de los grafos *RMFGEN Long* según el número de arcos



(c) Tiempo de carga de los grafos *RMFGEN Wide* según el número de vértices

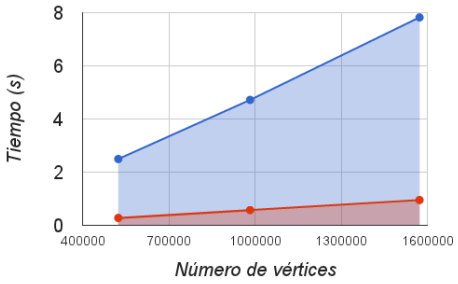


(d) Tiempo de carga de los grafos *RMFGEN Wide* según el número de arcos

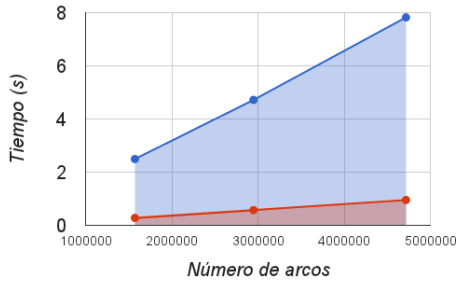
Figura 3.3: Tiempos de carga para los *RMFGEN's*

La figura 3.4 muestra, en azul, los tiempos de carga de los *Washington-RLG's*. Como sucedía con los *RMFGEN's*, el número de vértices y arcos de los *Washington-RLG's* crece de forma proporcional por lo que se pueden valorar los tiempos de carga del grafo respecto a sus vértices o arcos indistintamente.

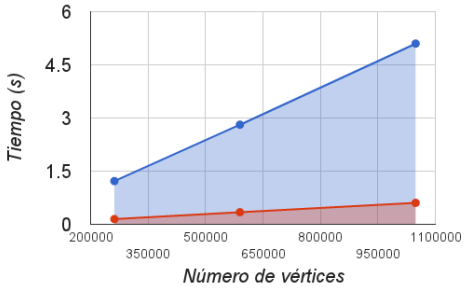
En rojo se muestra la proporción del tiempo de carga total que se dedica a rellenar la información de los arcos inversos. En comparación con los *RMFGEN's*, el tiempo consumido para esta tarea en los *Washington-RLG's* es, respecto al tiempo de carga total, proporcionalmente mayor. Este aumento en el tiempo de carga de los arcos inversos es debido a la estructura de los grafos leídos. Mientras que en los *RMFGEN's* es siempre posible determinar el número de arcos entrantes a un vértice dado, conocida su posición dentro del grafo, en los *Washington-RLG's* es necesario almacenar información adicional por cada columna de vértices. La información almacenada nos permite conocer los vértices de la columna anterior que tienen arcos a los de la actual. Sin embargo, aún con esta información, es necesario hacer pequeñas búsquedas entre los arcos salientes de los vértices de dicha columna anterior para determinar de forma precisa el arco inverso y almacenar la información en la lista correspondiente.



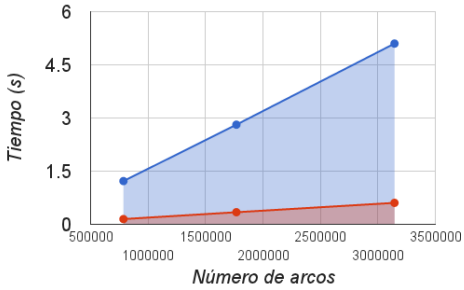
(a) Tiempo de carga de los grafos *Washington-RLG Long* según el número de vértices



(b) Tiempo de carga de los grafos *Washington-RLG Long* según el número de arcos



(c) Tiempo de carga de los grafos *Washington-RLG Wide* según el número de vértices



(d) Tiempo de carga de los grafos *Washington-RLG Wide* según el número de arcos

Figura 3.4: Tiempos de carga para los *Washington-RLG's*

Capítulo 4

Implementación

En las siguientes secciones se detalla la implementación de la versión paralela del algoritmo de *push-relabel*. La sección 4.1 introduce CUDA, la arquitectura sobre la cual se ha llevado a cabo la implementación. En la sección 4.2, se señalan los cambios necesarios para transformar el algoritmo clásico de *push-relabel* en una versión paralela. En la sección 4.3 se introduce el renombrado global, una optimización que reasigna la altura de los vértices según una heurística. Por último, en la sección 4.4, se muestran los detalles de implementación del algoritmo.

4.1. CUDA

CUDA es una plataforma desarrollada por NVIDIA para la creación de programas paralelos de carácter general enfocados a su ejecución en tarjetas gráficas. Para utilizar CUDA, los programadores deben codificar sus algoritmos en una extensión del lenguaje C diseñada para ello. Dicha extensión permite a los desarrolladores escribir código dirigido a la GPU en forma de funciones llamadas **kernels**.

Está en manos del programador decidir el número de hilos que se lanzan para cada kernel y cómo deben organizarse. CUDA permite agrupar los hilos en bloques, a su vez, los bloques pueden formar rejillas, como se muestra en la figura 4.1. Por defecto, los hilos y los bloques se organizan en listas de una sola dimensión, pero como se puede observar, existe la posibilidad de distribuirlos en forma de matrices de hasta tres dimensiones. Tanto los hilos, como los bloques y las rejillas, tienen asignado un identificador en el momento de su ejecución. Gracias a esto, el programador puede distribuir la carga de trabajo de su algoritmo entre todos los hilos disponibles, ejecutando las mismas instrucciones sobre diferentes datos, o incluso ejecutando diferentes secciones de código por cada hilo.

Internamente, cada bloque es enviado a un multiprocesador para su ejecución. El multiprocesador cuenta con una pequeña cantidad de memoria compartida por todos los hilos del bloque. A su vez, cada hilo tiene disponible su propia memoria privada para almacenar el valor de las variables asignadas durante su ejecución. Por último, todos los hilos comparten una memoria global, más grande, pero a la vez más lenta. La figura 4.2 resume la

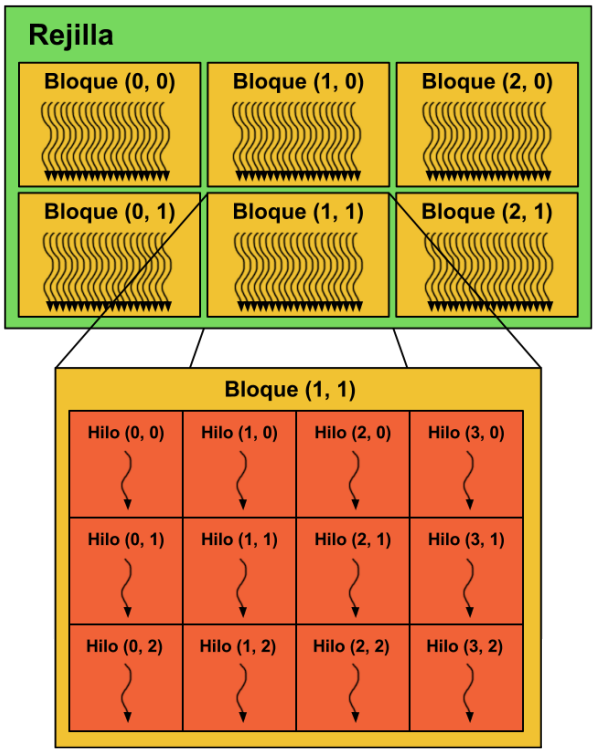


Figura 4.1: Estructura de los hilos en CUDA

distribución de los bloques y la jerarquía de memoria.

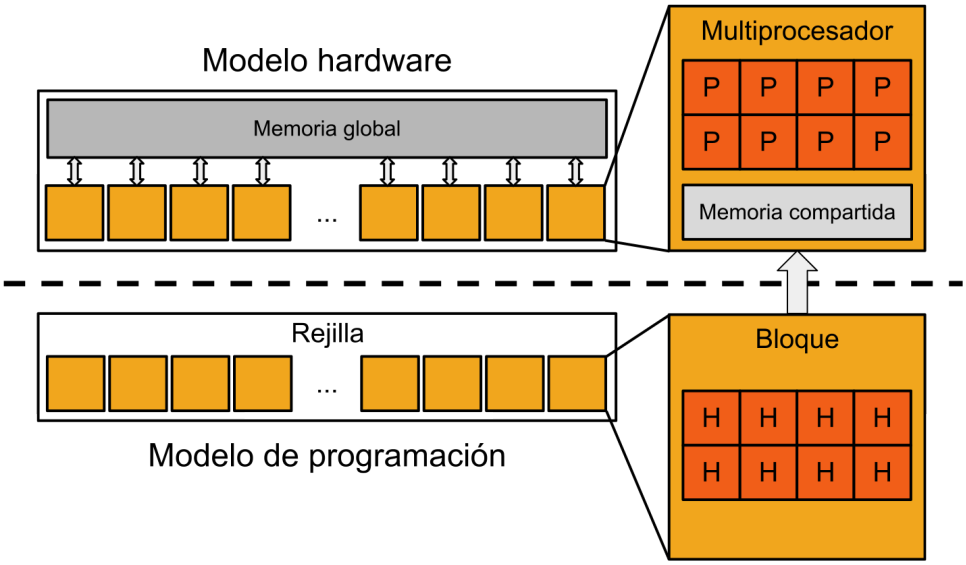


Figura 4.2: Modelo hardware y de programación de CUDA

CUDA cuenta con instrucciones de sincronización, pero sólo son efectivas entre hilos de un mismo bloque, limitando así su alcance. Además, a partir de la versión 1.1, CUDA soporta la ejecución de instrucciones aritméticas sencillas, de manera atómica.

Debido a la arquitectura de las GPUs modernas, los programas CUDA ven afectado su rendimiento de forma negativa cuando los hilos de un mismo bloque ejecutan instrucciones diferentes. Esto puede suceder, por ejemplo, cuando se introducen instrucciones condicionales en el código. Además, las operaciones de memoria tienen un bajo rendimiento cuando las direcciones accedidas no son contiguas. Estos puntos son clave a la hora de desarrollar programas CUDA y deben tenerse en cuenta si se desean obtener soluciones eficientes.

4.2. Algoritmo de *push-relabel* paralelo

Existen múltiples versiones paralelas del algoritmo de *push-relabel*. El algoritmo que nosotros hemos implementado fue ideado originalmente por Hong [9] en 2008. El hecho de que no sea necesario introducir mecanismos de sincronización, tales como cerrojos o semáforos, lo hace idóneo para su implementación en CUDA, ya que esta plataforma no proporciona ningún soporte a este tipo de mecanismos, aunque sí ofrece un muy alto rendimiento en la ejecución de programas paralelos.

En el artículo de Hong y posteriores [8][14], se propone introducir una serie de cambios sobre el algoritmo original de *push-relabel* para poder ejecutarlo de forma paralela. La prin-

cipal modificación consiste en enviar el flujo desde un vértice al vecino más bajo, en lugar de enviarlo a uno cualquiera de sus vecinos situado a una altura menor. Además, se requiere ejecutar una serie de operaciones aritméticas de forma atómica para garantizar la corrección del algoritmo.

El algoritmo está pensado para que se lance un hilo por cada vértice o grupo de vértices, encargándose cada uno de ellos de aplicar las operaciones *push* o *relabel*, según sea posible hacerlo. Un hilo principal se encarga de la inicialización de las variables del grafo antes de comenzar la fase paralela del programa. En la sección 4.4 se amplían los detalles de la implementación.

Pese a los cambios aplicados, puede probarse [9] que el programa es correcto y que el algoritmo finaliza después de, como mucho, $O\left(|V|^2 |E|\right)$ operaciones de *push* o *relabel*.

4.3. Renombrado global

En su publicación original, Goldberg y Tarjan [5] ya mencionaron la posibilidad de que, a lo largo de la ejecución del algoritmo, hubiese momentos en los que se realizan muchas operaciones *relabel* innecesarias. Para solucionar esta situación, Cherkassky y Goldberg [1], propusieron el uso de heurísticas y mejorar así el rendimiento del algoritmo de *push-relabel*. Concretamente, plantearon tener en cuenta la distancia de cada vértice al vértice fuente y destino. La heurística sugerida se denominó **renombrado global**, y consiste en actualizar las alturas de todos los vértices –menos el fuente y el destino– realizando una búsqueda primero en anchura inversa –esto es, utilizando arcos entrantes y no salientes– en el grafo residual. Esta búsqueda se encarga de calcular las distancias $d(v, s)$ y $d(v, t)$, para cada vértice v. La nueva altura asignada al vértice v es $\min(d(v, s) + |V|, d(v, t))$.

La idea principal detrás de esta heurística es la de actualizar la altura de los vértices de forma que, desde cada vértice conectado con el destino, haya un camino directo por el que enviar el exceso de flujo sin tener que aplicar operaciones *relabel* adicionales. Los vértices a distancia 1 del vértice destino pueden enviarle flujo en una sola operación *push* si los situamos a altura 1, los vértices a distancia 2 deben pasar, utilizando el camino más corto, por los vértices con altura 1 para enviar su exceso de flujo, por lo que son situados a altura 2 y así con el resto de vértices. De igual forma, aquellos vértices que no tienen conexión con el vértice destino, tratan de devolver su exceso de flujo hacia el vértice fuente sumando $|V|$ a la distancia que los separa de dicho vértice fuente.

Tal y como se explica en el artículo de Goldberg y Tarjan, es complicado determinar cuándo debería realizarse esta optimización y por lo tanto se sugiere aplicarla después de que el algoritmo haya realizado un número arbitrario de operaciones *relabel*.

Como Hong demostró en su trabajo [9], para que el algoritmo, en su versión paralela, proporcione resultados correctos, cada arco residual del grafo (u, v) debe cumplir que $h(u) \leq h(v) + 1$. Si el proceso de hacer *push* y *relabel* es interrumpido sin que haya finalizado, para hacer el renombrado global, pueden haberse formado arcos residuales que no cumplan esta propiedad y para los que sin embargo se cumpla que $h(u) > h(v) + 1$. Este tipo de arcos residuales desaparecen de manera automática, a lo largo de la ejecución del algoritmo, al

aplicar una operación *push* sobre ellos. Sin embargo, para poder considerar el renombrado global, en esta versión del algoritmo, es necesario arreglar estos arcos residuales prohibidos, de forma explícita. Para ello, se recorren todos los arcos residuales que no cumplan la propiedad y se les aplica la operación *push* mencionada. A este paso, que prepara el grafo residual para el renombrado global lo hemos llamado cancelación de las violaciones de envío de flujo.

4.3.1. Búsqueda primero en anchura

Como se ha dicho, es necesario realizar más de una búsqueda primero en anchura para aplicar la técnica de renombrado global. Empíricamente se ha comprobado que este proceso puede llegar a ocupar hasta el 50 % del tiempo de ejecución total del algoritmo, dependiendo de la implementación utilizada [16].

Nosotros hemos decidido implementar la búsqueda primero en anchura en GPU usando CUDA ya que se ha comprobado que el tiempo necesario para mover los datos entre CPU y GPU penaliza el rendimiento global del algoritmo.

El problema de la búsqueda primero en anchura sobre grafos utilizando CUDA ha sido ampliamente cubierto en diferentes publicaciones [7] [10] [11]. En la sección 4.4.5 se explica con detalle nuestra implementación.

4.4. Detalles de implementación

A continuación se muestra el código de la implementación realizada para la versión paralela del algoritmo de *push-relabel*. Se incluyen las funciones y los kernels desarrollados.

4.4.1. Algoritmo *push-relabel*

En el algoritmo 4.1 aparece el código de la función principal. Primero se inicializan los valores del grafo para después comenzar el bucle principal del algoritmo. La llamada al kernel de la línea 6 se encarga de aplicar las operaciones *push* y *relabel* un número determinado de veces. Por último, la función llamada en la línea 7 se encarga de aplicar la optimización de renombrado global. En la presentación que hacemos de los algoritmos, se han simplificado algunos detalles propios de CUDA a fin de hacer el código más comprensible. En todo caso, el código de los algoritmos explicados en esta sección se encuentra en el Apéndice B.

4.4.2. Inicialización

El algoritmo 4.2 muestra el código encargado de la inicialización de las variables del grafo. En las líneas 2-6 se asigna un valor 0 a las alturas y los excesos de cada vértice, a excepción del vértice fuente, cuya altura se inicializa al número de vértices del grafo. Las líneas 8-10 corresponden a la inicialización de las capacidades residuales de los arcos. Por último, se inicia el cálculo del flujo máximo, enviando tanto flujo como sea posible desde el

```
1  void pushrelabel(Grafo *g) {
2    inicializar(g);
3    bool terminado = false;
4    while (!terminado) {
5      terminado = true;
6      kernel_pushrelabel(&terminado, g);
7      renombrado_global(g);
8    }
9  }
```

Algoritmo 4.1: Versión paralela del algoritmo de *push-relabel*

vértice fuente. El código encargado de este último paso se compone de las instrucciones 12-17. Obsérvese que esto supone: (i) dejar a 0 los arcos salientes del vértice fuente, (ii) añadir arcos residuales entrantes al vértice fuente, con capacidad residual igual al flujo enviado, (iii) registrar el exceso de flujo que pasan a tener los vértices accesibles desde el vértice fuente y (iv) registrar el exceso de flujo negativo de la fuente.

```
1  void inicializar(Grafo *g) {
2    for (int i = 0; i < G->numv; i++) {
3      g->h[i] = 0;
4      g->e[i] = 0;
5    }
6    g->h[FUENTE] = g->numv;
7
8    for (int i = 0; i < G->nume; i++) {
9      g->cr[i] = g->c[i];
10   }
11
12   for (int i = g->v[FUENTE]; i < g->v[FUENTE + 1]; i++) {
13     g->cr[i] -= g->c[i];
14     g->cr[g->einv[i]] += g->c[g->einv[i]];
15     g->e[g->target[i]] = g->c[i];
16     g->e[FUENTE] -= g->c[i];
17   }
18 }
```

Algoritmo 4.2: Inicialización del grafo residual

4.4.3. Kernel *push-relabel*

El algoritmo 4.3 muestra el kernel que ejecuta cada hilo en paralelo. La variable *idx* corresponde con el identificador único que cada hilo tiene en CUDA. Dado que se lanza un hilo por cada vértice, *idx* se usa como el índice del vértice asociado al hilo. El bucle principal, compuesto de las instrucciones 6-37 se repite un número determinado de veces.

Se ha añadido el contador *ciclo* para que, transcurrido un periodo determinado de tiempo, se detenga el proceso de *push-relabel* y se ejecute la optimización del renombrado global.

La comparación de la línea 7 determina si el vértice está activo, es decir, tiene un exceso de flujo positivo y una altura válida. Obsérvese que la fuente no está activa. Si hay algún vértice activo entonces el algoritmo no ha finalizado y por lo tanto asignamos el valor booleano *false* a la variable *terminado*. Es importante observar que la comparación está dentro del bucle principal dado que una operación *push* o *relabel* puede provocar que un vértice vecino pase de estado inactivo a activo. El código comprendido entre las líneas 15 a 23 se encarga de recorrer los vecinos del vértice asociado al hilo, y determinar cuál es el más bajo de entre los conectados con él en el grafo residual. Las líneas 25-31 se encargan de enviar flujo desde el vértice actual al vecino más bajo, siempre que no estén a la misma altura. Como se puede observar, es necesario que las cuatro instrucciones implicadas en la operación *push* (líneas 27-30) sean instrucciones atómicas para que el algoritmo sea correcto. Por último, las líneas 32-34 corresponden a la operación *relabel*, que se encarga de levantar el vértice una unidad respecto a la mínima altura de entre sus vecinos.

4.4.4. Renombrado global

El algoritmo 4.4 muestra la función encargada de aplicar la optimización de renombrado global. Como se detalló en la sección 4.3, primero es necesario cancelar las violaciones de envío de flujo, provocadas al detener el kernel de *push-relabel* sin que éste haya finalizado. Una vez arreglado el estado del grafo, se reinician las alturas de cada vértice (asignando a cada uno el valor del mayor entero representable en la plataforma) para recalcularlas a partir de la heurística. La función llamada en la línea 6 realiza una búsqueda en anchura para calcular la distancia de todos los vértices respecto al destino. Se repite la misma operación en la línea 7 midiendo esta vez la distancia al vértice fuente.

El kernel encargado de cancelar las violaciones de flujo aparece en el algoritmo 4.5. Cada hilo está asociado a un vértice y se encarga de comprobar si existe algún vecino más de una unidad por debajo y al cual se le esté enviando flujo, es decir, se comprueba si existe algún envío de flujo prohibido a algún vértice vecino. Si la condición se cumple entonces se ejecutan las instrucciones de las líneas 10-13, encargadas de anular el envío de flujo, aplicando para ello una operación *push*.

4.4.5. Búsqueda primero en anchura

Como se dijo en la sección 4.3.1, se ha decidido implementar la búsqueda primero en anchura sobre GPU en lugar de hacerlo sobre CPU, debido al alto coste de enviar los datos de un dispositivo a otro.

Hemos estudiado la implementación de las diferentes alternativas para realizar la búsqueda primero en anchura sobre GPU. Ni la propuesta de Hong et al. [10] –que trata de maximizar la eficiencia de la búsqueda optimizando el uso eficiente de *warps*– ni la implementación que se usa en el artículo de Hussein et al. [11] –que hace uso de la primitiva *scan* para distribuir uniformemente el trabajo de los hilos en cada iteración– han resultado

```
1  __global__
2  void kernel_pushrelabel(bool &terminado, Grafo *g) {
3    int idx = blockIdx.x * blockDim.x + threadIdx.x;
4    if (idx != FUENTE && idx != DESTINO) {
5      int ciclo = NUM_CICLOS;
6      while (ciclo > 0) {
7        if (g->e[idx] > 0 && g->h[idx] < g->numv) {
8          terminado = false;
9          int ep = g->e[idx];
10         int min_v = INT_MIN;
11         int min_h = INT_MAX;
12         int min_e = INT_MAX;
13         int primer_arco = g->v[idx];
14         int ultimo_arco = g->v[idx + 1];
15         for (int i = primer_arco; i < ultimo_arco; i++) {
16           int vp = g->target[i];
17           int hp = g->h[vp];
18           if (hp < min_h && g->cr[i] > 0) {
19             min_v = vp;
20             min_h = hp;
21             min_e = i;
22           }
23         }
24
25         if (g->h[idx] > min_h) {
26           int d = min(ep, g->cr[min_e]);
27           atomicSub(&g->cr[min_e], d);
28           atomicAdd(&g->cr[g->einv[min_e]], d);
29           atomicAdd(&g->e[min_v], d);
30           atomicSub(&g->e[idx], d);
31         }
32         else {
33           g->h[idx] = min_h + 1;
34         }
35       }
36       ciclo -= 1;
37     }
38   }
39 }
```

Algoritmo 4.3: Kernel *push-relabel*

```
1  void renombrado_global(Grafo *g) {
2    cancelar_violaciones_flujo(g);
3    reiniciar_alturas(g);
4    g->h[FUENTE] = g->numv;
5    g->h[DESTINO] = 0;
6    bfs_inverso(g, DESTINO);
7    bfs_inverso(g, FUENTE);
8  }
```

Algoritmo 4.4: Implementación del renombrado global

```
1  void cancelar_violaciones_flujo(Grafo *g) {
2    int idx = blockIdx.x * blockDim.x + threadIdx.x;
3
4    if (idx < g->numv) {
5      int primer_arco = g->v[idx];
6      int ultimo_arco = g->v[idx + 1];
7      for (int i = primer_arco; i < ultimo_arco; i++) {
8        int v = g->target[i];
9        if (g->h[idx] > g->h[v] + 1) {
10         g->e[idx] = g->e[idx] - g->cr[i];
11         g->e[v] = g->e[v] + g->cr[i];
12         g->cr[g->einv[i]] = g->cr[g->einv[i]] + g->cr[i];
13         g->cr[i] = 0;
14       }
15     }
16   }}
```

Algoritmo 4.5: Función encargada de cancelar los envíos de flujo prohibidos

```
1  void reiniciar_alturas(Grafo *g) {
2    int idx = blockIdx.x * blockDim.x + threadIdx.x;
3    if (idx > 0 && idx < g->numv) {
4      g->h[idx] = INT_MAX;
5    }
6  }
```

Algoritmo 4.6: Se reinician las alturas de los vértices para reasignarlas después, utilizando una búsqueda en anchura

eficientes. La principal causa del bajo rendimiento de estos algoritmos es que están ideados para aprovechar al máximo la capacidad de la GPU cuando los vértices están, en general, lejos del vértice del cual se quiere calcular su distancia. Sin embargo, en los grafos utilizados para realizar las pruebas suele producirse el caso contrario, los vértices suelen estar cerca del vértice destino y fuente. Por todo ello hemos optado por implementar la versión de la búsqueda presentada en el artículo de Pawan et al. [7] y que se detalla a continuación.

El algoritmo asume que se han inicializado las alturas de todos los vértices, marcando con el valor INT_MAX todos aquellos cuya altura se quiere actualizar. El algoritmo 4.7 es el encargado de realizar la búsqueda. En cada iteración del bucle se consideran todos los vértices cuya altura sea igual a *nivel*. Para todos ellos, se llama a la función para GPU *kernel_bfs*, encargada de actualizar la altura de los vecinos no visitados y darles el valor $nivel + 1$. Este paso se realiza mediante el kernel que aparece detallado en el algoritmo 4.8. En cada iteración del bucle aumentamos el valor de *nivel* en uno para visitar los vecinos que hemos actualizado en la iteración anterior.

```
1 void bfs_inverso(Grafo *g, int primer_vertice) {
2   bool terminado = false;
3   int nivel = g->h[primer_vertice];
4   while (!terminado) {
5     terminado = true;
6     kernel_bfs(g, nivel, &terminado);
7     nivel += 1;
8   }
9 }
```

Algoritmo 4.7: Función que realiza la búsqueda en anchura.

El kernel 4.8 es el encargado de realizar las actualizaciones en cada iteración. Se lanzan tantos hilos como vértices haya en el grafo, y cada uno comprueba si el vértice que tiene asociado pertenece o no al nivel actual. Si es así, se recorren todos sus vecinos en el bucle de las líneas 5-11 y, por cada uno de ellos, se comprueba que no haya sido visitado –lo cual ocurre cuando el vértice conserva todavía su valor de inicialización en la búsqueda– y que exista un arco residual del vecino al vértice. En una búsqueda en anchura normal, la segunda comprobación sería innecesaria, pero puesto que estamos realizando una búsqueda inversa, lo que intentamos es calcular la distancia de todos los vértices al inicial y no al contrario, del vértice inicial al resto. Por último, si encontramos un vecino no visitado, y con un arco válido, actualizamos su altura al valor $nivel + 1$. Hay que tener en cuenta que varios hilos pueden modificar la altura de un mismo vértice a la vez, pero dado que el kernel se ejecuta para un nivel determinado, todos los hilos escribirán el mismo valor y por lo tanto no hay peligro de sobreescribir con valores incorrectos.

```
1  void kernel_bfs(Grafo *g, int nivel, bool &terminado) {
2    int idx = blockIdx.x * blockDim.x + threadIdx.x;
3
4    if (idx < g->numv && g->h[idx] == nivel) {
5      for (int i = g->v[idx]; i < g->v[idx + 1]; i++) {
6        int j = g->target[i];
7        if (g->h[j] == INT_MAX && g->cr[g->einv[i]] > 0) {
8          terminado = false;
9          g->h[j] = nivel + 1;
10       }
11     }
12   }
13 }
```

Algoritmo 4.8: Kernel que realiza la búsqueda en anchura para un solo nivel del grafo.

Capítulo 5

Resultados sobre la ejecución del algoritmo

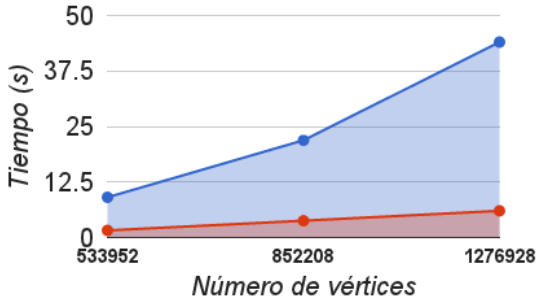
En este capítulo se muestran, analizan y comparan los tiempos de ejecución de la versión secuencial y paralela del algoritmo de *push-relabel*. En la sección 5.1 se muestran los tiempos de ejecución de la versión secuencial, haciendo énfasis en el tiempo utilizado para la aplicación del renombrado global. En la sección 5.2 se presentan los tiempos de ejecución para la versión paralela del algoritmo. Al igual que con la versión secuencial, se analiza el tiempo consumido en la ejecución del renombrado global. Por último, la sección 5.3 muestra la comparativa entre los resultados obtenidos con la versión secuencial y la versión paralela.

5.1. Tiempos de ejecución de la versión secuencial

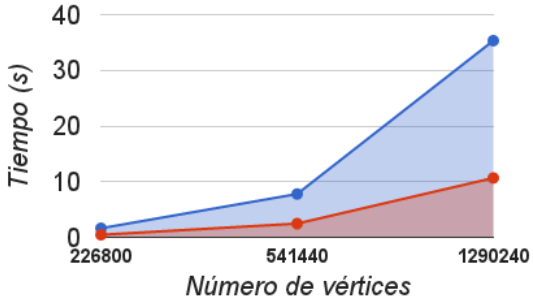
Hemos implementado la versión secuencial del algoritmo original de Goldberg y Tarjan para medir tiempos y comparar los resultados con la versión paralela. A fin de hacer las comparaciones más fiables, hemos implementado el renombrado global usando una búsqueda primero en anchura secuencial en CPU. En esta sección se muestran los tiempos promedios de ejecución. Las gráficas azules muestran el tiempo total de ejecución del algoritmo y las gráficas rojas el tiempo dedicado al renombrado global.

En la figura 5.1 se muestran los tiempos de ejecución obtenidos para *RMFGEN's*. Podemos observar cómo el tiempo de ejecución crece rápidamente a medida que aumenta el número de vértices del grafo. El tiempo utilizado para el renombrado global es mayor en los *RMFGEN Wide*. Como se vió en el capítulo 3, la principal diferencia de estos grafos con los *RMFGEN Long* es que los primeros tienen menos rejillas, pero las que tienen son más grandes. Ésta es la causa del mayor tiempo registrado en los *RMFGEN Wide* ya que la búsqueda primero en anchura se ejecuta más lentamente entre los vértices de una misma rejilla –por cómo están estructurados los vértices– que en el cambio de rejillas.

Como se puede observar en la figura 5.2, se han omitido los tiempos de ejecución para los *Washington-RLG's* más grandes dado que eran demasiado altos. Si comparamos estos resultados con los obtenidos con *RMFGEN's*, se puede apreciar que los tiempos de ejecución



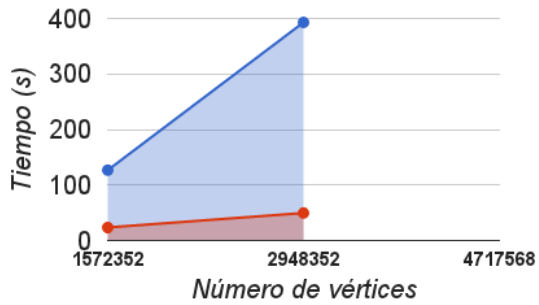
(a) *RMFGEN* Long



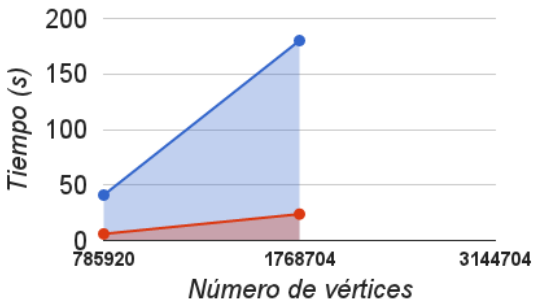
(b) *RMFGEN* Wide

Figura 5.1: Tiempo de ejecución promedio para la implementación secuencial del algoritmo sobre *RMFGEN's*

de estos últimos son menores. Esto es debido a que los *Washington-RLG's* utilizados tienen, en su mayoría, un número mayor de vértices y arcos que los *RMFGEN's*.



(a) *Washington-RLG* Long



(b) *Washington-RLG* Wide

Figura 5.2: Tiempo de ejecución promedio para la implementación secuencial del algoritmo sobre *Washington-RLG's*

Por último, la figura 5.3 muestra cómo los *ADG's* presentan tiempos más regulares respecto al número de vértices, tanto en la ejecución total como en el cálculo del renombrado global. Si comparamos este último resultado con los obtenidos en el resto de grafos podemos observar que el tiempo dedicado en los *ADG's* para el renombrado global es menor, tanto en proporción con el tiempo total de ejecución como en mediciones de tiempo absolutas. Este fenómeno puede ser explicado por la estructura de los grafos utilizados. Como todos los vértices están conectados entre sí, ya sea de forma directa o inversa, la distancia entre el vértice destino y el resto de vértices del grafo se mantiene potencialmente baja durante la ejecución del algoritmo.

Cabe señalar el hecho de que, en media, se dedica un 17 % a realizar el renombrado

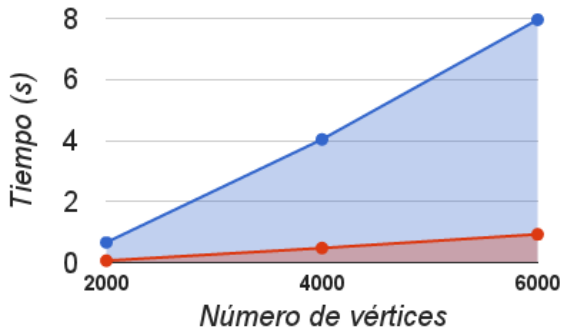


Figura 5.3: Tiempo de ejecución promedio para la implementación secuencial del algoritmo sobre *ADG's*

global. Este dato nos indica que, pese a utilizar las búsquedas en anchura sobre CPU, la mayor parte del tiempo se utiliza para la ejecución del algoritmo base.

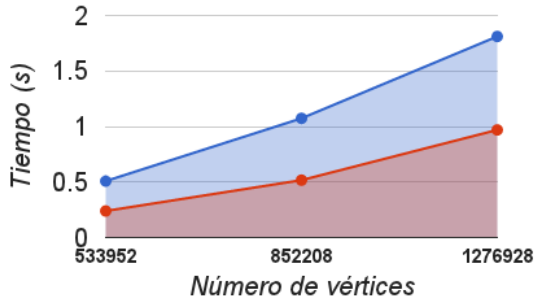
5.2. Tiempos de ejecución de la versión paralela

En esta sección se muestran los tiempos promedios de ejecución de nuestra implementación de la versión paralela del algoritmo. Se muestra, además, la cantidad de tiempo que se emplea para el renombrado global. Como en la sección anterior, se han dividido los resultados según el tipo de grafos utilizados.

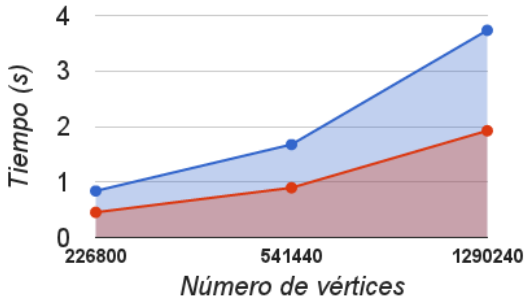
En los primeros resultados, presentados en la figura 5.4 y obtenidos sobre los *RMFGEN's*, se puede observar cómo los tiempos de ejecución son más lentos sobre los *RMFGEN Wide* que sobre los *RMFGEN Long*, pese a tener estos últimos mayor número de vértices. Esto es debido a que la estructura de los *RMFGEN Wide* provoca una propagación más lenta del flujo a lo largo de la ejecución del algoritmo debido a que las rejillas son más amplias. En ambas categorías de *RMFGEN's* se puede observar cómo el tiempo consumido en el renombrado global llega a alcanzar hasta el 50 % del tiempo total de ejecución.

En la figura 5.5 se muestran los tiempos de ejecución sobre los *Washington-RLG's*, los cuales son, en términos generales, más lentos que los obtenidos sobre *RMFGEN's*. Pese a que la estructura de los primeros favorece la propagación del flujo, hay que tener en cuenta que el número de vértices que se maneja es mucho mayor. Al igual que con los *RMFGEN's*, el tiempo utilizado para el renombrado global supera el 50 % del tiempo total de ejecución.

Finalmente, en la figura 5.6 se muestran los resultados conseguidos para los *ADG's*. Como se puede observar, estos siguen un comportamiento diferente respecto a los tiempos del resto de grafos utilizados. Esto es debido a que el formato de este tipo de grafos se beneficia enormemente de la versión paralela del algoritmo ya que cada vértice –tratado por un sólo hilo de la tarjeta gráfica– satura el arco que lo conecta con el vértice destino en pocas iteraciones del bucle principal. Hay que recordar que, en la inicialización, el vértice fuente envía todo el flujo posible a sus vecinos, que en este caso son todos los vértices del

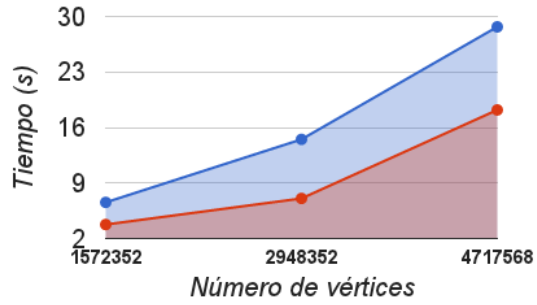


(a) *RMFGEN* Long

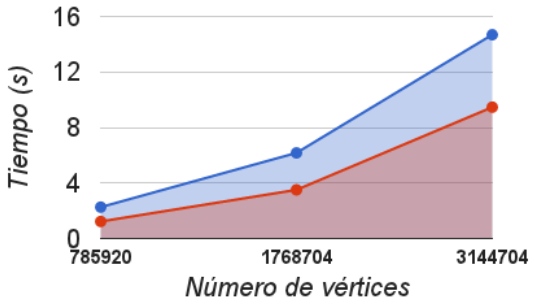


(b) *RMFGEN* Wide

Figura 5.4: Tiempo de ejecución promedio para la implementación paralela del algoritmo sobre *RMFGEN's*



(a) *Washington-RLG* Long



(b) *Washington-RLG* Wide

Figura 5.5: Tiempo de ejecución promedio para la implementación paralela del algoritmo sobre *Washington-RLG's*

grafo. Esto significa que se aprovecha al máximo el nivel de paralelismo puesto que todos los hilos pueden aplicar alguna de las operaciones en su vértice asociado. La alta velocidad a la hora de propagar el exceso de flujo entre los vértices provoca que en pocas iteraciones se saturen todos los arcos que llegan al vértice destino. Por eso, al realizar el renombrado global, todos los vértices son incapaces de llegar al vértice destino mediante arcos de la red residual y se levantan por encima del vértice fuente para devolver su exceso de flujo.

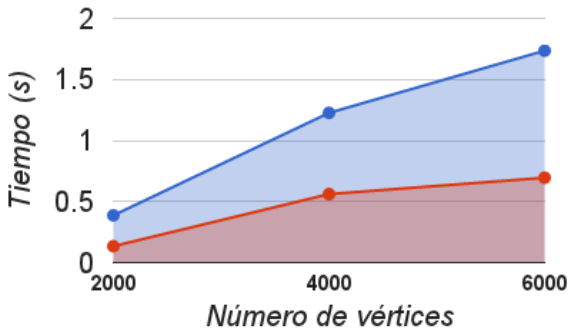


Figura 5.6: Tiempo de ejecución promedio para la implementación paralela del algoritmo sobre *ADG's*

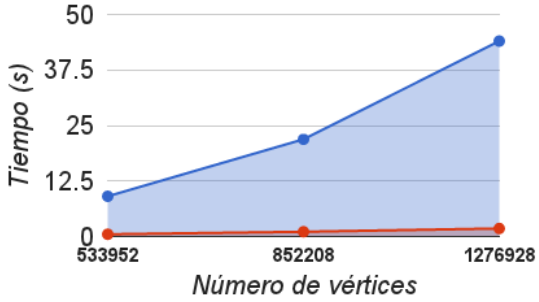
En esta implementación, el porcentaje del tiempo total dedicado al renombrado global es, de media, el 51 %. De todo lo dicho a este respecto se puede concluir que encontrar una forma eficiente de aplicar la optimización es crucial para conseguir un mejor rendimiento global.

5.3. Comparativa de resultados

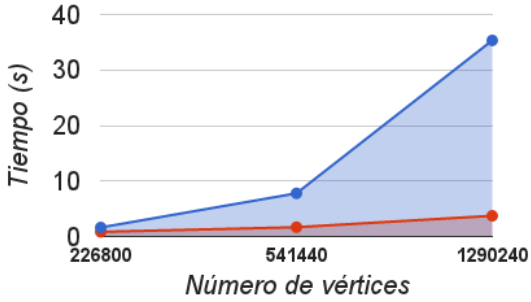
A continuación se analizan los resultados de ambas versiones del algoritmo de *push-relabel* equiparando los tiempos de ejecución total. En las figuras presentadas a continuación se entiende que las gráficas de color azul representan el tiempo de la versión secuencial y las gráficas de color rojo, el tiempo de la versión paralela.

En la figura 5.7 se muestran los tiempos de ejecución de ambas versiones del algoritmo sobre *RMFGEN's*. No sólo se obtienen mejores resultados en la versión paralela –hasta 13 veces más rápida– sino que esta versión escala mejor con el número de vértices.

Los resultados obtenidos utilizando *Washington-RLG's* se pueden observar en la figura 5.8. De nuevo, la versión paralela demuestra ser mucho más rápida, consiguiendo tiempos de ejecución de hasta 24 veces menores que el tiempo de la versión secuencial.

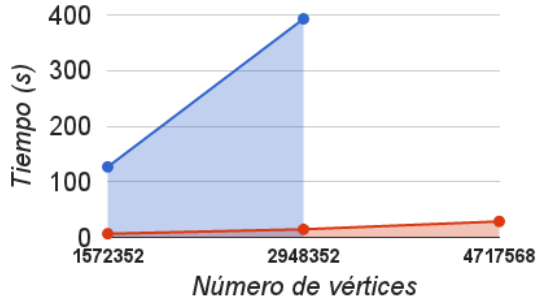


(a) *RMFGEN* Long

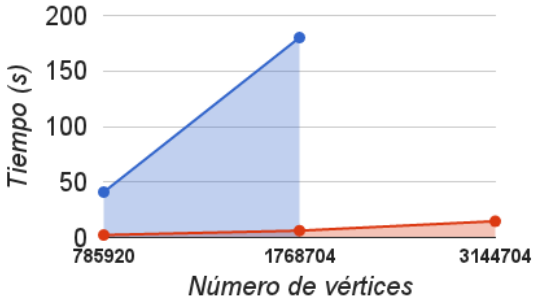


(b) *RMFGEN* Wide

Figura 5.7: Comparación de tiempos de ejecución promedios entre la implementación paralela y secuencial del algoritmo sobre *RMFGEN's*



(a) *Washington-RLG* Long



(b) *Washington-RLG* Wide

Figura 5.8: Comparación de tiempos de ejecución promedios entre la implementación paralela y secuencial del algoritmo sobre *Washington-RLG's*

Por último se pueden observar los resultados del algoritmo sobre los *ADG's* en la figura 5.9. En este tipo de grafos la diferencia de rendimiento no es tan grande si la comparamos con el resto de pruebas. De media, la versión paralela del algoritmo se ejecuta 3 veces más rápida que la versión secuencial.

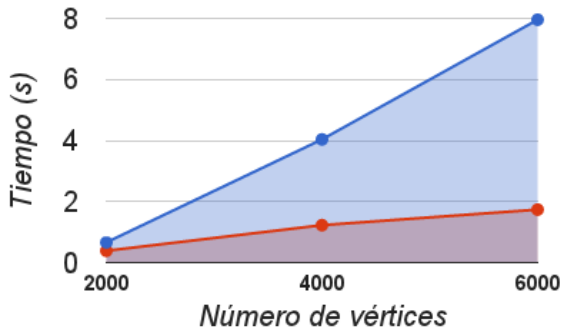


Figura 5.9: Comparación de tiempos de ejecución promedios entre la implementación paralela y secuencial del algoritmo sobre *ADG's*

Apéndice A

Construcción de grafos

A.1. Tipo de datos para la representación de grafos en CUDA

```
typedef struct Graph {
    // Número de vértices.
    int num_vertices;
    /* v[i] = Índice al primer arco que sale del vértice i (se
                añade un vértice extra para homogeneizar el acceso
                a los arcos) */
    int *v;
    // h[i] = Altura del vértice i
    int *h;
    // e[i] = Exceso de flujo del vértice i
    int *e;
    // Número de arcos del grafo residual.
    int num_edges;
    // v_target[i] = Índice al vértice al que apunta el arco i
    int *v_target;
    // f[i] = Flujo que transita el arco i
    //int *f;
    // c[i] = Capacidad del arco i
    int *c;
    // cf[i] = Capacidad residual del arco i
    int *cf;
    // cf_i[i] = Índice del arco inverso de i
    int *cf_i;
} Graph;
```

A.2. Lectura de grafos con formato de matriz de adyacencia

```
// Definición de tipos auxiliares
typedef std::vector<int> VectorInt;

Graph* readFromFile(std::string path) {
    FILE *f = fopen(path.c_str(), "r");

    char c;
    int intValue = 0;
    int numNodes = 0;
    int row = 0, col = 0;
    // Primero se lee el número de vértices del grafo
    do {
        c = fgetc(f);
        if (c != '\n')
            numNodes = numNodes * 10 + (c - '0');
    } while (c != '\n');
    // Se crea la matriz
    int **mat;
    mat = (int**) malloc(numNodes * sizeof(int*));
    for (int i = 0; i < numNodes; i++) {
        mat[i] = (int*) malloc(numNodes * sizeof(int));
    }

    do {
        c = fgetc(f);
        if (c == ' ') {
            mat[row][col] = intValue;
            col++;
            intValue = 0;
        }
        else if (c == '\n') {
            col = 0;
            row++;
            intValue = 0;
        }
        else if (c >= '0' && c <= '9') {
            intValue = intValue * 10 + (c - '0');
        }
    } while (c != EOF);
```

```
    // Se convierte la matriz al formato adecuado para CUDA
    Graph *graph = matToGraph(mat, numNodes);
    // Se libera la memoria de la matriz
    for (int i = 0; i < numNodes; i++)
        free(mat[i]);
    free(mat);
    return graph;
}

Graph* matToGraph(int **rmat, int n) {
    Graph *g = (Graph*) malloc(sizeof(Graph));

    /* Se hace una primera pasada por la matriz contando el número
        de arcos (para después reservar la memoria necesaria) */
    int num_edges = 0;
    int *edgeSizes = (int*) malloc(sizeof(int) * n);
    int *iEdges = (int*) malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        edgeSizes[i] = 0;
        iEdges[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (rmat[i][j] > 0 && (rmat[j][i] == 0 ||
                (rmat[j][i] > 0 && i > j))) {
                edgeSizes[i] += 1;
                edgeSizes[j] += 1;
                num_edges += 2;
            }
        }
    }

    // Se introduce toda la información en el grafo final
    // Información de los vértices
    g->num_vertices = n;
    g->v = (int*) malloc(sizeof(int) * (g->num_vertices + 1));
    g->h = (int*) malloc(sizeof(int) * (g->num_vertices + 1));
    g->e = (int*) malloc(sizeof(int) * (g->num_vertices + 1));
    int nextN = 0;
    for (int i = 0; i < n; i++) {
        g->v[i] = nextN;
        g->h[i] = 0;
```

```
        g->e[i] = 0;
        nextN += edgeSizes[i];
    }
    /* Se añade un último vértice extra para homogeneizar
        el acceso a los arcos */
    g->v[n] = nextN;
    g->h[0] = n;
    // Información de los arcos
    g->num_edges = num_edges;
    g->v_target = (int*) malloc(sizeof(int) * num_edges);
    g->c = (int*) malloc(sizeof(int) * num_edges);
    g->cf = (int*) malloc(sizeof(int) * num_edges);
    g->cf_i = (int*) malloc(sizeof(int) * num_edges);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (rmat[i][j] > 0) {
                // Índice del arco que se añade
                int iEdge = g->v[i] + iEdges[i];
                // Índice del arco inverso al que se añade
                int iEdgeInv = g->v[j] + iEdges[j];
                /* Si el arco inverso no está presente se encarga este
                    arco de rellenar su información. Lo mismo pasa si
                    ambos arcos son no nulos e i > j */
                if (rmat[j][i] == 0 || (rmat[j][i] > 0 && i > j)) {
                    g->v_target[iEdge] = j;
                    g->c[iEdge] = rmat[i][j];
                    g->cf[iEdge] = rmat[i][j];
                    g->cf_i[iEdge] = iEdgeInv;

                    g->v_target[iEdgeInv] = i;
                    g->c[iEdgeInv] = rmat[j][i];
                    g->cf[iEdgeInv] = rmat[j][i];
                    g->cf_i[iEdgeInv] = iEdge;

                    iEdges[i]++;
                    iEdges[j]++;
                }
            }
        }
    }

    return g;
}
```

```
bool contains(VectorInt vector, int element) {
    bool contains = false;

    for (int i = 0; i < vector.size(); i++) {
        if (vector[i] == element) {
            contains = true;
            break;
        }
    }

    return contains;
}

int pos(VectorInt vector, int element) {
    int pos = -1;

    for (int i = 0; i < vector.size(); i++) {
        if (vector[i] == element) {
            pos = i;
            break;
        }
    }

    return pos;
}
```

A.3. Lectura de grafos en formato DIMACS

```
#define START 0
#define STATE_C 'c'
#define STATE_P 'p'
#define STATE_N 'n'
#define STATE_A 'a'

using namespace std;

void initGraph(Graph *g);
void readCLine(FILE *F, char &c, int &a, int &b);
void readPLine(FILE *F, char &c, int &num_vertices, int &num_edges);
void readALine(FILE *f, char &c, int &iVSource, int &iVTarget, int &cap);
void skipToNumber(FILE *f, char &c);
```

```
int readNumber(FILE *f, char &c);

Graph *readRMFGEN(std::string path) {
    FILE *f = fopen(path.c_str(), "r");

    Graph *g = (Graph*) malloc(sizeof(Graph));
    int iEdge = 0;

    char c;
    char state = START;
    int a = 0;
    int b = 0;
    int *v_prev_frame;
    // Último vértice analizado
    int last_vertex = -1;
    do {
        c = fgetc(f);
        switch(state) {
        case START:
            state = c;
            break;
        case STATE_C:
            readCLine(f, c, a, b);
            state = START;
            break;
        case STATE_P:
            readPLine(f, c, g->num_vertices, g->num_edges);

            /* Se suman los arcos del grafo residual que suben (en lugar
                de bajar) entre frames */
            g->num_edges += (a * a) * (b - 1);

            initGraph(g);

            v_prev_frame = (int*) malloc(sizeof(int) * g->num_vertices);
            for (int i = 0; i < g->num_vertices; i++) {
                v_prev_frame[i] = -1;
            }

            state = START;
            break;
        case STATE_N:
            // No se analiza
```

```
    do {c = fgetc(f);} while (c != '\n');
    state = START;
    break;
case STATE_A:
    int iVSource, iVTarget, cap;
    readALine(f, c, iVSource, iVTarget, cap);

    /* Si es el primer arco del vértice, se sitúa el puntero
        de arcos en la posición correcta */
    bool firstEdge = false;
    if (g->v[iVSource + 1] == 0) {
        g->v[iVSource + 1] = g->v[iVSource];
        firstEdge = true;
    }
    g->v[iVSource + 1] += 1;
    g->v_target[iEdge] = iVTarget;
    g->c[iEdge] = cap;
    g->cf[iEdge] = cap;

    // Se comprueba si es un arco al siguiente frame
    int iFrameSource = iVSource / (a * a);
    int iFrameTarget = iVTarget / (a * a);
    if (iFrameSource != iFrameTarget) {
        v_prev_frame[iVTarget] = iVSource;
    }

    // Se actualiza el cf_i
    if (iVSource > iVTarget) {
        int iEdgeInv;
        for (iEdgeInv = g->v[iVTarget];
             iEdgeInv < g->v[iVTarget + 1];
             iEdgeInv++) {
            if (g->v_target[iEdgeInv] == iVSource) {
                break;
            }
        }
        g->cf_i[iEdge] = iEdgeInv;
        g->cf_i[iEdgeInv] = iEdge;
    }

    /* Si es un nuevo vértice, se comprueba si alguno del frame
        anterior le apunta (y crea el arco de éste al del
        frame anterior, si es necesario) */
```

```
            if (last_vertex != -1 &&
                last_vertex != iVSource &&
                v_prev_frame[iVSource] != -1) {
                iEdge++;
                int iVInverse = v_prev_frame[iVSource];

                g->v[iVSource + 1] += 1;
                g->v_target[iEdge] = iVInverse;
                g->c[iEdge] = 0;
                g->cf[iEdge] = 0;

                int iEdgeInv;
                for (iEdgeInv = g->v[iVInverse];
                     iEdgeInv < g->v[iVInverse + 1];
                     iEdgeInv++) {
                    if (g->v_target[iEdgeInv] == iVSource) {
                        break;
                    }
                }
                g->cf_i[iEdge] = iEdgeInv;
                g->cf_i[iEdgeInv] = iEdge;
            }


            last_vertex = iVSource;
            iEdge++;

            state = START;
            break;
        }
    } while (c != EOF);

    return g;
}

Graph *readWashingtonRLG(std::string path) {
    FILE *f = fopen(path.c_str(), "r");

    Graph *g = (Graph*) malloc(sizeof(Graph));
    int iEdge = 0;

    char c;
    char state = START;
```

```
int a = 0;
int b = 0;
// Último vértice analizado
int last_vertex = -1;

/* prev_col_refs[i] = Lista con los índices de los vértices de la
    columna anterior que apuntan al vértice i */
vector<int> **prev_col_refs;
/* cur_col_refs[i] = Lista con los índices de los vértices de la
    columna actual que apuntan al vértice i */
vector<int> **cur_col_refs;

do {
    c = fgetc(f);
    switch(state) {
    case START:
        state = c;
        break;
    case STATE_C:
        readCLine(f, c, a, b);
        state = START;
        break;
    case STATE_P:
        readPLine(f, c, g->num_vertices, g->num_edges);
        // Un arco inverso en la red residual por cada arco del grafo
        g->num_edges *= 2;

        initGraph(g);

        prev_col_refs = (vector<int>**) malloc(sizeof(vector<int>*) * a);
        cur_col_refs = (vector<int>**) malloc(sizeof(vector<int>*) * a);
        for (int iCol = 0; iCol < a; iCol++) {
            prev_col_refs[iCol] = new vector<int>();
            cur_col_refs[iCol] = new vector<int>();
        }

        state = START;
        break;
    case STATE_N:
        // No se analiza
        do {c = fgetc(f);} while (c != '\n');
        state = START;
        break;
```

```
case STATE_A:
    int iVSource, iVTarget, cap;
    readALine(f, c, iVSource, iVTarget, cap);

    /* Si es el primer arco del vértice, se sitúa el puntero
        de arcos en la posición correcta */
    bool firstEdge = false;
    if (g->v[iVSource + 1] == 0) {
        g->v[iVSource + 1] = g->v[iVSource];
        firstEdge = true;
    }
    g->v[iVSource + 1] += 1;
    g->v_target[iEdge] = iVTarget;
    g->c[iEdge] = cap;
    g->cf[iEdge] = cap;

    // Columna del vértice
    int currentCol = (iVSource + a - 1) / a;
    // Índice del vértice fuente en su columna
    int iVSourceMod = (currentCol == 0) ? 0 : (iVSource - 1) % a;
    // Índice del vértice destino en su columna
    int iVTargetMod = (currentCol == 0) ? 0 : (iVTarget - 1) % a;

    /* Si es un vértice nuevo, se recorre la columna anterior en
        busca de referencias al vértice actual para crear los
        arcos inversos */
    if (firstEdge && currentCol != 0) {
        /* Primer arco del primer vértice de la columna: limpiamos
            las prev_col_refs y pasamos cur_col_refs a
            prev_col_refs */
        if (iVSourceMod == 0) {
            for (int iV = 0; iV < a; iV++) {
                prev_col_refs[iV]->clear();
            }
            vector<int> **tmp_col_refs = prev_col_refs;
            prev_col_refs = cur_col_refs;
            cur_col_refs = tmp_col_refs;
        }

        /* Por cada vértice de prev_col_refs[iVSourceMod]
            se actualizan los arcos inversos */
        vector<int> refs = *prev_col_refs[iVSourceMod];
        for (vector<int>::iterator it = refs.begin();
```

```
                 it != refs.end();
                 it++) {
                int iV = *it;
                for (int iE = g->v[iV]; iE < g->v[iV + 1]; iE++) {
                    if (g->v_target[iE] == iVSource) {
                        // Se añade el arco inverso en el vértice actual
                        g->v[iVSource + 1] += 1;
                        iEdge++;
                        g->v_target[iEdge] = iV;
                        g->c[iEdge] = 0;
                        g->cf[iEdge] = 0;
                        // Se actualizan los índices a arcos inversos
                        g->cf_i[iEdge] = iE;
                        g->cf_i[iE] = iEdge;
                    }
                }
            }
        }

        cur_col_refs[iVTargetMod]->push_back(iVSource);

        last_vertex = iVSource;
        iEdge++;

        state = START;
        break;
    }
} while (c != EOF);

/* Se añaden los últimos arcos inversos (del destino a los
    de la última columna) */
g->v[g->num_vertices] = g->v[g->num_vertices - 1];
for (int iV = a * b - a + 1; iV < g->num_vertices - 1; iV++) {
    for (int iE = g->v[iV]; iE < g->v[iV + 1]; iE++) {
        if (g->v_target[iE] == g->num_vertices - 1) {
            // Se añade el arco inverso en el vértice actual
            g->v[g->num_vertices] += 1;
            g->v_target[iEdge] = iV;
            g->c[iEdge] = 0;
            g->cf[iEdge] = 0;
            // Se actualizan los índices a arcos inversos
            g->cf_i[iEdge] = iE;
            g->cf_i[iE] = iEdge;
```

```
                iEdge++;
            }
        }
    }

    return g;
}

/* Inicializa los arrays del grafo utilizando las variables de
    número de vértices y número de arcos */
void initGraph(Graph *g) {
    g->v = (int*) calloc((g->num_vertices + 1), sizeof(int));
    g->h = (int*) calloc((g->num_vertices + 1), sizeof(int));
    g->e = (int*) calloc((g->num_vertices + 1), sizeof(int));

    g->v_target = (int*) malloc(sizeof(int) * g->num_edges);
    g->c = (int*) malloc(sizeof(int) * g->num_edges);
    g->cf = (int*) malloc(sizeof(int) * g->num_edges);
    g->cf_i = (int*) malloc(sizeof(int) * g->num_edges);
}

/* Lee los parámetros de una linea del tipo C y los guarda en
    las variables a y b */
void readCLine(FILE *f, char &c, int &a, int &b) {
    skipToNumber(f, c);
    if (c >= '0' && c <= '9') {a = readNumber(f, c);}
    skipToNumber(f, c);
    if (c >= '0' && c <= '9') {b = readNumber(f, c);}
    do {c = fgetc(f);} while (c != '\n');
}

/* Lee los parámetros de una linea del tipo P y los guarda en
    las variables num_vertices y num_edges */
void readPLine(FILE *f, char &c, int &num_vertices, int &num_edges) {
    // Primer número = número de vértices
    skipToNumber(f, c);
    num_vertices = readNumber(f, c);
    // Segundo número = número de arcos
    skipToNumber(f, c);
    num_edges = readNumber(f, c);
}

void readALine(FILE *f, char &c, int &iVSource, int &iVTarget, int &cap) {
```

```
    skipToNumber(f, c);
    iVSource = readNumber(f, c) - 1;
    skipToNumber(f, c);
    iVTarget = readNumber(f, c) - 1;
    skipToNumber(f, c);
    cap = readNumber(f, c);
}

void skipToNumber(FILE *f, char &c) {
    do {
        c = fgetc(f);
    } while (c < '0' || c > '9' || c == '\n');
}

int readNumber(FILE *f, char &c) {
    int number = 0;
    do {
        number = (number * 10) + (c - '0');
        c = fgetc(f);
    } while (c != '\n' && c != ' ');
    return number;
}
```

Apéndice B

Algoritmo de *push-relabel* paralelo

```
// Número de ciclos en el kernel de push relabel
#define KERNEL_INNER_CYCLES 5
/* Número de veces que se llama al kernel de push relabel
    antes de aplicar el renombrado global */
#define KERNEL_OUTER_CYCLES 220

#define CUDA_SAFE_CALL( call) do { \
    cudaError err = call; \
    if( cudaSuccess != err) { \
        fprintf(stderr, "Cuda error in file '%s' in line %i : %s.\n", \
                __FILE__, __LINE__, cudaGetErrorString( err) ); \
                getchar(); \
    exit(EXIT_FAILURE); \
    } } while (0)

typedef struct {
    int vertices[CHUNK_SZ + 1];
    int levels[CHUNK_SZ];
} warp_mem_t;

using namespace std;

int main(int argc, const char* argv[]) {
    Graph *g = readWashingtonRLG("washington-rlg.txt");
    initialize_flow(g);

    // Variables del device
    int *dev_v, *dev_h, *dev_e, *dev_v_target, *dev_cf, *dev_cf_i;

    bool finished;
```

```
bool *dev_finished;
cudaMalloc((void **) &dev_finished, sizeof(bool));
int cycle = 0;

// Se copia el grafo del host al device
memcpy_host_to_dev(g, &dev_v, &dev_h, &dev_e,
                   &dev_v_target, &dev_cf, &dev_cf_i);

// Bucle principal del algoritmo de push-relabel
do {
    // Se inicializa finished (del device) a true
    finished = true;
    cudaMemcpy(dev_finished, &finished, sizeof(bool),
               cudaMemcpyHostToDevice);
    // Se ejecuta el kernel de push-relabel paralelo
    maxflow_kernel <<< (g->num_vertices / 1024) + 1, 1024 >>>
        (dev_finished, g->num_vertices, dev_v, dev_h,
         dev_e, dev_v_target, dev_cf, dev_cf_i);
    // Se comprueba si el algoritmo ha finalizado
    cudaMemcpy(&finished, dev_finished, sizeof(bool),
               cudaMemcpyDeviceToHost);
    /* Renombrado global si no se ha terminado y ha pasado
        un número determinado de ciclos */
    if (!finished && cycle == KERNEL_OUTER_CYCLES) {
        global_relabeling(g, dev_v, dev_h, dev_e,
                          dev_v_target, dev_cf, dev_cf_i);

        cycle = 0;
    }
    cycle++;
} while (!finished);

int max_flow;
CUDA_SAFE_CALL(cudaMemcpy(&max_flow, dev_e + g->num_vertices - 1,
                          sizeof(int), cudaMemcpyDeviceToHost));

// Se libera la memoria del device
cudaFree(dev_v);
cudaFree(dev_h);
cudaFree(dev_e);
cudaFree(dev_v_target);
cudaFree(dev_cf);
cudaFree(dev_cf_i);
```

```
    cudaEventDestroy(cuda_start);
    cudaEventDestroy(cuda_stop);
    cudaEventDestroy(cuda_bfs_start);
    cudaEventDestroy(cuda_bfs_stop);
    // Se libera la memoria del host
    free(g->v);
    free(g->e);
    free(g->h);
    free(g->v_target);
    free(g->c);
    free(g->cf);
    free(g->cf_i);
    free(g);

    // Evita que el programa termine sin mostrar la salida
    getchar();
}

void initialize_flow(Graph *g){
    // Se pone el vértice fuente a altura num_vertices
    g->h[0] = g->num_vertices;

    // Se envía todo el flujo posible desde el vértice fuente
    for (int i = g->v[0]; i < g->v[1]; i++) {
        // cf[i] = cf[i] - c[i] (= 0)
        g->cf[i] = 0;
        // cf[i'] = cf[i'] + c[i]
        g->cf[g->cf_i[i]] = g->cf[g->cf_i[i]] + g->c[i];
        g->e[g->v_target[i]] = g->c[i];
        g->e[0] -= g->c[i];
    }
}

__global__ void maxflow_kernel(
        bool *finished,
        int num_vertices,
        int *v,
        int *h,
        int *e,
        int *v_target,
        int *cf,
        int *cf_i) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
/* Para todos los vértices excepto el fuente(0) y
    el destino(num_nodes - 1) */
if (idx != 0 && idx < num_vertices - 1) {
    int cycle = KERNEL_INNER_CYCLES;

    while (cycle > 0) {
        if (e[idx] > 0 && h[idx] < num_vertices) {
            *finished = false;
            int ep = e[idx];
            // Se calcula el vecino que está situado a menos altura
            int min_v = INT_MIN; // Índice al vértice vecino más bajo
            int min_h = INT_MAX; // Altura de dicho vértice
            int min_e = INT_MIN; /* Índice del arco del vértice
                                    actual a dicho vértice */
            int firstEdge = v[idx];
            int lastEdge = v[idx + 1];
            for (int i = firstEdge; i < lastEdge; i++) {
                int vp = v_target[i];
                int hp = h[vp];
                if (hp < min_h && cf[i] > 0) {
                    min_v = vp;
                    min_h = hp;
                    min_e = i;
                }
            }

            // Push
            if (h[idx] > min_h) {
                int d = min(ep, cf[min_e]);
                atomicSub(&cf[min_e], d);
                atomicAdd(&cf[cf_i[min_e]], d);
                atomicAdd(&e[min_v], d);
                atomicSub(&e[idx], d);
            }
            // Relabel
            else {
                h[idx] = min_h + 1;
            }
        }
        cycle--;
    }
}
```

```
}

void global_relabeling(Graph *g,
        int *dev_v, int *dev_h, int *dev_e,
        int *dev_v_target, int *dev_cf, int *dev_cf_i,
        int *dev_mask, int *dev_active_mask) {

    /* Se cancelan las violaciones de flujo provocadas por el renombrado global de la
        versión paralela del algoritmo */
    cancel_violation_kernel <<< (g->num_vertices / 1024) + 1, 1024 >>>
        (g->num_vertices, dev_v, dev_h, dev_e, dev_v_target,
         dev_cf, dev_cf_i);
    // Se reinicializan las alturas
    int current = 0;
    init_heights_kernel <<< (g->num_vertices / 1024) + 1, 1024 >>>
        (g->num_vertices, dev_h);
    // Backward BFS desde el vértice destino
    bool finished_bfs = false;
    bool *dev_finished_bfs;
    cudaMalloc((void **) &dev_finished_bfs, sizeof(bool));
    do {
        finished_bfs = true;
        cudaMemcpy(dev_finished_bfs, &finished_bfs, sizeof(bool),
                   cudaMemcpyHostToDevice);
        bfs_kernel <<< (g->num_vertices / 1024) + 1, 1024 >>>
            (g->num_vertices, current, dev_h, dev_v, dev_v_target,
             dev_cf, dev_cf_i, dev_finished_bfs);
        cudaMemcpy(&finished_bfs, dev_finished_bfs, sizeof(bool),
                   cudaMemcpyDeviceToHost);
        current++;
    } while (!finished_bfs);
    // Backward BFS desde el vértice fuente
    current = g->num_vertices;
    do {
        finished_bfs = true;
        cudaMemcpy(dev_finished_bfs, &finished_bfs, sizeof(bool),
                   cudaMemcpyHostToDevice);
        bfs_kernel <<< (g->num_vertices / 1024) + 1, 1024 >>>
            (g->num_vertices, current, dev_h, dev_v, dev_v_target,
             dev_cf, dev_cf_i, dev_finished_bfs);
        cudaMemcpy(&finished_bfs, dev_finished_bfs, sizeof(bool),
                   cudaMemcpyDeviceToHost);
        current++;
```

```
    } while (!finished_bfs);
}

__global__ void cancel_violation_kernel(int num_vertices, int *v,
        int *h, int *e, int *v_target, int *cf, int *cf_i) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_vertices) {
        int firstEdge = v[idx];
        int lastEdge = v[idx + 1];
        for (int i = firstEdge; i < lastEdge; i++) {
            int v = v_target[i];
            if (h[idx] > h[v] + 1) {
                e[idx] = e[idx] - cf[i];
                e[v] = e[v] + cf[i];
                cf[cf_i[i]] = cf[cf_i[i]] + cf[i];
                cf[i] = 0;
            }
        }
    }
}

__global__ void init_heights_kernel(int num_vertices, int *height) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    /* Se ponen todas las alturas a lo máximo posible y sólo la
        del vértice fuente a 0 */
    if (idx > 0 && idx < num_vertices) {
        height[idx] = INT_MAX;

        /* Para evitar divergencias dentro de un warp, cada hilo
           realiza estas asignaciones */
        height[0] = num_vertices;
        height[num_vertices - 1] = 0;
    }
}

__global__ void bfs_kernel(int num_vertices, int current,
        int *levels, int *vertices, int *edges, int *cf, int *cf_i,
        bool *finished) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_vertices && levels[idx] == current) {
```

```
        for (int i = vertices[idx]; i < vertices[idx + 1]; i++) {
            int j = edges[i];
            if (levels[j] == INT_MAX && cf[cf_i[i]] > 0) {
                *finished = false;
                levels[j] = current + 1;
            }
        }
    }
}

void memcpy_host_to_dev(Graph *g, int **dev_v, int **dev_h,
        int **dev_e, int **dev_v_target, int **dev_cf, int **dev_cf_i) {
    CUDA_SAFE_CALL(cudaSetDevice(0));

    // Asignación de memoria en el device
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_v,
        (g->num_vertices + 1) * sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_h,
        (g->num_vertices + 1) * sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_e,
        (g->num_vertices + 1) * sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_v_target,
        g->num_edges * sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_cf,
        g->num_edges * sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**) dev_cf_i,
        g->num_edges * sizeof(int)));
    // Copia de memoria del host al device
    CUDA_SAFE_CALL(cudaMemcpy(*dev_v, g->v,
        (g->num_vertices + 1) * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(*dev_h, g->h,
        (g->num_vertices + 1) * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(*dev_e, g->e,
        (g->num_vertices + 1) * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(*dev_v_target, g->v_target,
        g->num_edges * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(*dev_cf, g->cf,
        g->num_edges * sizeof(int), cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(*dev_cf_i, g->cf_i,
        g->num_edges * sizeof(int), cudaMemcpyHostToDevice));
}

void memcpy_dev_to_host(Graph *g, int *dev_h, int *dev_e, int *dev_cf) {
```

```
    CUDA_SAFE_CALL(cudaMemcpy(g->h, dev_h,
        (g->num_vertices + 1) * sizeof(int), cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaMemcpy(g->e, dev_e,
        (g->num_vertices + 1) * sizeof(int), cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaMemcpy(g->cf, dev_cf,
        g->num_edges * sizeof(int), cudaMemcpyDeviceToHost));
}
```

Bibliografía

[1] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1994.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[3] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[4] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404.

[5] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[6] Donald Goldfarb and Michael Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13:81–123, 1988.

[7] Pawan Harish, Vibhav Vineet, and P J Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, 2009.

[8] Zhengyu He and Bo Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-hybrid platforms. *The 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, April 2010.

[9] Bo Hong. A lock-free multi-threaded algorithm for the maximum flow problem. In *IPDPS'08*, pages 1–8, 2008.

[10] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[11] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on CUDA.

[12] Jorkki Hyvonen, Jari Saramaki, and Kimmo Kaski. Efficient data structures for sparse network representation. *Int. J. Comput. Math.*, 85(8):1219–1233, August 2008.

[13] David S. Johnson, Catherine C. McGeoch, and DIMACS (Group). *Network Flows and Matching: First Dimacs Implementation Challenge*. Dimacs Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1993.

[14] Agnieszka Lupinska. Parallel implementation of flow and matching algorithms. *CoRR*, abs/1110.6231, 2011.

[15] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[16] Victoria Popic and Javier Velez. A parallel implementation of the push-relabel max-flow algorithm with heuristics. 2010.