UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA EN COMPUTACIÓN

TÉCNICAS DE DISEÑO DE ALGORITMOS DE FLUJOS

T E S I S

QUE PARA OPTAR POR EL GRADO DE
MAESTRO EN **CIENCIAS DE LA COMPUTACIÓN**

PRESENTA:
ARMANDO BALLINAS NANGUELU

DIRECTOR DE TESIS:
DR. JOSÉ DAVID FLORES PEÑALOZA
FACULTAD DE CIENCIAS, UNAM
COTUTOR:
DR. ARMANDO CASTAÑEDA ROJANO
INSTITUTO DE MATEMÁTICAS, UNAM

Ciudad Universitaria, Cd. Mx. agosto 2017

1. Datos del alumno
    - Armando Ballinas Nangüelú.
    - 57721086
    - Universidad Nacional Autónoma de México.
    - Posgrado en Ciencia e Ingeniería de la Computación.
    - Maestría en Ciencia e Ingeniería de la Computación.
    - 306001073
2. Datos del tutor
    - Dr. José David Flores Peñaloza.
3. Datos del cotutor
    - Dr. Armando Castañeda Rojano.
4. Datos del sinodal 1
    - Dra. Adriana Ramírez Vigueras.
5. Datos del sinodal 2
    - Dr. José de Jesús Galaviz Casas.
6. Datos del sinodal 3
    - Dr. Sergio Rajsbaum Gorodezky.
7. Datos del trabajo escrito
    - Técnicas de Diseño de Algoritmos de Flujos.
    - pp 84.
    - 2017

Dedicado a la Familia Romero Rodríguez, en especial a Maru q.e.p.d. por ser una segunda familia para mí.

Agradecimientos

A mi madre Marlene y a mi abuelita Celia por ser la familia que siempre ha estado conmigo y siempre me ha apoyado.

A mis compañeros y amigos Leandro Casuso, Nestaly Marín, Gerardo Zagasta, Carmen Cedillo, Erick Solis por compartir conmigo divertidos momentos durante los congresos asistidos durante el posgrado.

A mis amigos Karla Vargas, Diana Montes, Manuel Alcántara y Miguel Ángel Rodríguez por estar siempre conmigo y por sus consejos de vida.

A mis tutores David Flores y Armando Castañeda por sus consejos y tutorías para la realización de este trabajo y la culminación satisfactoria de mis estudios.

A mis sinodales Adriana Ramírez, Sergio Rajsbaum y José Galaviz por sus comentarios acerca de este trabajo.

A "Lulú", Amalia y Cecilia por su dedicación, paciencia y orientación para llevar a cabo los trámites académicos y administrativos durante el posgrado.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por haberme otorgado una beca para la realización de mis estudios de maestría y la conclusión de este trabajo.

Tabla de contenidos.

Lista de Algoritmos.

Introducción

En este trabajo se presenta un conjunto de técnicas de diseño de algoritmos cuya finalidad es servir como base para un curso a nivel maestría o licenciatura acerca del tema de algoritmos de flujos.

La motivación de este trabajo surge tras unas charlas que dio el Dr. John Hopcroft en la Facultad de Ciencias y el Instituto de Matemáticas de la UNAM durante 2015 en donde habló acerca de lo que él cree serán los fundamentos de las ciencias de la computación y de las ciencias de datos en el futuro. Como parte de estos fundamentos se encuentran los algoritmos que procesen cantidades masivas de datos.

Este tipo de algoritmos se ha hecho muy popular en los años recientes y en él se encuentran los algoritmos que, de acuerdo a su concepción, son capaces de procesar cantidades arbitrarias de datos.

Los *algoritmos de flujos* son un modelo que si bien se vislumbra desde los años 80 [12] no es hasta después del cambio de milenio cuando se acuña el término *streaming* y se da un auge de estos. En los recientes años el interés de estos algoritmos ha crecido mucho y se debe a su gran versatilidad para atacar problemas con cantidades masivas de datos que surgen hoy en día naturalmente por la manera en la que se han desarrollado ramas como la inteligencia artificial o la estadística computacional.

Varios científicos creemos que las ciencias de datos serán parte fundamental de la enseñanza del mañana, estas ciencias tienen como objetivo procesar, analizar, comparar e incluso generar cantidades arbitrarias de datos para poder predecir o analizar eventos y comportamientos sociales, económicos, políticos, biológicos, psicológicos, etc. Estos científicos de datos deben ser capaces de lidiar con cantidades masivas de información y tener a la mano herramientas (no solo computacionales) que les permitan llevar a cabo su objetivo.

En el trabajo se plantea el modelo de algoritmo de flujo y se describen y detallan diversas técnicas de diseño o lineamientos que permiten al lector comprender el modelo, sus limitaciones y virtudes. Estas técnicas también permiten ejemplificar el uso del modelo de flujos en distintos tipos de problemas, donde en todos se debe procesar una cantidad gigante de información.

Las técnicas de diseño aquí descritas han sido tomadas de diversos recursos disponibles en la bibliografía, en particular se puede empezar por consultar [20] en donde se exhibe una buena cantidad de ejemplos y técnicas para este tipo de algoritmos. Al igual que McGregor, en este trabajo los algoritmos trabajan sobre gráficas (o grafos, en algunos países). Es decir, para ejemplificar las técnicas descritas se trata de resolver un problema de teoría de gráficas; el resultado es un algoritmo de flujos que resuelva o aproxime una solución a un problema de teoría de gráficas dado. Por supuesto, para cada técnica se presenta un problema distinto, aunque algunos problemas se pudieran resolver mediante varias técnicas.

A lo largo del trabajo se supone que el lector ha cursado una asignatura de estructuras de datos y algoritmos, por lo que está familiarizado con estructuras de datos como: arreglos, listas, árboles binarios y con conceptos teóricos de algoritmos como: complejidad temporal,

espacial y el cálculo (informal) de estas. Así mismo, se supone que el lector conoce varios algoritmos básicos que actúan sobre diversas estructuras de datos y presentas distintas técnicas de implementación. Por la naturaleza de los ejemplos, se supone también que el lector ha tomado un curso de teoría de gráficas o por lo menos está familiarizado con el concepto de *gráfica* y algunos conceptos y algunas propiedades básicas de estas, por ejemplo qué es un *ciclo*, qué es una *coloración*, qué es un *árbol* o qué significa *conexo*.

A pesar de las suposiciones anteriores, la mayoría de los capítulos son autocontenidos o tienen referencias precisas de dónde encontrar más información. Así mismo se crea un apéndice donde se presenta una estructura de datos que, aunque es conocida, no es común. De hecho en el apéndice se presenta una variante ingeniosa y tal vez, novedosa para su uso en el modelo de flujos. En la bibliografía se encuentran libros básicos de algoritmos y estructuras de datos como [8] o [17] y de teoría de gráficas como [23] que el lector puede consultar para obtener más información acerca de conceptos empleados en este trabajo. Algunos de estos conceptos están citados para ayudar al lector con las referencias.

Las técnicas se presentan en capítulos separados. El primero presenta el modelo de algoritmos de flujos así como conceptos que son necesarios para varias técnicas como algoritmos de aproximación y esquemas de aproximación, también se presenta el concepto de algoritmos probabilistas. Estos conceptos no son novedosos, sin embargo no forman parte de un curso básico de algoritmos y estructuras de datos por lo que se decide presentarlos desde el primer capítulo. Dentro de la presentación del modelo se introduce un primer algoritmo de flujos para familiarizar al lector con la estructura básica de estos.

En el segundo capítulo se presenta una técnica que adapta un algoritmo existente en el modelo DRAM a uno en el modelo de flujos. Esta técnica permite construir algoritmos de flujos a partir de los existentes. Para ejemplificar la técnica se presenta el problema de determinar si una gráfica es *bipartita* o *2-coloreable*.

La técnica de diseño *greedy* es muy utilizada en los algoritmos convencionales y en el tercer capítulo se presenta esta técnica para los algoritmos de flujos. Así mismo se generaliza la técnica para que funcione en todo objeto matemático llamado *matroide* siendo esta una aportación al trabajo realizado previamente en flujos que reciben un matroide como entrada. En el capítulo se presentan todas las definiciones y conceptos relacionados a los matroides para ayudar al lector a comprender mejor la generalización de la técnica *greedy* realizada. El problema de ejemplo es el de encontrar un árbol generador de peso mínimo a partir de una gráfica arbitraria.

Debido a las limitaciones del modelo de flujos, no siempre es posible hallar soluciones óptimas (mejores posibles) a los problemas dados, por eso varias técnicas tienen como fundamento la aproximación de soluciones óptimas. En el capítulo 4 se presenta una técnica que se basa en encontrar una solución parcial y después mejorarla para hallar una mejor aproximación al problema dado.

En el capítulo 5 se presenta la técnica de *muestreo* que es otra forma de encontrar soluciones parciales o aproximaciones a los problemas dados. Esta técnica tiene su relevancia al considerar una muestra representativa de un conjunto grande de individuos (o datos) y permite calcular de manera exacta o simple soluciones sobre la muestra para después interpretar estas soluciones y conjeturar propiedades o soluciones del conjunto gigante de datos.

El último capítulo presenta una técnica de reducción de algoritmos de flujos, es decir, presenta una manera de utilizar algoritmos de flujos ya conocidos y estudiados para ser utilizados como subrutinas en nuevos algoritmos que resuelvan otros problemas. En este caso, los algoritmos básicos utilizados calculan algo llamado *momentos de frecuencia de una secuencia*. Este concepto de momentos de frecuencia ha sido estudiado y utilizado bastante en la Estadística, debido a su utilidad para estimar información acerca de un conjunto de datos. Esta técnica es la más complicada debido a los conceptos que requiere, pues no solo se necesita saber qué son estos momentos, cosa que se define en el capítulo, sino también requieren que el lector conozca de reducciones en algoritmos, algoritmos probabilistas y esquemas de aproximación. Todos los conceptos necesarios y relevantes se definen o bien dentro del capítulo o bien en el primer capítulo ya que son utilizados en otros puntos del trabajo. En este capítulo el problema a resolver es el de contar el número de triángulos (K_3) que forman las aristas de una gráfica. Para realizar este conteo se establece una relación entre el número de triángulos de una gráfica con el cálculo de los momentos de frecuencia de una secuencia relacionada con las aristas de la gráfica.

Finalmente se presentan las conclusiones del trabajo así como el posible trabajo futuro, tanto en el área como para extender este trabajo.

Como apéndice a este trabajo se presenta una estructura de datos con una pequeña mejora o generalización que es utilizada en el capítulo 2 y que puede ser utilizada en otros contextos y algoritmos, incluso en el modelo convencional o en otros modelos de algoritmos. Cabe mencionar que esta estructura es una aportación propia.

1. Marco de Trabajo

En este primer capítulo se introduce el modelo de cómputo que se presentará en la tesis, así como algunos conceptos necesarios para su entendimiento.

En el trabajo se presentan diversos algoritmos que reciben como entrada una gráfica y que actúan sobre ella, sin embargo el modelo sobre el que trabajan es distinto al convencional, por lo que se presenta en detalle en este capítulo. También se describe y analiza un algoritmo sencillo para identificar las sutilezas del modelo y familiarizarse con el mismo. En la última sección se establece el concepto de *esquema de aproximación*; el cual es necesario conocer para lidiar y razonar con varios de los algoritmos presentados más adelante.

1.1. Algoritmos de Flujos

De manera informal, un algoritmo se entiende como una serie de instrucciones claramente definidas que al recibir una entrada producen una salida que es la solución a un problema dado. En estos algoritmos convencionales la entrada se considera accesible y guardada en una memoria (usualmente DRAM) y en muchos algoritmos se utiliza esta idea para acceder arbitrariamente a ella.

Una de las ventajas de pensar en este modelo de entrada es que los datos siempre están accesibles y más aún, cada uno de ellos se puede acceder con el mismo costo, de manera aleatoria y arbitraria. Esto implica que si se usa un arreglo, por ejemplo, se pueden acceder a todas y cada una de las posiciones del mismo con el mismo costo, de manera aleatoria (sin un orden específico) y las veces que se requiera.

Este modelo es muy conveniente y estándar y muchas técnicas de diseño de algoritmos lo han utilizado para modelar la entrada durante décadas. Sin embargo, hoy en día hay circunstancias donde resulta inviable o hasta imposible pensar que la entrada de los algoritmos se presente de esta manera.

Debido a que actualmente hay escenarios donde se genera demasiada información para que pueda ser almacenada de manera convencional, se debe considerar otra forma de presentar la entrada. Esta debe tratar esta inmensa cantidad de información de una forma práctica y sencilla.

Ejemplos de estos escenarios son: la cantidad de tuits emitidos durante eventos de gran impacto como el Súper Tazón o durante una elección, el número de peticiones realizadas a un servidor de Google o la información arrojada por el Gran Colisionador de Hadrones del CERN durante un experimento. En los ejemplos anteriores si la información recabada se almacenará, entonces se necesitaría una gran infraestructura para ello y esta podría no estar al alcance de todos, lo cual se volvería impráctico y costoso.

El *modelo de flujo de datos* tiene como objetivo capturar este fenómeno y está formulado para poder crear algoritmos que requieran procesar una cantidad inmensa de datos y que estos no puedan ser almacenados en una memoria de la manera convencional. El problema es

abordado de modo que se cambie la manera de introducir la entrada y lo hacen al presentarla como un flujo.

Un *flujo de datos*, o simplemente *flujo*, es una secuencia arbitrariamente larga de elementos, en donde se van procesando uno a uno en un orden desconocido. Así mismo, una vez que se procesa un elemento del flujo, este se descarta y "se olvida". Los flujos, a pesar de ser inmensos, se consideran finitos, por lo que a la acción de procesar todos los elementos de él se le conoce como una *pasada* al flujo.

Observe que en este modelo no se permite ni alterar los elementos del flujo ni tampoco reordenarlos o acceder a ellos de manera aleatoria. De hecho, conviene pensar que estos elementos se van generando mientras el algoritmo se ejecuta, de modo que podría incluso, desconocerse el tamaño final del flujo.

Un algoritmo creado utilizando el modelo de flujo de datos se conoce como *algoritmo de flujo* y este no solo cambia la manera de percibir y utilizar la entrada, sino que usualmente pone restricciones al tiempo de procesamiento requerido por elemento del flujo y también a la cantidad de espacio adicional que pueda utilizar durante su ejecución.

Las restricciones varían en la literatura y algunas pueden ser bastante estrictas y tienen como objetivo lidiar con la inmensa cantidad de información de manera eficiente, en particular no se permite almacenar todo el contenido del flujo y se espera que cada elemento de él se procese de manera "rápida".

Por ejemplo, en el modelo más aceptado de algoritmos de flujos si el tamaño del flujo es n, entonces el tiempo de procesamiento por elemento debe ser $O(\log(n)^k)$, con k de tamaño constante con respecto a n, lo que significa que tome tiempo proporcional a un polinomio con respecto al logaritmo del tamaño de la entrada, a esto se le conoce como tiempo *polilogarítmico*. También suele restringirse el espacio que se utiliza para almacenar datos en el algoritmo y usualmente se pide que tenga también orden $O(\log(n)^k)$ como se muestra en [3].

Sin embargo, existe otro modelo ampliamente utilizado que se conoce como modelo de *semi-flujo*, en el que se relajan un poco más las restricciones aceptadas por el modelo original. En este modelo se permite almacenar más espacio de trabajo para el algoritmo, estando este en $O(n \log(n))$ para un flujo de tamaño n.

Además de las restricciones anteriores, también suele restringirse el número de veces que se puede recorrer todo el flujo (número de pasadas al flujo). En particular, el mínimo es 1. Esta restricción tiene sentido debido al costo que puede implicar acceder al flujo o también puede deberse a la pérdida de información inherente. Piense por ejemplo, en las peticiones hechas a un servidor web; en este caso, a lo mejor, no se pueden almacenar de ninguna manera todas las peticiones, por lo que un elemento del flujo es "destruido" una vez que es procesado. En este escenario, el número de pasadas posibles al flujo es estrictamente 1.

Hay otros escenarios en donde el flujo de datos sí puede almacenarse, aunque puede resultar costoso el acceso a él; por ejemplo, si se piensa que el flujo consiste en elementos guardados en un disco al otro lado del mundo y accedido por medio de una red. En este escenario los datos se pueden consultar un número arbitrario de veces, pero cada consulta resulta costosa por lo que se podría tratar de acotar este número.

Así, en los algoritmos de flujo, usualmente se trata de acotar el número de pasadas necesarias al flujo para que el algoritmo encuentre una solución. En el modelo de semiflujo se permite que este número sea constante con respecto al tamaño del flujo, o incluso tenga un orden polilogarítmico con respecto al tamaño del flujo (n). En este trabajo, todos los algoritmos tienen un número constante de pasadas con respecto al tamaño del flujo, sin embargo, en muchos de ellos este número depende de otros parámetros.

En la sección 1.2 se presenta un algoritmo sencillo en el modelo de semiflujo que permitirá tener más en claro el funcionamiento del modelo y las limitaciones del mismo.

1.1.1. Un Flujo de Aristas

Un flujo de datos puede estar formado por prácticamente cualquier cosa: números, caracteres, parejas de elementos, etc. Sin embargo, se espera que cada elemento del flujo sea muy pequeño (de orden constante con respecto al tamaño del flujo) y más aún que quepa en unas pocas palabras de la computadora. Por ejemplo se pueden considerar flujos que contengan números enteros o de punto flotante, pero no se consideran válidos aquellos que contengan listas ligadas o árboles binarios, es decir, no se consideran flujos válidos aquellos en donde cada elemento sea un objeto complejo.

En este trabajo se presentan varios algoritmos de flujos que ilustran diversas técnicas de diseño, para propósitos ilustrativos, la entrada de estos (el flujo) representa a las aristas de una gráfica $G = (V, E)$. El conjunto E de aristas se puede modelar como una colección de parejas de números enteros (u, v). Esta representación tiene la ventaja de que solo se necesitan dos palabras de la computadora para modelar a cada elemento del flujo.

El flujo consta de una secuencia, en un orden arbitrario, de parejas de números; donde cada pareja representa a una arista de alguna gráfica. El tamaño (número total de vértices) y el orden (número total de aristas) de la gráfica podrían desconocerse o no, de acuerdo al problema en específico. A lo largo del texto se utilizará la letra n para referirse al número de vértices de una gráfica y m para el número de aristas. Por lo general se conocerá el tamaño de la gráfica (n) antes de empezar a leer el flujo. Esto facilita la comprensión y diseño de los algoritmos. En caso de desconocer estos datos, en el modelo de semiflujo, se puede dar una pasada previa al flujo para obtenerlos.

1.2. α-Generador

En esta sección se presenta un primer algoritmo de flujos que actúa sobre gráficas. Este algoritmo trata de ilustrar el esquema general de los algoritmos presentados en este trabajo, así como ejemplificar los puntos que se deben tener en cuenta al diseñar un algoritmo de flujo.

Para presentar el problema a resolver es necesario dar unas definiciones de teoría de gráficas:

Dada una gráfica $G = (V, E)$ una *trayectoria* entre dos vértices (u, v) es una secuencia ajena de aristas distintas $e_1, e_2, \ldots, e_k$ donde $e_1 = (u, v_1), e_2 = (v_1, v_2) \ldots, e_k = (v_{k-1}, v)$. Una *componente conexa* de G es un subconjunto $V' \subseteq V$, $V \neq \emptyset$ tal que para cualesquiera $x, y \in V'$, existe una trayectoria entre x e y. Una gráfica es conexa si y solo si el número de componentes conexas es uno.

La *distancia* en G del vértice u a v se denota como $d_G(u, v)$ y es k si la trayectoria más corta que los une tiene k aristas. Dos vértices adyacentes están a distancia 1 y si no están conectados, es decir, no existe una trayectoria que los una, entonces "están" a distancia ∞.

Lo que se querría sería determinar exactamente la distancia entre cualesquiera dos vértices de G. Observe que si se quiere determinar exactamente la distancia entre dos vértices dados, podría ser necesario recorrer muchas aristas de la gráfica (por ejemplo, si esta es solo una trayectoria). Sin embargo, si se quiere calcular la distancia entre cualesquiera dos vértices, es necesario almacenar al menos todas las aristas, pero ya se dijo en la sección anterior que no se permite almacenar todo el flujo, por lo que se debe cambiar la estrategia.

La idea será entonces aproximar estas distancias y con esto en mente se define un tipo de subgráfica de G.

Definición 1.1 (α-generador)**.** Dada una gráfica G y un valor $\alpha > 1$, una subgráfica H de G es un $\alpha-$*generador* de G si $\forall u, v \in V(G)$:

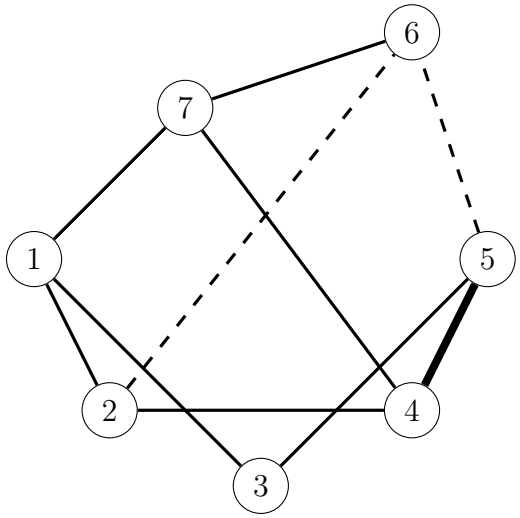
$$d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v)$$

A partir de la definición se puede observar que un α-Generador es una subgráfica que acota las distancias originales de G para cualesquiera dos vértices. Evidentemente, si se remueven aristas de G, puede que la distancia entre dos vértices aumente demasiado, pero al considerar un α-Generador, este número está acotado por un factor α, por ejemplo, si $\alpha = 2$, entonces se permite que las distancias se dupliquen.

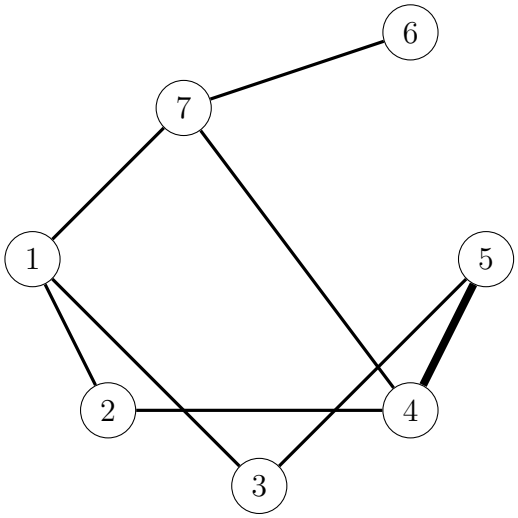
En la figura 1.1 se muestra una gráfica G con siete vértices y un $3-$generador ($\alpha = 3$) obtenido a partir de un flujo compuesto por las aristas de G en un orden particular.

En el algoritmo 1.1 se presenta, en el modelo de semiflujo, un algoritmo que calcula un α-generador para una gráfica G. Recuerde que el flujo consiste en una secuencia, en un orden arbitrario, de todas las aristas de G y se procesan una a una en el orden dado. Cada arista se representa como una pareja de números enteros entre 1 y $|V(G)|$. También recibe un factor α que determina qué tanto se pueden alejar las distancias en el generador de las originales.

El análisis de un algoritmo de flujo es un poco más detallado que el usual para un algoritmo secuencial convencional, ya que, además de la corrección y el número total de operaciones realizadas, importa también el tiempo de procesamiento por cada elemento del flujo, el número de pasadas que se necesiten dar y también se considera el espacio adicional requerido. En los siguientes capítulos se enuncian teoremas que engloban todos estos conceptos y se argumenta la corrección de los algoritmos. Para este problema, por ser de carácter introductorio, solo se establecen algunas propiedades de manera un tanto informal.



(a) Una gráfica G de la cual se obtendrá un 3−generador.



(b) 3−Generador de G.

Figura 1.1: En la gráfica izquierda se encuentra una gráfica G con 7 vértices a la cual se le obtendrá un 3−generador utilizando el algoritmo de flujo. El flujo, en este ejemplo, consta de las aristas en el siguiente orden: $\{(1,7),(1,3),(4,7),(2,4),(1,2),(6,7),(3,5),(4,5),(2,6),(5,6)\}$
Las aristas continuas forman parte del 3−generador, la arista más gruesa ($(4,5)$) es una que se agrega para cumplir la condición de la distancia ya que cuando el algoritmo la procesa, la distancia entre 4 y 5 es mayor a tres.
En la gráfica derecha se encuentra un 3-generador para G. Las aristas punteadas no forman parte de este pues cuando se presentan en el flujo ya no son necesarias.

Algoritmo 1.1 α-generador de una gráfica G.

Entrada S es un flujo de aristas que define a una gráfica no dirigida y conexa G y un valor $\alpha > 1$.
Salida Una subgráfica H que es un $\alpha-$generador de G.

```
H ← ∅.
for (u, v) ∈ S do      //Cada arista se lee en el orden dado por el flujo.
    if d_H(u, v) > α then
        H ← H ∪ {(u, v)}.
    end if
end for
return H.
```

Primero observe la corrección del algoritmo: Sea $d = d_G(u, v)$, entonces se cumple que $d \leq d_H(u, v)$ pues H es una subgráfica de G y posiblemente tiene menos aristas que G por lo que la distancia entre dos vértices en H no puede ser más pequeña que la distancia de esos mismos vértices en G.

Ahora, $\forall u, v \in V(G)$ también es cierto que si $(u, v) \in H$, entonces $d_H(u, v) = 1 = d_G(u, v)$ ya que $(u, v) \in E(G)$. Si $(u, v) \notin H$ hay dos casos:

1. $(u, v) \in G$: En este caso, cuando el algoritmo de flujo procesa a (u, v) la descarta (pues no está en H), si la descarta es porque la distancia en H entre u y v ya es menor o igual que α, por lo que: $1 = d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v) = \alpha$

2. $(u, v) \notin G$: En este caso $d_G(u, v) = d > 1$. Sean $u = u_1, u_2, \ldots, u_d = v$ los vértices de una trayectoria más corta que une a u con v en G.

   Por cada arista $(u_i, u_{i+1}), 1 \leq i < d$ se cumple, por el caso anterior, que:

   $$1 = d_G(u_i, u_{i+1}) \leq d_H(u_i, u_{i+1}) \leq \alpha \cdot d_G(u_i, u_{i+1}) = \alpha$$

   Entonces, cada una de las aristas que forman una trayectoria más corta de u a v en G puede ser sustituida por una trayectoria de longitud a lo más α en H y hay d aristas, con lo que:

   $$d = d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d = \alpha \cdot d_G(u, v)$$

De este modo se observa que el algoritmo efectivamente entrega un α-Generador para G. El tiempo de procesamiento por arista está dado por el tiempo que tome calcular la distancia entre dos vértices en una gráfica G. Esta distancia puede calcularse mediante un algoritmo de exploración de búsqueda por amplitud (`BFS`). Existen técnicas más avanzadas que incluso procesan cada arista del flujo en tiempo constante como se presenta en [4]. En ese trabajo también se establecen cotas superiores para el espacio de almacenamiento suficiente para el problema, una de ellas se presenta a continuación.

Un *ciclo* es una trayectoria cerrada, es decir, un ciclo es una secuencia de al menos tres aristas distintas: $e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_k = (v_k, v_1)$, donde $v_1 \neq v_2 \neq v_3$ y en este caso, longitud del ciclo es k. A continuación se presenta una cota de almacenamiento basada en una propiedad de los ciclos de H.

El *cuello* de una gráfica es el ciclo de menor longitud que existe en la gráfica. El cuello de H es al menos $\alpha+2$ pues si se completa un ciclo con la arista (u, v) en H es porque al procesarla la distancia que ya había entre ellos era mayor a α, por lo que el ciclo ya tenía al menos $\alpha+1$ aristas, así que no puede haber un ciclo de longitud menor a $\alpha+2$ en H. Con esta idea en [6] el autor establece que toda gráfica G de n vértices y con cuello impar de la forma: $2t+1$ tiene a lo más: $O(n^{1+\frac{1}{t}})$ aristas.

Con lo anterior, si $\alpha = 2t-1$, con t, entero positivo, se tiene que la gráfica tendría a lo más: $O(n^{1+\frac{2}{\alpha+1}})$ aristas, así que con este espacio de almacenamiento adicional es suficiente para almacenar a H. Observe que este espacio es estrictamente menor que el orden de una gráfica en general, ya que hay gráficas con $O(n^2)$ aristas, por lo que el tamaño del flujo es mucho más grande que el espacio de almacenamiento requerido.

Para concluir, se debe recordar que en los siguientes capítulos siempre se buscará hacer entender primero cómo generar un algoritmo de flujos utilizando una técnica de diseño y con esa idea resolver o aproximar una solución a un problema dado. Los problemas con los que se trabajan son problemas de la teoría de gráficas, sin embargo, estas técnicas se pueden transportar a otros algoritmos para resolver otros problemas.

1.3. Algoritmos de Aproximación

En esta sección se presentan unos conceptos teóricos del Diseño de Algoritmos que son necesarios para razonar y comprender algunos algoritmos presentados en este trabajo.

Hay ocasiones en las que hallar una solución exacta u óptima a algún problema es muy costoso; sin embargo, hallar una aproximación o una solución cercana a la óptima es suficientemente bueno y mucho más eficiente. En particular para algunos problemas NP-Completos se pueden crear algoritmos, tal como lo muestra [8, capítulo 35], que aproximen una solución óptima y que sean mucho más rápidos que aquellos que calculen directamente una solución.

En el caso de los algoritmos de flujos, a veces, es imposible o simplemente no conviene encontrar una solución exacta u óptima, ya sea porque se requieren muchas pasadas al flujo, porque el espacio de almacenamiento necesario excede el límite permitido o porque es más simple (en términos de complejidad algorítmica) hallar una aproximación. En esta sección se definen lo que son estos algoritmos y la manera usual con la que se trabaja con ellos.

Los algoritmos de aproximación por lo general se aplican a problemas de optimización en donde se quiere maximizar o minimizar una función objetivo. En [8, capítulo 35] se definen los algoritmos de aproximación considerando que toda posible salida puede medirse mediante un costo C. La definición es la siguiente:

Definición 1.2 (Radio de aproximación para un algoritmo)**.** Se dice que un algoritmo para un problema P tiene un *radio de aproximación* $\rho(n)$ si para cualquier entrada de tamaño n, el costo C de la salida producida por el algoritmo está dentro de un factor $\rho(n)$ de una solución óptima de costo C^*, es decir:

$$\text{máx}\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Si un algoritmo alcanza un radio de aproximación $\rho(n)$, se le llama un algoritmo de $\rho(n)$-*aproximación*. La definición anterior aplica tanto a problemas de minimización como a problemas de maximización. Si el problema trata de minimizar la solución, entonces una solución aproximada será mayor que la óptima y de forma análoga, en uno de maximización se encontrará una solución menor a la óptima.

A veces, estas soluciones se pueden acercar tanto como sea requerido a una óptima. Este requerimiento proviene de un parámetro ϵ que es ingresado como entrada adicional al algoritmo y por lo general, es independiente del tamaño de la entrada del algoritmo. Con esto se define lo que es un *esquema de aproximación*.

Definición 1.3 (Esquema de aproximación para un problema de optimización)**.** Un *esquema de aproximación para un problema* P es un algoritmo de aproximación que toma como entrada un ejemplar del problema P y además, un factor $\epsilon > 0$, tal que para toda ϵ fija, el esquema es un algoritmo de $(1 + \epsilon)$-aproximación.

Cuando se trabaja con estos esquemas la complejidad temporal de los algoritmos, por lo general, depende tanto del tamaño de la entrada n, como del *factor de aproximación* ϵ, aunque ϵ muchas veces no depende de n. Si el algoritmo se ejecuta en tiempo polinomial con respecto a n, se dice que el esquema de aproximación se ejecuta en tiempo *polinomial*. Si el tiempo de ejecución del algoritmo es polinomial con respecto a ambos factores (n y ϵ), entonces el esquema es *completamente polinomial* en tiempo.

Como ejemplo: en [20] se encuentran varios algoritmos que son esquemas de aproximación con radios $(1 + \epsilon)$. Esto quiere decir que si el problema es de minimización, entonces conforme ϵ tiende a cero, el esquema encuentra una aproximación más y más cercana a la óptima, pero siempre con costo mayor. También puede haber esquemas de aproximación, donde se encuentre una $k\epsilon$-aproximación, esto implica que conforme ϵ tiende a cero, entonces el factor de aproximación tiene a k y si ϵ tiende a infinito, entonces la solución hallada se aleja más y más de una óptima.

1.4. Algoritmos Probabilistas

Un algoritmo probabilista da garantías en cuanto a su salida con una probabilidad dada (recordar que las probabilidades están acotadas entre 0 y 1). Por ejemplo si algoritmo A para un problema P da como salida S con probabilidad p quiere decir que con probabilidad p la salida S es la correcta para el algoritmo y con probabilidad $1 - p$ el algoritmo emitirá una salida no válida.

Este tipo de algoritmos son útiles cuando conocer una solución exacta (con probabilidad 1) es muy costoso o hasta imposible. En este trabajo se presentan algunos algoritmos que funcionan así y que entregan soluciones coherentes con cierta probabilidad.

En el modelo de semiflujo hay problemas para los cuales no solo no se obtiene una solución exacta (sino una aproximación) sino que tampoco se tiene certeza que la aproximación sea correcta (probabilidad) por lo que ambos conceptos se juntan creando el siguiente tipo de

algoritmos.

Definición 1.4 (Algoritmos de (ϵ, δ)-aproximación)**.** Sea P un problema de optimización computable en el modelo de flujo, sea A un algoritmo que obtiene una ϵ-aproximación de una solución óptima para P, $\epsilon > 0$ y lo hace con probabilidad $1 - \delta$, $\delta > 0$. Se dice que A es un algoritmo que obtiene una (ϵ, δ)-*aproximación* del problema P.

Esta definición quiere decir que el algoritmo trabaja de modo que hallará una aproximación con rango menor o igual a ϵ solo con probabilidad $1-\delta$, por lo que si la δ tiende a cero, entonces la probabilidad de éxito crece. Observe que incluso aunque δ sea casi cero, el algoritmo sigue siendo de aproximación. Si ϵ también tiende a cero, entonces, casi siempre se hallaría una solución muy cercana a la óptima.

2. Gráficas Bipartitas

Una técnica muy simple para desarrollar algoritmos de flujos es adaptar un algoritmo ya existente en el modelo convencional para que acepte como entrada un flujo. Esto implica adaptar los tiempos de procesamiento por unidad de entrada, así como también el espacio de trabajo del algoritmo para que estos parámetros estén en los rangos permitidos por el modelo. Además se debe considerar que no se tiene a disposición la entrada completa por lo que no se pueden utilizar técnicas de preprocesamiento como ordenar la entrada o almacenarla de cierta manera.

Con lo anterior en mente, en este capítulo se presenta un algoritmo de semiflujo que resulta después de hacer transformaciones a uno en el modelo convencional, el algoritmo que se presenta aquí determina si una gráfica es o no bipartita.

De nuevo, en el flujo se encuentran las aristas de una gráfica simple. El algoritmo obtendrá como salida una bipartición de la gráfica o dirá que no es bipartita.

Las *gráficas bipartitas* o *2-coloreables* son una familia de gráficas muy estudiada y sobre la que se conocen resultados como el teorema de König [23, sección 2.1] o propiedades de gráficas perfectas [23, sección 5.5]. Una pregunta básica acerca de esta familia de gráficas es la de conocer si una gráfica dada es bipartita o no. En computación se estudia la parte algorítmica de esta pregunta, es decir, se estudian algoritmos que decidan (respondan) esta pregunta a partir de una gráfica dada como entrada al algoritmo. De acuerdo a la definición de esta familia se debe encontrar una 2-coloración por vértices para la gráfica, lo que significa definir una partición de los vértices en dos conjuntos ajenos no vacíos, donde todas las aristas tengan un extremo en cada conjunto.

Una caracterización de estas gráficas propuesta y demostrada por [23, proposición 1.6.1] dice lo siguiente:

Teorema 2.1 (Gráficas Bipartitas)**.** *Una gráfica $G = (U, V)$ es bipartita si y solo si G no tiene ciclos de longitud impar.*

De este resultado surge la idea de un algoritmo que consiste en verificar si la gráfica tiene un ciclo de longitud impar. En caso negativo, se obtiene como salida que la gráfica es bipartita y se calcula la bipartición, en caso afirmativo se sabe que no es bipartita. En la primera sección del capítulo se presenta un algoritmo en el modelo convencional que resuelve el problema y se discute cómo se puede transformar este algoritmo para que funcione en el modelo de semiflujo.

En la segunda sección se habla más a fondo y en detalle acerca de la creación del algoritmo de flujo y se presenta un pseudocódigo que concreta las ideas discutidas, también se demuestran las propiedades del algoritmo que son relevantes en el modelo de semiflujo.

2.1. Discusión del Problema y Algoritmo Convencional

La estrategia del algoritmo es verificar que todos los ciclos que tenga la gráfica sean de longitud par, en el proceso también se halla una bipartición, de modo que cuando el algoritmo termina, arroja como salida una bipartición de la gráfica.

Para mantener la bipartición se le asignará un color simbólico a cada vértice, de entre los dos posibles (rojos y azules), pues se requiere una 2-coloración. Este color indicará el conjunto de la partición al que pertenece ese vértice. Si se logra que todas las aristas tengan un vértice de un color y el otro del color contrario, entonces la gráfica es bipartita y un conjunto de la partición está formado por todos vértices rojos y el otro por todos los azules. Observe que si la gráfica es bipartita, entonces los colores se pueden intercambiar, es decir, que se pueden pintar a todos los rojos de azul y viceversa y la coloración sigue siendo válida.

El algoritmo irá tomando un vértice a la vez e irá coloreándolo de acuerdo a como le convenga, al inicio puede iniciar en cualquier vértice y pintarlo de rojo, a partir de ese momento, para que la coloración siga siendo válida, deberá pintar a todos los vecinos del primer vértice de color azul y a los vecinos de estos de rojo y así sucesivamente hasta haber pintado a todos los vértices. Como la gráfica está codificada en la memoria, se puede acceder a los vecinos de todos los vértices y hacer este proceso. Observe que si la gráfica no es conexa, entonces puede repetir este proceso en cada una de las componentes.

Al final del proceso anterior, cuando se han pintado a todos los vértices, se habrá obtenido una 2-coloración de la gráfica. Se debe determinar si es una coloración válida, es decir, si toda arista cumple que tiene un extremo rojo y el otro azul. En caso afirmativo se habrá obtenido una bipartición. Además, en este proceso se fue construyendo un árbol desde el vértice inicial s y este árbol corresponde a explorar la gráfica desde s mediante una búsqueda por amplitud.

La búsqueda por amplitud es uno de los tipos de búsqueda más utilizados en los algoritmos y se le conoce, por sus siglas en inglés, como BFS (*Breadth First Search*). El recorrido BFS inicia en un vértice arbitrario de la gráfica y lo considera la raíz de un árbol de exploración, a continuación, recorre cada uno de los vecinos y los añade al árbol. Estos serán vecinos de s y por cada uno de los vecinos de s recorre sus vecinos y los agrega al árbol como vecinos de los vecinos de s (si no estaban ya en el árbol) y así sucesivamente hasta agotar a todos los vértices de la gráfica (si es conexa). El árbol que se va construyendo se va formando por niveles, es decir, primero se llena el nivel cero con s, después el 1 con los vecinos de s, después el 2 con los vecinos de los vecinos de s y así hasta terminar.

El proceso de colorear alternadamente los vértices de la gráfica se puede implementar mediante el algoritmo de BFS y al terminar se tiene una coloración. Sin embargo, observe qué ocurre si al final se tiene a dos vértices u, v a la misma distancia de la raíz tales que son vecinos. Considere a z como el nodo en el árbol de exploración que es ancestro de u y de v, pero además está lo más alejado posible de s, a z se le conoce como *el ancestro común más cercano de u y v*.

A partir de z hay un camino de u y otro completamente distinto a v, sin embargo como u

y v están a la misma distancia de s y por lo tanto de z, en cada uno de estos caminos hay, digamos k aristas, pero u y v son vecinos, entonces hay una arista entre ellos, por lo tanto hay un ciclo en G formado de la siguiente manera: el camino de z a u, luego la arista (u, v) y después el camino de v a z. Observe que el número de aristas que tiene el ciclo es impar, pues tiene $2k$ por las aristas de los caminos entre z y u y entre z y v más la arista que une a u con v. Por lo tanto G no es bipartita.

Observe, entonces, que como u y v están a la misma distancia de s (el nodo raíz), entonces por el proceso de coloreo, deben tener el mismo color. Así que si se encuentra una arista coloreada con ambos extremos del mismo color, por la explicación anterior, se ha encontrado un ciclo de longitud impar en G, por lo que no es bipartita.

El algoritmo presentado en 2.1 fue construido a partir de la discusión presentada en [17, capítulo 3,sección 3.4] y en esa referencia se pueden consultar más detalles de la demostración de corrección, así como de la complejidad. El algoritmo está basado en la implementación de BFS propuesta en la misma referencia.

Algoritmo 2.1 Algoritmo para determinar si una gráfica G es bipartita en el modelo convencional.

Entrada Una gráfica G tomada como entrada y un vértice s desde el que empezará el recorrido.

Salida Una lista de vértices con colores rojo o azul.

```
for v ∈ V(G) do
    discovered[v] ← False.      //Vértices por explorar
    Color[v] ← red.             //Inician con un color arbitrario
end for
discovered[s] ← True.
L[0] ← s.                       //Lista de elementos a revisar
i ← 0.                          //Nivel actual del árbol
while L ≠ ∅ do
    u ← deque(L).
    for (u, v) ∈ E(G) do        //Se revisan los vecinos de u
        if discovered[v] = False then
            discovered[v] ← True.
            queue(L, v).
            if i  mód 2 = 1 then
                color[v] ← blue.    //Si el nivel es impar se cambian de color
            end if
        end if
    end for
    i ← i + 1.
end while
return color.
```

El algoritmo devuelve una lista de todos los vértices de G donde cada uno tiene un color asignado: rojo o azul. Lo único que falta ver es si dos vértices adyacentes tienen el mismo color, en cuyo caso la gráfica no es bipartita, debido a las observaciones anteriores. Esta parte

del código se omite.

Con este algoritmo en mente se quiere crear uno que resuelva el mismo problema en el modelo de semiflujo pero para ello se debe entender la parte fundamental de este algoritmo e intentar restringir o adaptar los pasos del mismo al modelo.

La idea fundamental del algoritmo consiste en obtener una 2-coloración de los vértices de la gráfica y finalmente ver si esta 2-coloración es útil o contiene dos vértices adyacentes del mismo color, en el segundo caso, la gráfica no es bipartita pues se ha hallado un ciclo de longitud impar. En el modelo de flujos se podría intentar también hallar una 2-coloración válida, sin embargo la estrategia debe cambiar ya que no se conoce a todos los vecinos de un vértice de antemano.

Por la forma en la que se procesa a la gráfica en el modelo de flujos, esta se va descubriendo cada que se procesa una arista, por lo que la idea de hallar una 2-coloración debe seguir esta premisa de modo que cada que se "conoce" una nueva arista de G, la coloración actual sea compatible con este descubrimiento o se modifique para que siga siendo válida. También se debe detectar cuando no se pueda modificar debido a que se ha encontrado un ciclo de longitud impar.

Si se piensa que la gráfica se va descubriendo conforme se procesan las aristas, se puede pensar que se inicia con una gráfica compuesta por solo los vértices y las aristas se irán agregando conforme se procesa el flujo. Con esta idea, es claro que una arista puede unir dos conjuntos de vértices que eran ajenos anteriormente, es decir, unir dos componentes conexas o puede unir dos vértices que ya se encontraban en la misma componente. En cada uno de estos casos se debe decidir qué hacer para mantener una coloración válida.

Los vértices se pueden separar en dos conjuntos: aquellos del color rojo y aquellos del color azul. Al inicio del algoritmo todos los vértices forman parte del mismo conjunto, entonces esta partición irá cambiando al procesar cada arista y los vértices pueden cambiar de color mientras se preserve la propiedad de que toda arista tiene un extremo de un color y el otro del color contrario. Esto es una propiedad que el algoritmo debe cumplir. A esta se le llamará la *propiedad de color*.

Ya que existen gráficas bipartitas con n vértices que tienen $O(n^2)$ aristas, el algoritmo debe de almacenar solo la información necesaria y no todas las aristas. Lo que se necesita almacenar en realidad es la bipartición y las aristas solo sirven para determinar a qué conjunto pertenece cada uno de sus extremos.

En el algoritmo se mantiene una estructura de datos en memoria que permite determinar fácilmente si dos vértices dados están o no en la misma componente conexa del bosque procesado hasta ese momento, esta estructura también permite conocer el color actual de cada uno de los vértices. La estructura se irá modificando a lo largo del proceso pero siempre mantendrá un tamaño permitido.

Cuando se procesa una arista del flujo, se están conectando dos vértices o que estuvieran separados, es decir, en componentes conexas distintas o que estuvieran en la misma. En el primer caso, se están uniendo dos componentes en una y en el segundo, se está formando un ciclo, pues si ambos están en la misma componente, hay un camino que los une y si a este se le agrega la arista procesada, se completa un ciclo. Estos casos, por supuesto, son ajenos y a

continuación se analizan.

Primero se tratará el caso cuando se forma un ciclo: al procesar la arista, esta une dos vértices en la misma componente conexa. Sea $e = (u, v)$ la arista procesada que completa el ciclo $C = c_1c_2 \dots c_nc_1$ (recordar que este ciclo no se mantiene en memoria pero se puede determinar si existe o no, fácilmente, esto se explica con más detalle en la siguiente sección), donde $c_1 = u$, $c_n = v$. Si se supone la propiedad de color, entonces en C se tiene una sucesión de colores que empieza (s.p.g.) con el color rojo perteneciente a u. Si v tiene color azul, entonces al agregar la arista e se mantiene la propiedad, pero observe cuál es la paridad de la longitud de C. Como la sucesión empieza en rojo, cuando se recorre la primera arista se llega al color azul, cuando se recorre la segunda se llega de nuevo al rojo y aquí se observa el patrón: cada que se recorre una arista impar se llega al color azul. Como v tiene color azul, entonces en el camino $c_1c_2 \dots c_n$ hay un número impar de aristas, así que al agregar a e y formar C se agrega una arista más, por lo que C tiene longitud par.

Por otro lado, si v también tiene color rojo entonces, siguiendo el argumento anterior, la longitud del ciclo es impar. Aquí se puede establecer otra propiedad. El algoritmo debe cumplir que si la gráfica no es bipartita, entonces se debe hallar un ciclo de longitud impar. A esta propiedad se le llamará la *propiedad de paridad*.

En el caso de que se procese una arista en donde se unen dos vértices pertenecientes a componentes conexas distintas, se observa que no se forman ciclos y todos los ciclos de ambas componentes son de longitud par debido a la propiedad observada anteriormente. Entonces, si se tiene un ciclo $C = c_1c_2 \dots c_nc_1$ de longitud par con colores alternantes, estos se pueden intercambiar preservando la propiedad de color. Por lo que si ambos extremos de $e = (u, v)$ son del mismo color, entonces en una de las dos componentes se intercambian primero los colores de los vértices y después se procesa la arista y se unen estas dos componentes en una sola. Esta nueva componente cumple con la propiedad de color y no se agregó ciclo alguno, por lo que no se ha descubierto ningún ciclo de longitud impar.

Observe que esta idea de intercambiar las coloraciones para que sigan siendo válidas no es necesaria en el algoritmo del modelo convencional, esto es porque en él se tiene perfectamente claro quienes son todas las aristas incidentes a un vértice dado, incluso existen codificaciones de gráficas que se enfocan en responder justamente esa pregunta de manera eficiente. Esta propiedad hace que se pueda establecer claramente que si un vértice es de color rojo, entonces todos sus vecinos deben ser de color azul y se colorean en ese momento y se sabe que esa coloración ya creada no puede cambiar a lo largo de la ejecución. Sin embargo, en el modelo de flujos, al no conocer a todos los vecinos de un vértice, pudiera ser que no conviene colorearlo de un color sino del otro y esto podría cambiar después, por eso se debe considerar intercambiar los colores de los vértices en reiteradas ocasiones.

Con las ideas discutidas hasta el momento se tiene claro que se puede considerar un algoritmo en el modelo de semiflujo que determine si una gráfica es bipartita o no, adaptando la idea fundamental de tener una 2-coloración válida para los vértices, con la diferencia de que esta se irá modificando a lo largo de la ejecución, en lugar de irse construyendo como se hace en el modelo convencional. En la siguiente sección se formalizan estas ideas y se describe el algoritmo detalladamente.

2.2. El Algoritmo de Flujos

En esta sección se formalizará el algoritmo de flujo utilizando las ideas descritas previamente y también se analizará la eficiencia del mismo teniendo en cuenta todos los criterios para este tipo de algoritmos.

Al inicio del algoritmo todos los vértices estarán en componentes conexas distintas (tantas como vértices haya). El algoritmo tomará una arista del flujo a la vez y verificará si sus extremos son parte de la misma componente conexa o no. En caso afirmativo se verificará que el ciclo que se forme tenga longitud par. Aunque no es necesario calcular la longitud del ciclo debido a lo siguiente: si los extremos de la arista tienen el mismo color, entonces el camino que ya se tiene es de longitud par y también se cumple la inversa pues si el camino que ya los une tiene longitud par, entonces, por la propiedad de color, ambos extremos tendrán que tener el mismo color. Por lo tanto, no se necesita calcular la longitud del ciclo, sino solo conocer el color de ambos extremos. Si es el mismo, entonces el algoritmo termina y se garantiza que la gráfica no es bipartita (se ha encontrado un ciclo de longitud impar); si son distintos, se omite esta arista y no se almacena información adicional alguna.

En caso de que los extremos estén en componentes conexas distintas, se procede a comparar los colores de estos. Si son iguales, se escoge la componente con el menor número de elementos y se intercambian los colores de todos ellos; al mismo tiempo, se establece que todos ellos forman parte de la misma componente que el otro extremo. Si los colores son distintos, solo se reetiquetan a los vértices de aquella componente con menor número de elementos para indicar que ahora forman parte de una sola componente.

El algoritmo necesita utilizar una estructura de datos que sea capaz de mantener las componentes conexas y unirlas conforme sea necesario. También debe mantener los colores de los vértices y cambiarlos de manera eficiente. Ademas, debe responder rápidamente si dos vértices están actualmente en la misma componente conexa o no. Estas operaciones se pueden conseguir de manera muy eficiente al utilizar la estructura de datos conocida como *conjuntos disjuntos* (*disjoint-sets*) [8, capítulo 21]. Esta estructura de datos admite las operaciones de creación de un conjunto unitario, unión de dos conjuntos y permite determinar si dos elementos pertenecen al mismo conjunto o no; todo lo anterior de manera eficiente.

En el algoritmo de flujo deseado se debe mantener un color por vértice, pero observe lo siguiente: En el transcurso del algoritmo se deben mantener componentes conexas ajenas y 2-coloreadas, es decir, si se considera una componente por separado, esta debe estar 2-coloreada por lo que se puede ver a su vez como 2 conjuntos disjuntos: los *rojos* y los *azules*. La unión (ajena) da como resultado a la componente completa. Con esta observación se puede simular la 2-coloración mediante un par de conjuntos disjuntos. Entonces cada componente está formada a su vez, por dos conjuntos.

Las operaciones deseadas son: la creación de una componente a partir de un solo elemento, la unión de dos componentes en una sola permitiendo el intercambio de colores de una de ellas y también se debe determinar si dos elementos (sin importar su color) pertenecen a la misma componente o no. Estas operaciones se pueden realizar de manera muy similar a las de los conjuntos disjuntos originales y se habla más acerca de ellas en el apéndice A.1. En la tabla 2.1 se resumen estas operaciones así como su costo en operaciones. La función $\alpha(n)$, presente

Cuadro 2.1: Tabla con las operaciones de la estructura de datos conjuntos disjuntos bicoloreados.

Nombre de Operación	Tiempo en peor caso	Tiempo amortizado
$make\_set(v)$	$O(1)$	$O(1)$
$find\_set(v)$	$O(\log(n))$	$O(\alpha(n))$
$component(v)$	$O(1)$	$O(1)$
$union(u, v)$	$O(\log(n))$	$O(\alpha(n))$

en la tabla, representa a la función inversa de la *función de Ackerman*, esta función α tiene un ritmo de crecimiento muy lento, por lo que un algoritmo o función con esta complejidad es muy eficiente. Para más detalles consultar [8, capítulo 21, sección 21.4].

En particular, cabe destacar que la operación de unión debe considerar el posible recoloreo de todo un uniendo, sin embargo este se puede hacer eficientemente de la siguiente manera: Suponga que se quiere unir la componente C_i formada por los 2 conjuntos C_{i1} y C_{i2} con la componente C_j formada por los 2 conjuntos C_{j1} y C_{j2}. Estos dos conjuntos representan a los vértices azules y rojos dentro de cada componente; suponga también que se deben intercambiar los colores de la componente C_j para que en la unión se preserve la propiedad de color. Entonces, en lugar de intercambiar el color de todos los vértices de C_j basta con indicar que todo el conjunto de vértices azules, ahora forma parte del conjunto de vértices rojos de $C_k = C_i \cup C_j$ pero este está formado por los vértices rojos de C_i junto con los vértices azules de C_j por lo que basta solo unir C_{i1} con C_{j2} y para el conjunto de vértices azules de C_k se hace algo análogo. Para lograr eficiencia estas uniones se pueden realizar de manera similar a la operación de unión en los conjuntos disjuntos tradicionales, es decir haciendo que el conjunto con menos elementos se una al que tiene más elementos. Observe que para determinar el color de un vértice dentro de una componente, basta con saber a cuál de los dos conjuntos pertenece y conocer el color del representante de ese conjunto. Hay que hacer notar que para que esta operación siga siendo eficiente se deben considerar algunos casos adicionales y tratarlos con cuidado. Estos casos se encuentran más detallados en el apéndice A.1.

En el algoritmo se utiliza la estructura descrita anteriormente, así como un par de arreglos adicionales. El primer arreglo almacena, por cada vértice en la gráfica, el nodo de la estructura que lo contiene. Este nodo se almacena durante la operación `make_set`. El segundo arreglo guarda el color de cada vértice. Este inicialmente se compone de puros `0` y se puede modificar durante la operación de unión. También se utiliza una variable que va nombrando las componentes conexas, esto debido a que se debe conocer si dos vértices pertenecen a la misma componente o no sin importar si pertenecen a su conjunto de vértices rojos o al de azules. Esta variable es un entero y se incrementa durante cada operación `make_set` y durante cada operación `union`. Debido a que solo se hacen n operaciones `make_set` y a lo más $n - 1$ operaciones de unión, el valor de esta variable está acotado.

En el Algoritmo 2.2 se presenta el proceso descrito previamente. En el pseudocódigo se utilizan la función `make_set(x)` que forma una nueva componente conexa que consta de dos conjuntos: uno unitario con el elemento x y el otro vacío. También se utiliza la función `find_set(x)` que devuelve el representante del conjunto al que pertenece x. La función `component(x)` devuelve la componente conexa a la que pertenece el elemento x. Observe que

una componente tiene dos elementos, por lo que a pesar de que dos nodos tengan distintos representantes, podrían estar en la misma componente. Finalmente, la función `union(x,y)` une los conjuntos que tienen como elementos a `x` y a `y` teniendo en cuenta las consideraciones descritas anteriormente.

Algoritmo 2.2 Algoritmo para calcular una bipartición de una gráfica en el modelo de semi-flujo.

Entrada S es un flujo de aristas E de una gráfica $G = (V, E)$.
Salida Una pareja de conjuntos indicando una bipartición de G o una leyenda indicando que la gráfica no es bipartita.

```
for v ∈ V do      //Preprocesamiento
    make_set(v).
end for
for e = (u, v) ∈ S do      //Procesamiento del flujo
    x = find_set(u).
    y = find_set(v).
    if component(x) == component(y) then
        if x == y then      //Se ha encontrado un ciclo de longitud impar
            return "La gráfica no es bipartita".
        end if
    else
        union(u, v).
    end if
end for
for v ∈ V do      //Posprocesamiento para obtener la bipartición
    x = find_set(v).
    if component(x).red == x then
        colors[v] = 0.
    else
        colors[v] = 1.
    end if
end for
return colors.
```

En el algoritmo se observa que se regresa o un arreglo con la bipartición (2-coloración) de la gráfica o se devuelve un aviso indicando que la gráfica no es bipartita. A partir del pseudocódigo se procede a demostrar que el algoritmo es correcto y se analiza el tiempo de procesamiento por arista, el espacio adicional utilizado y el número de pasadas al flujo, así como el número total de operaciones realizadas. Estos son los criterios que determinan la eficiencia de un algoritmo en el modelo de semiflujo.

Teorema 2.2 (Propiedades del algoritmo de semiflujo)**.** *El algoritmo 2.2 determina si una gráfica $G = (V, E)$ con n vértices, m aristas, almacenada en un flujo S es bipartita o no utilizando 1 pasada por el flujo, un tiempo de procesamiento, en el peor caso, $O(\log(n))$ por arista y un espacio adicional de $O(n \log(n))$ bits. Todo lo anterior realizando $O(m \log(n))$ operaciones.*

Este teorema se demostrará por partes en los párrafos siguientes, la corrección se deja al final.

Número de Pasadas
Evidentemente el número de pasadas es 1 pues se procesa cada arista del flujo una sola vez. Esto por supuesto es lo mínimo que se necesita, ya que se debe considerar toda arista de G al momento de determinar si hay un ciclo de longitud impar o no.

Tiempo de Procesamiento por Arista
Al procesar una arista se verifica si une vértices en la misma componente o no, para esto se deben determinar las componentes a las que pertenecen ambos extremos, cada una de estas operaciones toma a lo más $O(\log(n))$. Si están en la misma componente, se procede a verificar si tienen el mismo color, esto se hace en tiempo constante. Si no están en la misma componente, se procede a unir ambas. Por la forma en que se hace la unión, que es muy similar a la realizada por la estructura de conjuntos disjuntos, esta toma también tiempo $O(\log(n))$ ya que se deben encontrar a los representantes de ambos uniendo. De este modo, el procesar una arista toma a lo más $O(\log(n))$, aunque en tiempo amortizado toma $O(\alpha(n))$ como lo muestra la tabla 2.1.

Espacio de Almacenamiento
El espacio de almacenamiento necesario está dado por el requerido para alojar a la estructura de datos. En el apéndice se establece que una cota superior para almacenar una serie de conjuntos disjuntos de n elementos es $O(n\log(n))$ bits. Además se utilizan dos arreglos adicionales, el primero almacena direcciones de memoria y el de colores, bytes. Así el primero ocupa a lo más $O(n\log(n))$ bits y el segundo $O(n)$ bits. Así que el espacio total adicional está en el orden de $O(n\log(n))$ bits, por lo que la cota de almacenamiento es válida en el modelo de semiflujo. Observe que esta cota está establecida en términos del número de vértices de la gráfica y no en el número de elementos del flujo que es, posiblemente, mucho mayor.

Número total de operaciones
Dado que procesar cada arista toma a lo más $O(\log(n))$, entonces procesar todo el flujo toma $O(m\log(n))$ ya que hay m aristas en la gráfica, el crear las primeras n componentes y asignarles el color final a cada vértice requieren $O(n)$ operaciones, por lo que el número total de operaciones está en el orden de $O(m\log(n))$.

Por supuesto, todas estas garantías no sirven de nada si el algoritmo no es correcto. La demostración de la corrección de este se presenta a continuación.

Corrección del algoritmo Primero se demostrarán unas propiedades auxiliares. En ellas se demuestran las propiedades que el algoritmo cumple: la propiedad de color y la de paridad.

Lema 2.3 (propiedad de color)**.** *El algoritmo 2.2 cumple la propiedad de color; es decir si al procesar una arista $e = (u, v)$ del flujo el algoritmo no termina, entonces la subgráfica*

generada por los n vértices y todas las aristas procesadas hasta el momento tiene una 2-coloración.

Demostración.
Por inducción sobre el número de aristas.

Observe que en el caso base, como no hay aristas procesadas, entonces la gráfica generada consta solo de los n vértices y todos ellos tienen el mismo color, por lo que la 2-coloración es válida.

Suponga que al procesar las primeras k aristas del flujo, el resultado es cierto, esto es la Hipótesis de Inducción (HI).

Considere la arista $k+1$ formada por los vértices (u, v) en el flujo S y suponga que el algoritmo no termina. Al procesar la arista primero se verifica si los extremos están en la misma componente conexa o no. Aquí hay dos casos.

- Si lo están, entonces se procede a comparar los colores de u y v. Como el algoritmo no termina, entonces no pueden tener el mismo color, por lo tanto deben tener colores distintos. Y la coloración que se tenía para la subgráfica generada por las primeras k aristas sigue siendo la misma, por lo que sigue siendo válida.
- Si no están en la misma componente, entonces proceden a unirse ambas componentes en una sola. Al hacer la unión se debe verificar el color de ambos vértices. Si es distinto, el conjunto de vértices rojos de la unión es la unión de los conjuntos de vértices rojos y esto es análogo al de azules, por lo tanto ningún vértice cambió de color, así que por HI la subgráfica generada por los vértices y las primeras k aristas tiene una 2-coloración válida, al agregar entonces a la arista (u, v), como ambos tienen colores distintos entonces la coloración en esta nueva subgráfica es la misma, por lo que sigue siendo válida.

  El segundo caso es cuando u y v tienen el mismo color. Al hacer la unión, los conjuntos de los uniendos se cruzan, es decir: el conjunto rojo resultante está formado por el conjunto rojo de un uniendo y el azul del otro. Y el conjunto azul resultante se forma de manera análoga. En este caso se están intercambiando de color todos los vértices de una componente, pues los que eran parte del conjunto azul pasan a ser parte del rojo de la unión y viceversa. Al intercambiar los colores de toda una componente la coloración sigue siendo válida en toda la unión y en particular ahora u y v tienen colores distintos. Por lo que la nueva coloración es una 2-coloración válida para la subgráfica generada hasta el momento.

□

Con esto se concluye que al procesar todas las aristas del flujo, si el algoritmo no termina, la subgráfica resultante, que es la gráfica G, tiene una 2-coloración válida.

Ahora se demostrará la propiedad de paridad.

Lema 2.4 (Propiedad de paridad)**.** *Si la gráfica $G = (V, E)$ cuyas aristas forman el flujo S no es bipartita, entonces el algoritmo encuentra un ciclo de longitud impar.*

Demostración.
Suponga que G no es bipartita, entonces tiene al menos un ciclo de longitud impar. De todos estos ciclos considere a C como el primero en ser completamente procesado en el flujo S, es decir C es el primer ciclo de longitud impar que se puede formar con las aristas dadas en el orden en el que aparecen en S. Más aún, sea $e = (u, v)$ la última arista de C en aparecer en S.

Primero hay que observar que el algoritmo no ha concluido su ejecución antes de procesar a e. Esto es cierto ya que la única manera de terminar la ejecución de manera anticipada es que se procese una arista que una a dos vértices en la misma componente conexa y que estos tengan el mismo color en ese momento, pero si ocurre esto, entonces se está formando un ciclo $C' = c_1c_2 \cdots c_nc_1$. Sin embargo, como se cumple la propiedad de color, los vértices de C' deben tener colores distintos, pero c_1 tiene el mismo color que c_n por lo que debe haber un número impar de vértices en C', formando así un ciclo de longitud impar que se completa antes que C en S; contradiciendo la elección de C.

Entonces el algoritmo no ha concluido su ejecución y procede a procesar a e. Como ya procesó a todas las demás aristas de C, todos los vértices de C deben estar ya en la misma componente conexa, pues ya hay un camino que los une entre sí. Al procesar a e la primera comparación debe ser verdadera, pues tanto u como v están en la misma componente conexa. El ciclo C se puede ver de la siguiente manera: $C = c_1c_2 \cdots c_{2n+1}c_1$, donde $c_1 = u$ y $c_{2n+1} = v$. Como aquí hay un número impar de vértices y se debe mantener la propiedad de color, c_1 debe tener color distinto a c_2 que debe tener color distinto a c_3 y así hasta c_{2n+1}, pero entonces los vértices impares: $\{c_1, c_3, \ldots, c_{2n+1}\}$ tienen que tener el mismo color (pues solo hay dos colores posibles). Con esto se fuerza a que tanto u como v en este momento tengan el mismo color, por lo que al procesar a e el algoritmo terminará y concluirá que la gráfica no es bipartita pues habrá encontrado un ciclo de longitud impar.

Observe que la gráfica puede tener más ciclos de longitud impar pero a partir de esta demostración se concluye que siempre terminará con el primero que sea totalmente procesado según el orden de las aristas en el flujo. □

Con estos resultados se puede demostrar que el algoritmo es correcto.

Teorema 2.5 (Corrección del algoritmo de semiflujo)**.** *El algoritmo 2.2 procesa todo el flujo de entrada S si y solo si G es bipartita.*

Demostración.

⇒)
Si el algortimo termina de procesar todo el flujo de aristas, entonces por la propiedad de color, se tiene una 2-coloración válida de los vértices de la gráfica. Por lo tanto G es bipartita.

⇐)
Esto se demuestra por contrapositiva, es decir: si el algoritmo no procesa todo el flujo de aristas S, entonces G no es bipartita.

La única manera de que el algoritmo no procese todo el flujo es porque al procesar una arista, los extremos de esta estaban en la misma componente conexa y tenían el mismo color, pero como se muestra en la propiedad de paridad, esto implica que se ha encontrado un ciclo de longitud impar en G, por lo que la gráfica no es bipartita.

□

Con lo anterior se ha demostrado que el algoritmo propuesto determina si una gráfica es bipartita y además devuelve una bipartición en caso afirmativo. Este es un algoritmo en el modelo de semiflujo que resuelve un problema común y que puede ser utilizado como subrutina para encontrar una 2-coloración (bipartición) de una gráfica dada para después encontrar o calcular otras propiedades utilizando la bipartición como se verá en un capítulo posterior.

3. Algoritmos *Greedy*

En este capítulo se presenta una técnica muy conocida y usada en el diseño de algoritmos: la técnica *greedy*. Esta técnica es muy utilizada cuando se intenta resolver un problema de optimización, es decir, cuando se tiene una función de costos C asociada a los elementos de entrada (cada elemento tiene un costo no negativo).

En este tipo de problemas se debe encontrar una solución factible y que además maximice o minimice el costo total de esta con respecto a la función de pesos.

En teoría de gráficas se puede asociar a una gráfica $G = (V, E)$ una función de pesos w que recibe una arista de G y devuelve un número real distinto de cero, que es el peso de la arista. Dada una gráfica con pesos se puede plantear el problema siguiente: dados dos vértices, encontrar el camino de menor peso entre ellos. Este problema se puede resolver con el famoso algoritmo de Dijkstra, si los pesos son no negativos o con el algoritmo de Bellman-Ford si los hay. Existen otras soluciones al problema, pero estas dos son las más conocidas.

Otro ejemplo es el generalizar la noción de emparejamiento máximo en una gráfica y considerar el emparejamiento máximo por tamaño pero mínimo por costo, también se puede considerar encontrar, por ejemplo una subgráfica generadora de peso (costo) mínimo.

El problema de encontrar un *bosque generador de peso mínimo* para una gráfica con pesos se ha estudiado extensamente desde hace muchos años en el modelo convencional y es uno de los problemas clásicos en teoría de gráficas que se resuelven mediante un algoritmo. En muchos libros de teoría de gráficas y algoritmos es presentado este problema por su importancia en la noción de conexidad y optimización, en las referencias [8], [17] y [23] se pueden encontrar secciones o capítulos que hablan de este problema.

En la primera sección se presenta la técnica de diseño *greedy* en el modelo convencional, así como la idea para llevar a cabo esta técnica en un algoritmo de flujo. En la segunda sección se presenta un algoritmo para resolver el problema del bosque generador de peso mínimo para una gráfica en el modelo de semiflujo, así como demostraciones y argumentos que hablan acerca de la corrección y complejidad del algoritmo. Finalmente, en la tercera sección se presenta una generalización del algoritmo de la sección 2 para un concepto combinatorio llamado *matroide* que es más general que una gráfica. En esta sección se habla también de este concepto y se presentan algunos resultados en la teoría de matroides que ayudan a comprender mejor a estos objetos y facilitan las demostraciones del algoritmo.

3.1. Técnica *greedy*

En esta sección se presenta una técnica para intentar resolver problemas de optimización. Como ya se mencionó, esta técnica es muy famosa y socorrida al lidiar con estos problemas.

La idea detrás de crear algoritmos que optimicen soluciones es que estos sean capaces de tomar decisiones en el transcurso de su ejecución. En la técnica *greedy* se consideran estas

decisiones de manera local, es decir, con solo la información procesada hasta el momento, en lugar de considerar el resto de la entrada. Esta forma de tomar decisiones da como resultado algoritmos simples y concretos. Sin embargo, esta técnica no siempre garantiza que estas decisiones locales tengan como resultado una solución óptima de manera general.

La técnica *greedy* se caracteriza porque una vez que se tomó una decisión y se generó una solución parcial, la decisión no es reconsiderada en un futuro. Como dicen los autores en [17]: "Los algoritmos *greedy* crean soluciones en pequeños pasos, tomando una decisión en cada uno de ellos de manera miope para optimizar un criterio subyacente".

Ellos también mencionan que es complicado definir exactamente a qué se refiere esta técnica, sin embargo es fácil compararla con otra llamada *programación dinámica*. La técnica de programación dinámica también se utiliza para encontrar soluciones factibles en problemas de optimización, sin embargo consiste en adquirir estas soluciones poco a poco pero recordar todas las mejores decisiones en cada paso, de modo que estas sirvan en pasos futuros. Esta técnica requiere recordar (guardar en memoria) varias soluciones parciales para que al momento de tomar una decisión futura se utilicen algunas (o muchas) de estas soluciones parciales.

En la técnica *greedy* solo se almacena una solución parcial y con base en esta y en los criterios del algoritmo se crea una solución más general en cada paso hasta obtener una solución total.

Hay muchos problemas para los que se ha ideado un algoritmo en el que esta técnica da una solución óptima, sin embargo hay otros para los cuales los algoritmos propuestos solo aproximan una solución óptima.

Un ejemplo de un algoritmo *greedy* ya visto, es el usado para calcular un emparejamiento máximo en una gráfica. Las gráficas usadas para este problema no tienen pesos, pero se puede pensar que los pesos de las aristas son 1, por lo que un emparejamiento máximo es equivalente a uno máximo y de peso mínimo. El algoritmo *greedy* visto anteriormente en este capítulo encuentra un emparejamiento maximal, donde se garantiza que el tamaño es al menos la mitad del tamaño de un máximo. Este algoritmo es *greedy* debido a que siempre se mantiene solo un emparejamiento, se empieza con uno vacío y se va agrandando y solo se considera el estado de la arista procesada actualmente con respecto al emparejamiento almacenado hasta el momento. Esta idea es clásica en la técnica pues una vez que se acepta o rechaza una arista ya no se vuelve a considerar, en términos coloquiales: en estos algoritmos no existe el "hubiera".

Otro ejemplo clásico de un algoritmo *greedy* es el algoritmo de Dijsktra para encontrar el camino más corto (de menor costo total) entre dos vértices. Este es uno de los algoritmos más famosos en gráficas y utiliza esta técnica en su diseño de modo que las decisiones tomadas en la ejecución son locales y no se rectifican. En este caso, esta técnica lleva a una solución óptima, es decir, el camino reportado por el algoritmo siempre es uno de peso mínimo.

3.2. Bosque Generador de Peso Mínimo

En esta sección se presenta un algoritmo *greedy* en el modelo de semiflujo para resolver otro de los problemas de optimización más comunes en teoría de gráficas: encontrar una subgráfica generadora, conexa minimal y de menor costo, a esta gráfica se le conoce como un *bosque generador de peso mínimo*.

El problema de hallar un bosque o árbol (si la gráfica es conexa) generador de peso mínimo ha sido muy estudiado y presentado en prácticamente todo libro de teoría de gráficas o algoritmos. En particular hay varios algoritmos famosos que resuelven este problema como el algoritmo de Prim y el de Kruskal. A continuación se describe el segundo.

El algoritmo de Kruskal debe su nombre a Joseph Kruskal quien lo presentó en [18]. En el documento citado se describe una construcción que es similar a la que se utilizará en semiflujo y consiste en ir eligiendo una arista más conveniente en cada paso. Esta estrategia es *greedy* y consiste en tomar todas las aristas no procesadas y escoger la más pequeña de ellas que de como resultado un bosque, es decir que no forme ciclos con ninguna de las aristas ya consideradas en el árbol.

La idea de escoger una arista más pequeña de entre todas en cada paso, sugiere que es conveniente ordenarlas primero por pesos y esta es la forma clásica de presentar el algoritmo. El pseudocódigo se presenta en 3.1.

Algoritmo 3.1 Algoritmo de Kruskal.

Entrada Una gráfica $G = (V, E)$, con una función de pesos w.
Salida Un bosque generador de peso mínimo para G.

```
F ← ∅.
for v ∈ V(G) do
    make_set(v).
end for
Ordenar las aristas de G en orden creciente por pesos.
for e = (u, v) ∈ E(G) tomadas en orden do
    if find_set(u) ≠ find_set(v) then
        F ← F ∪ {e}.
        union(u, v).
    end if
end for
return F.
```

El algoritmo de Kruskal entrega un bosque generador de peso mínimo y es eficiente debido a que ordena las aristas de G. Esto hace que las decisiones acerca de una arista particular sean más fáciles de tomar pues basta saber si esa arista forma un ciclo o no con lo que se lleva hasta el momento. Cabe mencionar que esta implementación hace uso de la estructura de datos de *conjuntos disjuntos* discutida en el capítulo anterior y presentada en detalle en el apéndice de estructuras de datos. La demostración de corrección se omite.

El tiempo de ejecución del algoritmo de Kruskal está acotado por el tiempo de la ordenación

que es $O(E\log(E))$, donde E es el número de aristas de G, sin embargo como $E < |V(G)|^2$, se tiene que $\log(E) \in O(\log(V))$, por lo que el tiempo total del algoritmo es $O(E\log(V))$. El pseudocódigo y este argumento de la complejidad son tomados de [8, capítulo 23, sección 23.2].

3.2.1. La técnica *greedy* en semiflujo

Como ya se mencionó, este problema ha sido estudiado por mucho tiempo, tanto desde el punto de vista matemático y algorítmico, pero también desde el punto de vista de implementación. En [24, capítulo 6] Robert Tarjan hace un capítulo dedicado a árboles generadores de peso mínimo y presenta varios algoritmos (entre ellos Kruskal y Prim). En las notas finales del capítulo hace mención a un algoritmo en línea (*online*) para calcular el árbol generador de peso mínimo. Este modelo de algoritmos en línea funciona de manera muy similar al modelo de semiflujo, por lo que se puede transcribir fácilmente el algoritmo descrito por Tarjan para que funcione en este modelo.

La idea del algoritmo sigue siendo *greedy*, es decir, cuando se procesa una arista del flujo se debe decidir si se agrega a la estructura construida hasta el momento o si se rechaza y ya no se vuelve a considerar nunca, por supuesto solo se dará una pasada al flujo.

La primera observación es que en el modelo de semiflujo, no se pueden ordenar las aristas, pues esto implica guardarlas y ya se sabe que eso no se permite por las restricciones de memoria.

Si no se pueden ordenar los elementos, entonces se procesarán conforme se vayan viendo y aquí se debe tomar la decisión pertinente, esto deja ver, que agregar una arista podría hacer que sea necesario sacar otra, por lo que la estructura construida irá cambiando a lo largo de la ejecución mediante inserciones e intercambios.

Como en este algoritmo se necesita que cada arista tenga un peso distinto de cero asociado, entonces el contenido del flujo se modifica de manera que cada elemento sea ahora una terna de números: (u, v, w), donde u, v son números enteros y representan a la arista que une a los vértices u con v en la gráfica y el elemento w es el peso asociado a esa arista.

El algoritmo descrito por Tarjan en [24] y formalizado por McGregor en [20] se presenta en 3.2. La idea de este algoritmo es muy sencilla: se inicia con un bosque vacío H y se van agregando elementos hasta que se forme un ciclo, en ese momento se busca la arista más pesada del ciclo y se remueve. Esta decisión se toma al procesar cada arista y al final del proceso se devuelve lo que esté en H.

A continuación se muestran las demostraciones de corrección del algoritmo 3.2; la complejidad, así como los otros criterios de un algoritmo de semiflujo se discuten más adelante.

Teorema 3.1. *El algoritmo 3.2 devuelve un bosque generador de peso mínimo para una gráfica $G = (V, E)$ con función de peso w cuyas aristas recibe como entrada en el flujo S. Esto lo hace realizando solo una pasada al flujo.*

Demostración. Primero hay que observar que en cada paso del algoritmo lo que se tiene es

Algoritmo 3.2 Bosque generador de peso mínimo en el modelo de semiflujo.

Entrada S es un flujo de aristas con pesos que define a una gráfica no dirigida G.
Salida Una subgráfica H que es un bosque generador de peso mínimo de G.

```
H ← ∅.
for each ((u, v), w) ∈ S do
    H ← H ∪ {(u, v)}.
    if H tiene un ciclo then
        Remover la arista de mayor peso en el ciclo de H.
    end if
end for
return H.
```

un bosque de G pues si se forma un ciclo al procesar una arista e, se forma solo uno y e forma parte de él, por lo que al retirar cualquier arista del ciclo, este se rompe, así que H siempre es un bosque.

Además H es generador por que si no, existiría un vértice v de $V(G)$ tal que todas las aristas incidentes en v no están en H. Pero si se considera la última de estas en ser procesada, se observa que si no está en H, es porque formó un ciclo, pero incide en v, por lo que entonces, había ya un camino que unía a v con su otro extremo y en particular, v sería generado por H. Por lo tanto H genera a todo vértice que tenga una arista incidente a él en G.

Finalmente se debe argumentar que H es de peso mínimo. Para simplificar esta demostración se supondrá que los pesos de las aristas son distintos, aunque se puede iterar el argumento aquí descrito si no lo son.

Para argumentar que H es de peso mínimo se utilizará algo que se conoce como *argumento de intercambio* el cual es muy utilizado para demostrar optimalidad en algoritmos *greedy*. Este argumento toma una estructura óptima y la creada por el algoritmo y va intercambiando elementos hasta que una se transforme en la otra preservando en todo momento el costo de la calculada, así se puede argumentar que la creada es tan buena como la óptima por lo que también lo es.

Suponga que H no es de peso mínimo y sea T un bosque generador de G de peso mínimo, esto implica que $w(T) < w(H)$, donde el peso de una subgráfica es la suma de los pesos de las aristas que los contienen.

Observe que si se ordenan a las aristas de T y de H por pesos, como $w(T) < w(H)$, entonces existe una arista $e = (u, v) \in (T)$ que no está en H. Pero cuando se procesa a e en el flujo, esta se agregó a H, entonces en algún momento se retiró (pudo ser incluso en esa misma iteración), pero si se retiró es porque formaba parte de un ciclo, sea C_H el ciclo al que pertenecía e al momento de ser eliminada.

Observe también, que si se elimina a e de T, el árbol al que pertenece e en T se desconecta y forma 2 componentes conexas: C_1 y C_2. Estas cumplen que $u \in C_1$ y $v \in C_2$. Además, todos los vértices de C_H pertenecen a C_1 o a C_2 pues en C_H ya están en la misma componente conexa, así que en T también deben pertenecer a la misma componente, de lo contrario se podrían juntar dos componentes de T. En el ciclo C_H se puede llegar de u a v sin pasar por

e, pero como un extremo está en C_1 y el otro en C_2 entonces debe haber una arista distinta de e que también tenga un extremo en C_1, el otro en C_2, observe que esta arista, llamada $\tilde{e}$ no puede estar en T pues de o contrario en $T \setminus \{e\}$, esta arista seguiría conectando a C_1 con C_2, lo cual no es posible pues T es un bosque, así que en C_H se tiene una arista que no está en T, que no es e, que tiene menor peso que e (pues e es la de mayor peso en C_H) y que hace que $(T \setminus \{e\}) \cup \{\tilde{e}\}$ sea:

1. Un bosque ya que no se ha agregado ningún ciclo
2. Generador pues reconecta a C_1 con C_2 formando de nuevo el mismo conjunto generado por el árbol donde se hallaba e
3. De un peso menor a T pues $w(\tilde{e}) < w(e)$.

El último punto contradice la elección de T por lo que H debe de ser de peso mínimo. □

Complejidad y Criterios de Evaluación

En estos párrafos se discute la complejidad, así como los otros criterios con los que se evalúa un algoritmo de semiflujo.

Complejidad

La complejidad del algoritmo depende de qué tan eficiente sea corroborar si existe un ciclo en $H \cup \{(u, v)\}$ y además depende también de cuánto cuesta encontrar el elemento de mayor peso en el ciclo. Observe que el ciclo formado puede ser del orden de $O(|H|)$ y dado que H es un bosque, se tiene que $|H| \in O(|V|)$. Esto quiere decir que determinar si se forma un ciclo y después remover la arista más pesada de ese, requeriría tiempo proporcional al número de vértices en cada iteración. Por lo que en el peor caso tomaría: $O(|V||E|)$ que es un orden cúbico con respecto al número de vértices. Sin embargo, gracias al amplio estudio de las gráficas en la computación y al desarrollo de estructuras de datos de árboles y conjuntos esto se puede hace mucho más eficiente.

En primer lugar, comprobar si se forma un ciclo se reduce a saber si ambos vértices, de la arista procesada, están en la misma componente conexa. Esto se puede saber si se almacenan los vértices en una estructura de conjuntos disjuntos, al igual que en el caso del modelo convencional en Kruskal. Con esta estructura cada pregunta toma tiempo logarítmico en el peor caso (y mucho menor tiempo amortizado).

El hecho de guardar de manera eficiente los vértices de H logra que la primera parte se optimice, sin embargo, aún faltaría hallar la arista más pesada del ciclo y de nuevo, el ciclo podría ser muy grande (esencialmente todas las aristas de H).

En [24, capítulo 5] se presenta una estructura de datos conocida como *link-cut tree* o árbol de corte y enlazado, en español. Esta estructura de datos permite, realizando unos cambios propuestos por el mismo autor, determinar cuál es la arista más pesada de un ciclo formado por agregar una arista a un árbol en tiempo $O(\log(n))$ con respecto al tamaño del árbol. Esta estructura no es diseñada en sí para representar los árboles de teoría de gráficas, sino árboles vistos como estructuras de datos. De hecho, en el capítulo, Tarjan los utiliza para almacenar

enteros. Sin embargo, haciendo uso implícito de que una gráfica árbol es una estructura de datos árbol, se puede utilizar esta técnica para hallar lo deseado mucho más rápido.

A pesar de que la estructura presentada por Tarjan, hace más eficiente el algoritmo, la implementación sigue siendo complicada, pues primero se debe lidiar con una estructura poco utilizada en la literatura clásica de algoritmos y después se deben realizar los cambios y extensiones que propone el autor tanto en el capítulo 5, como en las notas del capítulo 6, que es donde presenta el algoritmo en línea. Por lo anterior, la implementación se deja fuera del trabajo y se invita al lector interesado a revisar las referencias correspondientes. Aquí solo se discute la complejidad total con estas optimizaciones.

Con ambas optimizaciones, tanto responder si se ha formado un ciclo como eliminar a la arista más pesada de él pueden hacerse, en el peor caso, en tiempo $O(\log(n))$, donde $n = |V|$. Con esto el algoritmo tiene una complejidad total de $O(E \log(V))$.

Tiempo de procesamiento por arista
Una vez discutida la complejidad es fácil ver que el tiempo de procesamiento por arista es de $O(\log(n))$, donde n es el tamaño de la gráfica, recuerde que el tamaño del flujo puede ser de orden $O(n^2)$.

Espacio adicional
El espacio adicional utilizado por el algoritmo es proporcional al necesario para almacenar las estructuras de datos descritas en la parte de complejidad. Ambas estructuras pueden almacenarse utilizando $O(n \log(n))$ palabras en memoria, aunque se debe recalcar que la implementación de la estructura de árboles de corte y enlazado es bastante compleja, por lo que se invita al lector interesado a consultar [24, capítulo 5].

Con esto concluye la discusión acerca del algoritmo de semiflujo. En la siguiente sección se hablará de una generalización de este algoritmo.

3.3. Matroides y Algoritmos *Greedy*

En esta sección se trabajará con un concepto combinatorio llamado *matroide*. Este concepto se le atribuye al matemático Hassley Whitney, quien al estar trabajando en propiedades y generalizaciones de independencia lineal en álgebra, presentó estos objetos combinatorios.

Al inicio los matroides eran un objeto puramente matemático, sin embargo al estudiar sus propiedades y relaciones se encontró que se podía construir un esquema *greedy* que funcionara en cualquier matroide, en particular Jack Edmonds lo presenta en [10]. Esto implica que si un problema se puede codificar mediante un matroide, entonces una estrategia *greedy* para resolver el problema siempre será óptima pues trabaja sobre un matroide, por ejemplo el problema de encontrar un bosque generador de peso mínimo.

Con la idea de que una estrategia *greedy* siempre funciona en un matroide, se generaliza el algoritmo 3.2 a modo de plantearlo puramente en términos de matroides y usando la teoría

de estos objetos se argumentara que esa estrategia *greedy* siempre era óptima en un matroide que se recibe en un flujo.

En la sección se presenta primero la teoría y definiciones que involucran a este concepto matemático. Después, se presenta el algoritmo en el modelo de semiflujo para matroides que generaliza la estrategia *greedy* del algoritmo para bosques generadores de peso mínimo y finalmente, se demuestra que efectivamente esta generalización funciona.

3.3.1. Matroides.

Definición.

Definición 3.2 (Matroide)**.** Un *matroide* es un par ordenado (M, S) que cumple las siguientes propiedades:

1. S es un conjunto finito, no vacío.
2. I es una familia no vacía de subconjuntos de S, llamado el conjunto de **independientes** de S, de modo que si $B \in I$ y $A \subseteq B$, entonces $A \in I$. En este caso se dice que I es **hereditario**. Observe que el conjunto vacío siempre está en I.
3. Si $A \in I$, $B \in I$ y $|A| < |B|$, entonces existe $x \in B \setminus A$ tal que: $A \cup \{x\} \in I$. Se dice que M satisface la **propiedad de intercambio**.

Todo subconjunto de S que no está en I se denomina *dependiente*.

Como ejemplo de un matroide concreto se presenta el llamado *matroide gráfico*:
Proposición 3.3. *Sea $G = (V, E)$ una gráfica simple con al menos una arista y sea $I = \{B \subseteq E|\ B$ no tiene ciclos $\}$. Entonces $M_G = (E, I)$ es un matroide.*

La demostración de que M_G es un matroide es sencilla pues la propiedad 1 se cumple ya que el conjunto E de aristas es finito y no vacío. Cumple la propiedad de ser hereditario pues si un conjunto B de aristas no tiene ciclos, entonces todo subconjunto de B tampoco, de lo contrario B tendría un ciclo.

Además si se tienen dos bosques $B_1 \neq B_2$, tales que $|B_1| < |B_2|$, entonces observe que B_2 tiene menos componentes conexas que B_1 y por el principio de las casillas o pichoneras, debe haber una componente conexa C de B_2 tal que C tiene vértices de dos o más componentes de B_1. Entonces toda arista de C que tiene a uno de sus extremos en una componente de B_1 y al otro en otra se puede agregar a B_1 sin formar un ciclo, por lo que cumple la propiedad de intercambio.

Bases y Circuitos

Dado un matroide $M = (S, I)$, un subconjunto $A \in I$ y un elemento $x \notin A$. Se dice que x es una *extensión* de A si $A \cup \{x\} \in I$, es decir, x es una extensión de A si al agregar a x a A se preserva la independencia.

Definición 3.4 (Base)**.** Sea $M = (S, I)$ un matroide, sea $A \in I$. A es una *base* de M si A no tiene extensiones, es decir, si A no está contenido en un conjunto independiente más grande.

Es útil poder comparar a la bases de un matroide, por lo que se tiene el siguiente teorema:

Teorema 3.5. *Todas las bases de un matroide $M = (S, I)$ tienen el mismo tamaño.*

Demostración. Por contradicción: Sean A y B bases de M tales que $|A| > |B|$. Como ambos conjuntos son bases, entonces son independientes y además $|A| > |B|$, así que se puede usar la propiedad de intercambio. Esta dice que: $\exists x \in A \setminus B$ tal que $B \cup \{x\} \in I$, lo cual implica que B no es un conjunto independiente maximal. Esto contradice la propiedad de que B es base.

Por lo tanto: $|A| = |B|$. □

También es conveniente discutir la noción de un conjunto dependiente minimal, es decir, el concepto dual al de una base:

Definición 3.6 (Circuito)**.** Sea $M = (S, I)$ un matroide, un *circuito* C de M es un conjunto dependiente minimal, es decir: $\forall X \subset C$, $X \in I$. Si $|C| = 1$, se dice que ese elemento es un lazo, si $|C| = 2$, esos elementos son elementos paralelos.

Por simplicidad se consideran matroides sin lazos ni elementos paralelos, por lo que todo circuito tiene tamaño al menos 3.

A continuación se presentan un par de resultados para circuitos en matroides que serán importantes y útiles en las demostraciones de corrección del algoritmo de flujo de datos.

Teorema 3.7 (Harary,1994)**.** *Sean C_1, C_2 circuitos de un matroide $M = (S, I)$, tales que $C_1 \neq C_2$ y $x \in C_1 \cap C_2$, entonces $\exists C$ un circuito en M, tal que $C \subseteq (C_1 \cup C_2) \setminus \{x\}$.*

Demostración. Por contradicción: Suponga que $(C_1 \cup C_2) \setminus \{x\}$ es independiente. Como $C_1 \neq C_2$, entonces $C_1 \setminus C_2 \neq \emptyset$. Sea $x \in C_1 \setminus C_2$. Por definición de circuito: $C_1 \setminus \{x\} \in I$, entonces extienda este conjunto a un conjunto independiente máximo en $C_1 \cup C_2$. Sea X ese conjunto. Como se extiende a partir de $C_1 \setminus \{x\}$, entonces $C_1 \nsubseteq X$, además tampoco contiene a C_2, pues C_2 es un circuito y $X \in I$. Entonces $|X| < |(C_1 \cup C_2) \setminus \{x\}|$, lo cual contradice la elección de X pues es el máximo independiente.

Por lo tanto $(C_1 \cup C_2) \setminus \{x\}$ es dependiente y sea C el conjunto dependiente mínimo por contención en $(C_1 \cup C_2) \setminus \{x\}$. C es el circuito buscado. □

Proposición 3.8. *Sean $M = (S, I)$ un matroide, A un conjunto independiente de M y si existe $x \in S$ tal que $B = A \cup \{x\} \notin I$, entonces B tiene un único circuito.*

Demostración. Por contradicción: Sean C_1, C_2, circuitos de B, tales que $C_1 \neq C_2$. Entonces observe que $x \in C_1$, pues si no, $C_1 \subseteq A$ pero $A \in I$. Análogamente, $x \in C_2$, entonces por el teorema 3.7: $(C_1 \cup C_2) \setminus \{x\}$ tiene un circuito C. Pero $C \subseteq A$. Contradiciendo que A es independiente.

Por lo tanto B tiene solo un circuito. □

A continuación se enuncia un lema que relaciona a los circuitos con las bases y que permite intercambiar elementos entre un circuito y una base.

Lema 3.9. *Sea $M = (S, I)$ un matroide. Sean C un circuito de M y B una base de M, tales que $C \cap B \neq \emptyset$, entonces $\forall e \in B \cap C, \exists x \in (C \setminus (B \cap C))$, tal que $(B \setminus \{e\}) \cup \{x\}$ es base de M.*

Demostración. Por contradicción. Suponga que $\exists e \in B \cap C$, tal que $\forall x \in (C \setminus (B \cap C))$ se tiene que $(B \setminus \{e\}) \cup \{x\}$ no es base de M.

Sea $C = \{x_1, x_2, \ldots, x_k, e_1, \ldots, e_j\}$, donde $\{e_1, \ldots, e_j\} = B \cap C$. Como para toda x_i, $(B \setminus \{e\}) \cup \{x_i\}$ no es independiente (pues no es base), pero $B \setminus \{e\}$ sí por herencia, entonces por la proposición 3.8, hay un único circuito en $(B \setminus \{e\}) \cup \{x_i\}$. Sea C_{x_i} ese circuito, para toda x_i.

Observe que si $x_i \neq x_j$, entonces $C_{x_i} \neq C_{x_j}$, pues C_{x_i} tiene puros elementos de B y a x_i y C_{x_j} también tiene puros elementos de B y a x_j, por lo que son circuitos distintos. Además $\forall i, C_{x_i} \neq C$, pues este último tiene a e.

A continuación se construirá una sucesión de circuitos distintos con la cual eventualmente se llegará a una contradicción. Considere a C y a C_{x_1}, como son distintos y $x_1 \in C \cap C_{x_1}$, entonces por el teorema de Harary (3.7), $(C \cup C_{x_1}) \setminus \{x_1\}$ tiene un circuito. Sea C_1 este circuito. Este circuito debe tener elementos de $C \setminus (C \cap B)$, o de lo contrario, tendría puros elementos de $C_{x_1} \setminus \{x_1\}$, o sea, puros elementos de B, lo cual es una contradicción. Entonces sea x_i un elemento de $C \setminus (C \cap B)$ en C_1. Si considera al C_{x_i} correspondiente, entonces se puede usar el teorema de nuevo ya que comparten en particular a x_i. Teniendo así un circuito $C_2 \subseteq ((C_1 \cup C_{x_i}) \setminus \{x_i\})$. Este proceso se puede seguir, pero en cada paso se va eliminando al menos un elemento de x_i de C. Por lo que en algún momento se habrán eliminado todos (en a lo más k iteraciones). Entonces se seguiría teniendo un circuito y ya no habría elementos x_i, por lo que habría puros elementos de B. Lo cual es una contradicción pues B es base.

Por lo tanto, dada una arista $e \in B \cap C$, siempre existe un elemento $x \in (C \setminus (B \cap C))$, tal que $(B \setminus \{e\}) \cup \{x\}$ es una base para M. □

Restricciones

A veces es conveniente pensar en solo una parte de un matroide en lugar de todo. El concepto de *restricción* formaliza esta noción intuitiva de "parte".

Definición 3.10 (Restricción de un matroide)**.** Sea $M = (S, I)$ un matroide. Sea $A \subseteq S$, $A \neq \emptyset$. La *restricción de M bajo A* se denota por $M_{|A}$ y es un matroide definido de la siguiente manera: $M_{|A} = (A, I_{|A})$, donde $I_{|A} := \{X \in I | X \subseteq A\}$.

La siguiente proposición establece cómo se comportan las bases con respecto a una restricción.

Proposición 3.11 (Extensión de bases bajo restricciones)**.** *Sea $M = (S, I)$ un matroide. Sea $A \subseteq S$, y $M_{|A}$ la restricción de M bajo A. Sea B una base para $M_{|A}$. Entonces $\forall X \supseteq A$, considere la restricción de M bajo X, entonces existe en $M_{|X}$ una base $\tilde{B}$ tal que $B \subseteq \tilde{B}$.*

Demostración. Observe que como B es base en $M_{|A}$, entonces es independiente con respecto a los elementos de A, pero también lo es con respecto a los elementos de X pues no se ha agregado ningún elemento a B, entonces sea $\tilde{B}$ el máximo independiente en $M_{|X}$ tal que contiene a B. Es evidente que $\tilde{B}$ es base en $M_{|X}$ y contiene a B. □

Matroides con Pesos

A un matroide se le puede añadir una *función de pesos*, como a una gráfica y la intención es la misma: añadir un costo o peso a cada elemento para considerar subconjuntos de elementos de menor o mayor costo. Por ejemplo, una base de peso mínimo o un circuito de peso máximo, etc. A continuación se presentan las definiciones pertinentes a extender a un matroide con una función de pesos.

Definición 3.12 (Matroide con pesos)**.** Un *matroide con pesos* $M = (S, I, w)$ es una terna que consta de un conjunto finito no vacío S, una familia I de subconjuntos de S y una función $w : S \mapsto \mathbb{R}^+$ llamada función de peso, tales que (S, I) forman un matroide.

Definición 3.13 (Peso de un conjunto de elementos)**.** Sea $M = (S, I)$ un matroide con una función de pesos w. Sea $X \subseteq S$, el *peso* de X se define como $w(X) = \sum_{e \in X} w(e)$.

Con estas definiciones y teoremas ya se puede plantear el algoritmo.

3.3.2. Algoritmos *Greedy* en Matroides

Si se considera un matroide con pesos, se puede pensar en considerar una base de peso mínimo y con esto surge la idea de crear algoritmos que encuentren estas bases, en particular se puede considerar idear un algoritmo de flujos que reciba como entrada los elementos de un matroide M con pesos w y que encuentra una base de peso mínimo para ese matroide.

Dado un matroide $M = (S, I)$ con función de pesos $w :: S \mapsto \mathbb{R} \setminus \{0\}$, el algoritmo recibe como entrada un flujo que consta de parejas $(e, w(e))$, donde e es un elemento de S y $w(e)$ es el peso asociado a e bajo la función w. El algoritmo procesa cada pareja del flujo y al final encuentra una base que es de menor peso (óptima) para el matroide M.

La idea del algoritmo es generalizar el algoritmo 3.2 que encuentra un bosque generador de peso mínimo para una gráfica con pesos que recibe en el flujo. El algoritmo para matroides toma una a una los elementos del matroide con sus pesos del flujo y los va agregando a un conjunto H que inicialmente está vacío. En cada paso verifica si H es independiente o no (forma parte de I), en caso afirmativo procede con el siguiente elemento, pero en caso negativo entonces debe remover un elemento de H.

Si se debe remover un elemento de H porque ya no es independiente, entonces se busca el elemento más pesado del circuito formado en H y se remueve. Observe que esto tiene sentido gracias a la proposición 3.8 que dice que si H deja de ser independiente, entonces tiene un

único circuito, por lo que se puede hallar ese circuito C y extraer su elemento más pesado. El pseudocódigo se presenta en 3.3.

Algoritmo 3.3 Algoritmo de flujos *greedy* para hallar una base de menor peso en un matroide M.

Entrada Un flujo S que consta de parejas $(e, w(e))$, donde e es un elemento de un matroide $M = (S, I)$ con función de pesos w y $w(e)$ es el peso asociado a e bajo w.
Salida Una base de menor peso con respecto a w para M.

```
H ← ∅.
for (e, w(e)) ∈ S do
    H ← H ∪ {e}.
    if H ∉ I then
        C ← el circuito hallado en H.
        Remover de H el elemento más pesado en C.
    end if
end for
return H.
```

A continuación se presenta la corrección del algoritmo mediante dos teoremas. El primero establece que el conjunto H, al final de la ejecución, es una base para M y el segundo que efectivamente, es una base de peso mínimo.

Teorema 3.14. *H es una base de M al final de la ejecución del algoritmo 3.3.*

Demostración. Por inducción sobre $\tilde{S}$, donde $\tilde{S}$ son los elementos del matroide vistos hasta el momento en el flujo.

Caso base: $\tilde{S} = \{x\}$. En este caso, como se consideran que los elementos unitarios son independientes, y H al inicio es vacío, entonces $H = \{x\}$ y si se considera la restricción de M bajo $\{x\}$, claramente H es base en esa restricción.

H.I. Dados los primeros k elementos de S, $\tilde{S} = \{e_1, \ldots, e_k\}$, entonces $\tilde{H}$ es una base para $M_{|\tilde{S}}$. P.I. Considere el elemento $k+1$ del flujo. Sea $M_{|\tilde{S}} = \tilde{S} \cup \{e_{k+1}\}$ y sea $M_{|S'}$ la restricción correspondiente. De acuerdo al algoritmo, al considerar a $H = \tilde{H} \cup e_{k+1}$ hay dos casos:

1. $\tilde{H} \cup e_{k+1} \in I$. En este caso hay un conjunto independiente de tamaño $|\tilde{H}| + 1$, por lo que toda base tiene al menos ese tamaño, pero como solo se agrega un elemento al conjunto de restricción, entonces toda base en $M_{|S'}$ puede tener a lo más un elemento más que $\tilde{H}$, de lo contrario, se contradiría la hipótesis de inducción. Por lo tanto toda base tiene tamaño a lo más $|\tilde{H}| + 1$ y como todas las bases tienen el mismo tamaño por el teorema 3.5, entonces H en base para $M_{|S'}$.

2. $\tilde{H} \cup e_{k+1} \notin I$. En este caso se argumentará que toda base en $M_{|S'}$ tiene el mismo tamaño que $\tilde{H}$.

   Claramente, por la proposición 3.8, al quitar un elemento de H, lo que queda es independiente, pues se destruye el único circuito formado, solo falta ver que tiene el tamaño de una base en $M_{|S'}$. Sea $n = |H| = |\tilde{H}|$

Suponga que las bases en $M_{|S'}$ tienen tamaño al menos $n+1$. Observe que por la proposición 3.11, como $\tilde{H}$ es base para $M_{|\tilde{S}}$, entonces para todo súperconjunto de $\tilde{S}$, existe una base que la contiene, en particular si el súperconjunto es S', existe una base B para $M_{|S'}$, tal que $\tilde{H} \subseteq B$, pero por la suposición del tamaño de B con respecto a H, se tiene que $\tilde{H} \subset B$.

Observe ahora que B no puede tener puros elementos de $\tilde{S}$ pues se contradiría la hipótesis de inducción, esto implica que $e \in B$, pero como solo se agregó un nuevo elemento y B contiene a $\tilde{H}$, entonces $\tilde{H} \cup \{e_{k+1}\} \subseteq B$, pero esto no es independiente, por lo tanto B no es base, contradiciendo la proposición. La contradicción surge de suponer que las bases de $M_{|S'}$ tienen mayor tamaño que H, por lo tanto, H es independiente y tiene tamaño máximo, por lo que es base.

□

Ahora se demostrará que H es de peso mínimo.

Teorema 3.15. *Al final de la ejecución del algoritmo 3.3, H es una base de peso mínimo de M.*

Demostración. Por contradicción: Suponga que H no es de peso mínimo. Sea T una base de peso mínimo para M tal que $w(T) < w(H)$ y $H \neq T$. Ordene a los elementos de T y de H por peso, sea e el primer elemento de T que no aparece en H. Cuando se procesó a e en el flujo, se añadió a H. Si e no está en H es porque se eliminó en alguna iteración posterior (o en la misma), pero si se eliminó es porque estaba en un ciclo C_e y era el elemento más pesado de ese ciclo. Entonces si se considera al ciclo C_e y a la base T, por el lema 3.9, existe un elemento x de C_e que no está en T, tal que $T_1 = (T \setminus \{e\}) \cup \{x\}$ es una base para M, pero $w(T_1) = w((T \setminus \{e\}) \cup \{x\}) = w((T \setminus \{e\}) + w(x) = w(T) - w(e) + w(x)$. Pero observe que el peso de x es menor al peso de e pues en C_e, e es la de mayor peso, así que: $w(x) - w(e) < 0$ Por lo tanto:

$$w(T_1) = w(T) - c$$

Lo cual es una contradicción en la optimalidad de T.

Por lo anterior H debe ser una base de peso mínimo de M. □

Con lo anterior se establece que el algoritmo propuesto, efectivamente entrega una base óptima (en este caso, de peso mínimo) para cualquier matroide con pesos, cuyos elementos y pesos se pasen como entrada en forma de flujo. Sin embargo, observe que el tiempo de procesamiento por arista, así como la complejidad del algoritmo y el espacio de almacenamiento están sujetos a cómo se representen los matroides y qué tan eficiente sea determinar si un conjunto es independiente o no y después, hallar el circuito subyacente para eliminar el elemento más pesado.

En gráficas se utiliza la estructura propuesta por Tarjan pero esta hace uso de la fuerte relación que existe entre una gráfica abstracta y su representación visual, en particular, en la idea de que un árbol generador, visto como una base para el matroide gráfico subyacente, se puede representar visualmente como una estructura de datos *árbol* donde los nodos son

los extremos del elemento del matroide y el elemento en sí (la arista) se puede pensar que conecta dos nodos.

La idea anterior no se puede generalizar para los matroides, de hecho, pudiera haber matroides cuyas representaciones ni siquiera pudieran ser visualizadas claramente. Así mismo, se debe considerar qué tan eficiente es, en general, saber si se ha formado un circuito o no y de acuerdo a la representación del matroide, se debe hallarlo. Estos procesos son pensados de manera abstracta y no se puede establecer una serie de estructuras de datos o algoritmos concretos que los generalicen para cualquier matroide.

Por los motivos anteriores, la discusión acerca de la eficiencia del algoritmo de flujos 3.3 queda abierta y sujeta a representaciones particulares de los matroides que se reciban como entrada. En un futuro se puede clasificar este algoritmo de acuerdo a ciertas familias de matroides que puedan poseer representaciones similares y por lo tanto, tiempos de procesamiento en el mismo orden.

Por lo pronto, este trabajo solo se limitó a generalizar el concepto de estrategia *greedy* en flujos cuyos elementos forman un matroide con pesos tal como lo hace [8, capítulo 16,sección 16.4] de modo que se tenga un algoritmo general que en el modelo de flujo de datos obtenga bases de peso mínimo (o máximo) para cualquier matroide.

4. Emparejamiento Máximo

En capítulos anteriores se mostraron algoritmos de flujo que hallaban una solución exacta o completa a un problema dado, por ejemplo, deciden si una gráfica es bipartita o no o encuentran un árbol generador de peso mínimo. En este capítulo se presenta un algoritmo de flujo que no encuentra una solución exacta, sino una aproximación a la mejor solución posible del problema.

La técnica de crear algoritmos de aproximación, en lugar de obtener soluciones exactas es muy empleada en el diseño de algoritmos de flujos ya que por las limitaciones de memoria, tiempo de procesamiento por elemento o número de pasadas, a veces es imposible encontrar una solución exacta u óptima. La técnica se basa en empezar con una solución parcial, simple de obtener, y a partir de ella, realizar más pasadas al flujo para ir mejorando la solución hasta cierto punto. En ese punto se tendrá una aproximación razonablemente buena para la solución óptima.

Un *emparejamiento* M de una gráfica $G = (U, V)$ es un conjunto de aristas tales que ninguna comparte un vértice con otra del emparejamiento, formalmente:

Definición 4.1 (Emparejamiento de Aristas de una Gráfica)**.** Sea $G = (V, E)$ una gráfica simple, sea $M \subseteq E$, M es un *emparejamiento de las aristas* de G o simplemente un *emparejamiento* de G si: $\forall v \in V(G)$ a lo más una arista de M incide (tiene como uno de sus extremos) a v.

El problema que se trata en este capítulo es el de hallar un emparejamiento (*matching*) con la mayor cantidad de aristas posibles en una gráfica bipartita, a este tipo de emparejamientos se les llama *emparejamientos máximos*. En la figura 4.1 se encuentra una gráfica G con unas aristas más gruesas que indican que estas forman un emparejamiento máximo para esa gráfica.

Hay varios algoritmos convencionales que encuentran emparejamientos máximos, sin embargo, en el modelo de semiflujo, solo se ha podido aproximar una solución.

El capítulo comienza con la consideración de un algoritmo de aproximación y expone algunos algoritmos convencionales para hallar una solución al problema del emparejamiento máximo en gráficas bipartitas, también se describe porqué estos algoritmos no son viables en el modelo de semiflujo y continúa con la exposición del algoritmo de aproximación a utilizar. Finalmente, se demuestran las propiedades y garantías de este algoritmo.

4.1. Descripción del Problema y Algoritmos Convencionales

El problema del emparejamiento máximo en gráficas bipartitas es muy conocido y existen varios algoritmos convencionales que lo resuelven. El problema se ve como un problema de

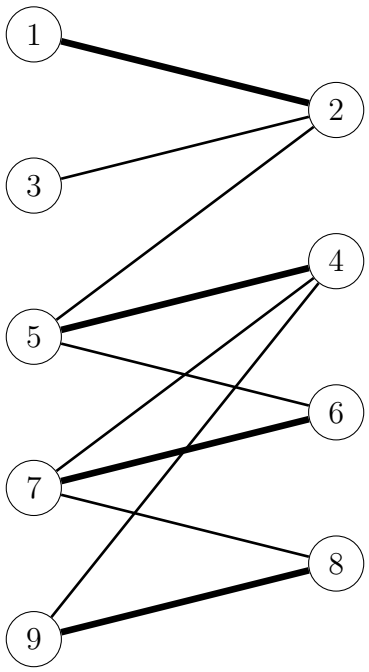


Figura 4.1: En esta figura se observa una gráfica G cuyas aristas marcadas forman parte de un emparejamiento máximo.

optimización, en donde se quiere maximizar la cardinalidad (número de aristas) que contiene un emparejamiento.

Un algoritmo que resuelve el problema puede ser el algoritmo *Ford-Fulkerson* para hallar el flujo máximo en una gráfica, este algoritmo sirve para un problema de flujos pero el problema del emparejamiento máximo se puede reducir muy fácilmente a uno de flujos en la gráfica. Si la entrada es una gráfica bipartita $G = (L \cup R, E)$, se puede construir una gráfica dirigida $E' = (L \cup R \cup \{s, t\}, E')$, donde

$$E' = \{(u, v) | (u, v) \in E\} \cup \{(s, v), v \in L\} \cup \{(v, t), v \in R\}.$$

G' es una gráfica dirigida, donde hay flechas que van de un vértice especial s, llamado *fuente*, hacia toda la parte izquierda (L) de la gráfica, y ahí se dirigen todas las aristas ya existentes en G hacia la parte derecha (R) y a estas se le agrega a cada vértice una flecha hacia otro vértice especial t, llamado *destino*. De este modo G' es una red de flujos donde todas las flechas se ven como tuberías que envían un fluido (agua, por ejemplo) y tienen capacidad 1. En esta red se puede ejecutar el algoritmo Ford-Fulkerson para encontrar el flujo máximo de G'.

Esta cantidad de flujo es equivalente al número máximo de flechas entre L y R por las que pasa el fluido, estas flechas deben ser ajenas por vértices pues cada vértice de R puede retransmitir a lo más 1 unidad a t. Con lo anterior se observa que la capacidad de flujo máximo en G' es equivalente al número de aristas más grande posible, ajenas por vértices que se puede hallar entre L y R, que es un emparejamiento máximo. Por lo tanto, al hallar el flujo máximo en G', se habrá hallado un emparejamiento máximo en G. Para más detalle del algoritmo y de la reducción del problema puede consultar [14].

La complejidad de este algoritmo depende del tamaño y orden de la gráfica, si la gráfica tiene n vértices y m aristas entonces Ford-Fulkerson tiene complejidad de $O(nm)$. Hay algoritmos más eficientes como el que se muestra en [22]. Si se consideran estos algoritmos en el contexto del modelo de semiflujo se observa que requieren tener la gráfica en memoria y esto no es posible en el modelo, por ello es que se debe idear otra manera de resolver el problema.

El problema con estos algoritmos convencionales es que requieren conocer toda la estructura de la gráfica y tenerla a disposición en la memoria durante toda su ejecución y esto no está permitido en el modelo, ya que requeriría almacenar todo el flujo. Con lo anterior no queda claro si el algoritmo de Ford-Fulkerson se pueda adaptar al modelo de semiflujo, pues en general requiere calcular subgráficas tan grandes como la gráfica original.

Más allá de no saber o no poder adaptar los algoritmos convencionales al modelo de semiflujo, tampoco se ha podido hallar en este modelo una solución exacta al problema, es decir, un algoritmo de flujos que tome una gráfica bipartita como entrada y devuelva un emparejamiento máximo para ella. Aunque en [11] se demuestra que determinar si dado un emparejamiento, verificar si es máximo en una pasada, requiere $\Omega(m)$ bits, donde m es el orden de la gráfica (el tamaño del flujo).

Sin embargo, es fácil encontrar una solución parcial y se han creado algoritmos que mejoren esa solución parcial poco a poco aproximando el tamaño del emparejamiento hallado al de uno máximo. En particular en [11] se define un algoritmo que encuentra una buena aproximación al problema y es el que se presenta en las siguientes secciones.

4.2. Idea del Algoritmo de Flujos

La solución al problema deberá aproximarse y para ello se considera que el algoritmo vaya construyendo una solución cada vez mejor, de modo que después de cierto número de etapas del algoritmo se tenga una mejor solución que la inicial y esta sea una aproximación a la solución óptima. Para lograr esto, el algoritmo, en cada etapa, incrementará el conjunto solución hallado hasta el momento utilizando un concepto llamado *caminos 3-aumentables*.

Definición 4.2 (Vértice libre)**.** Sea G una gráfica, sea M un emparejamiento sobre G, decimos que $v \in V(G)$ es *libre*, con respecto a un emparejamiento M si v no forma parte de ninguna arista en M.

Definición 4.3 (Camino 3-aumentable)**.** Sea $G = (V, E)$ una gráfica, sea M un emparejamiento sobre G. Un *camino 3-aumentable* para una arista $(u, v) \in M$, es una 4-tupla (x, u, v, y) de vértices de G tales que , $(x, u), (v, y) \in E$ y x, y son libres con respecto a M.

Al hablar de caminos 3-aumentables, de acuerdo a la definición anterior, se considera a los extremos x, y como *las puntas*, siendo x, la punta izquierda, y la derecha. Mientras que a la arista (x, u) se le llama *ala izquierda* y a la arista (v, y) *ala derecha*. La figura 4.2 presenta una situación donde hay un emparejamiento y un camino 3-aumentable en él.

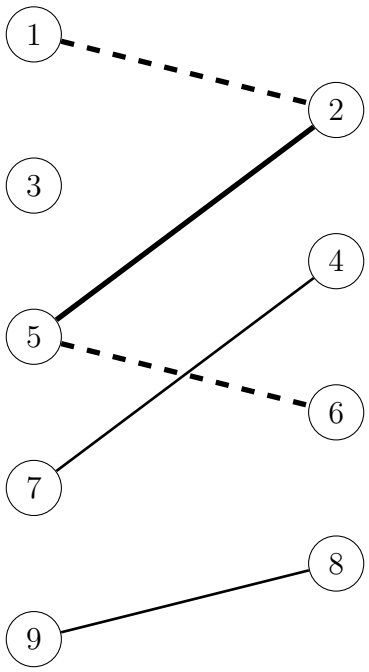


Figura 4.2: En esta figura se observa un emparejamiento de una gráfica G. Las aristas continuas son parte del emparejamiento. La arista más gruesa es una que tiene un camino 3-aumentable, marcado por las aristas punteadas, estas no forman parte del emparejamiento, pero se pueden agregar y se puede remover la arista $(5, 2)$ para aumentar el tamaño de este emparejamiento.

Esta noción de caminos 3-aumentables es básica para mejorar el emparejamiento, ya que, en este caso, se puede retirar del emparejamiento a la arista (u, v) y agregar a las aristas (x, u) y (v, y) pues los extremos x, y estaban libres en el emparejamiento y al remover a (u, v) se liberan también a u y a v. Con esta acción, se ha incrementado el tamaño del emparejamiento en 1 por lo que se ha mejorado, si se repite esta operación se puede seguir mejorando hasta que ya no haya caminos 3-aumentables disponibles.

El algoritmo debe hallar esos caminos y transformar el emparejamiento inicial hasta que ya no queden más, esta será la idea fundamental. Al tratarse de un algoritmo de aproximación, además de la entrada dependerá de un parámetro de aproximación ϵ. Con este factor, el algoritmo encuentra una $(\frac{2}{3}-\epsilon)$-aproximación de un emparejamiento máximo. Recordando las definiciones presentadas en el Capítulo 1, sección 3, esto significa que el algoritmo entrega un emparejamiento que tiene tamaño menos $(\frac{2}{3}-\epsilon)$ del máximo, por lo que si ϵ es muy pequeña, el tamaño será al menos de casi $\frac{2}{3}$ con respecto al máximo posible. Algo que hay que tener en cuenta con estos algoritmos es que pudiera darse el caso de que el emparejamiento entregado sea el máximo, es decir, esta aproximación establece una garantía con el tamaño mínimo que tendrá el emparejamiento entregado, más no establece ninguna garantía del tamaño máximo (evidentemente, no puede ser mayor al máximo, posible).

4.3. Algoritmo en Detalle

En esta sección se presenta formalmente el algoritmo que aproximará la solución de un emparejamiento máximo en una gráfica *bipartita* que recibe como entrada en el flujo. Como la gráfica debe ser bipartita, se sugiere utilizar el algoritmo presentado en el capítulo correspondiente de este trabajo para determinar si la gráfica de entrada en el flujo es bipartita o no y para obtener la bipartición correspondiente.

La idea general es ir mejorando la solución con cada etapa del algoritmo. Una etapa requiere de varias pasadas al flujo, pero este número es constante en cada etapa (siempre es el mismo). Debido a que el número de pasadas pudiera ser muy grande, se requiere de un mecanismo que determine cuando el emparejamiento ya no puede mejorar y detener el algoritmo en ese momento.

Primero se debe obtener una solución base, es decir, un emparejamiento que sea fácil de obtener y sobre el que se trabajará para mejorarlo. La mejor idea para obtener esta solución base es crear un emparejamiento maximal, es decir, uno en donde ya no se puedan agregar aristas. Para ello, naturalmente, se debe procesar el flujo al menos una vez. En 4.1 se presenta un algoritmo que en una ronda (pasada) crea un emparejamiento maximal para cualquier gráfica que reciba como entrada.

Algoritmo 4.1 Algoritmo para obtener un emparejamiento maximal de una gráfica en el modelo de semi-flujo.

Entrada S es un flujo de aristas E de una gráfica $G = (V, E)$.
Salida Un arreglo de tamaño $|V|$ que codifica un emparejamiento maximal.

```
array ← Arreglo de tamaño |V| inicializado en 0.
for e = (u, v) ∈ S do
    if array[u] == 0 ∧ array[v] == 0 then
        array[u] ← v.
        array[v] ← u.
    end if
end for
return array.
```

El resultado del emparejamiento creado en 4.1 está codificado en el arreglo `array`. Este arreglo inicia con todas sus entradas en 0, indicando que todos los vértices están libres, la entrada de un vértice se modifica si se procesa una arista en donde ambos extremos estén libres, en este caso se codifica en el arreglo que el vértice u fue emparejado con el vértice v y viceversa. A continuación se demuestra que el resultado es efectivamente un emparejamiento maximal de G y que cumple con las restricciones de memoria y tiempo de procesamiento del modelo de semiflujo.

Proposición 4.4. *El algoritmo 4.1 entrega un emparejamiento maximal de una gráfica bipartita G con n vértices que recibe como entrada utilizando $O(n \log(n))$ bits de espacio adicional, realizando una pasada al flujo y procesando cada arista en tiempo constante.*

Demostración. El algoritmo entrega un emparejamiento maximal M, porque si se supone lo contrario, habría dos vértices u y v libres en M pero conectados en G, pero si ambos están libres es porque al procesar la arista (u, v) al menos uno de ellos ya no debería estar libre, contradiciendo la elección de u y v.

La memoria utilizada está dada por el arreglo *array* de n posiciones que requiere $O(n \log(n))$ bits de memoria para ser almacenado, en el proceso de cada arista solo se pregunta si dos posiciones en el arreglo son cero y en caso afirmativo se realizan dos asignaciones, por lo que todo toma $O(1)$. Todo lo anterior solo requiere una pasada al flujo y el algoritmo realiza un número total de operaciones de $O(m)$, donde m es el tamaño del flujo, es decir, el número de aristas de G.

□

Para mejorar el algoritmo maximal encontrado se utilizará la idea de hallar caminos 3-aumentables ajenos por vértices y sustituir las aristas de en medio por las alas para mejorar la cardinalidad del emparejamiento. Para esto, se requiere un algoritmo que permita obtener estos caminos ajenos por vértices. Este algoritmo utiliza el mismo flujo y el emparejamiento ya encontrado. Observe que a partir del arreglo de salida del algoritmo 4.1 se puede saber en tiempo constante si un vértice está libre o no.

El algoritmo 4.2 encuentra un conjunto de caminos 3-aumentables ajenos por vértices, pero utiliza un factor extra que permite medir la cantidad de caminos encontrados, si esta es muy pequeña, el algoritmo se detiene y ya no prosigue.

El factor que sirve para medir esto se llama δ. Este argumento es utilizado para determinar qué tanto puede crecer el emparejamiento ya obtenido, si el posible crecimiento es menor a este valor el algoritmo termina y ya no aumenta el emparejamiento encontrado. Debido a que en cada etapa del algoritmo se realizan varias pasadas al flujo se debe estimar si vale la pena seguir realizando estas pasadas o no, en ese sentido este factor δ actúa como un factor de *mejora*.

Este algoritmo realiza tres pasadas al flujo en cada etapa y en cada una de las etapas elimina vértices de tal modo que en las siguientes, toda arista incidente en estos vértices ya no es considerada.

El algoritmo utiliza varios arreglos para almacenar la información necesaria, estos arreglos sirven para almacenar las alas izquierdas y derechas encontradas así como para marcar los vértices ya eliminados.

A continuación se detalla el pseudocódigo para implementar cada uno de los primeros tres pasos del algoritmo 4.2. En cada uno de los primeros tres pasos se requiere dar una pasada al flujo, así que el algoritmo da tantas pasadas como etapas se requieran, considerando que en cada ronda se realizan tres pasadas. El número de etapas está determinado por el factor de mejor δ.

El primer paso del algoritmo pide hallar un conjunto maximal ajeno por vértices de alas izquierdas para el emparejamiento M. Para lograr esto se procesa el flujo y para cada arista se determina si su extremo derecho está en M. En este caso el extremo izquierdo está libre en M y se considera una punta izquierda.

Algoritmo 4.2 Algoritmo para encontrar un conjunto de caminos 3-aumentables ajenos por vértices a partir de un emparejamiento M y un factor de calidad δ en el modelo de semiflujo.

Entrada S es un flujo de aristas E de una gráfica $G = (V, E)$, M un emparejamiento maximal de G codificado en un arreglo $array$, $0 < \delta < 1$ un parámetro de calidad.

Salida Un par de arreglos de vértices codificando a las alas izquierdas y derechas de un camino 3-aumentable de acuerdo al emparejamiento maximal tomado como entrada.

1: En una pasada, encontrar un conjunto maximal ajeno por vértices de alas izquierdas, codificado en un arreglo `left_wings`. Si el tamaño del arreglo es menor o igual a δM, terminar y regresar los arreglos `left_wings` y `right_wings`.

2: En una segunda pasada, para las aristas de M con alas izquierdas hallar un conjunto maximal de alas derechas ajenas por vértices y codificarlo en el arreglo `right_wings`.

3: En una tercera pasada se identifican a los vértices que cumplen al menos una de las siguientes condiciones:
   (a) Son los extremos de una arista de M que tiene un ala izquierda.
   (b) Son los extremos de alas de una arista de M que obtuvo ambas alas.
   (c) Son los extremos de una arista de M que ya no es 3-aumentable, es decir, que o no tuvo ala izquierda o que no obtuvo ala derecha.

   Todos estos vértices se marcan en un arreglo `ids` y se omite en pasadas subsecuentes toda arista que incida en un vértice de este conjunto.

4: Repetir desde el paso 1.

Si el extremo derecho de una arista está en M, entonces su posición correspondiente en el arreglo $array$ no es 0. En el algoritmo 4.3 se encuentra el pseudocódigo necesario para realizar este proceso, en él se considera la condición de terminación del algoritmo 4.2, por medio del factor de mejora δ. También se considera omitir aristas que incidan en un vértice del arreglo `ids`.

En el segundo paso se debe encontrar un conjunto maximal de alas derechas ajenas por vértices para aquellas aristas de M que hayan obtenido un ala izquierda en el paso anterior. Esto se verifica viendo la entrada de un vértice en el arreglo $left\_wings$. El conjunto buscado se codifica en el arreglo $right\_wings$ declarado en el paso anterior.

El pseudocódigo se encuentra en 4.4. Primero se verifica que la arista no deba ser omitida, después que el extremo izquierdo esté en M y finalmente que el extremo izquierdo de la arista en M tenga un ala izquierda.

A continuación se demuestra que los algoritmos 4.3 y 4.4 son correctos:

Proposición 4.5. *Los algoritmos 4.3 y 4.4 producen conjuntos ajenos por vértices (disjuntos) y maximales.*

Demostración. Para verificar que son ajenos por vértices basta suponer que no. Que hay dos alas izquierdas (o derechas) que comparten un vértice x, pero en la posición x del arreglo correspondiente solo puede haber uno de estos dos extremos en disputa, por lo que solo el último de estos dos en ser procesado será el emparejado con x.

Algoritmo 4.3 Desglose del primer paso del algoritmo 4.2

```
left_wings ← un arreglo de tamaño |V| iniciado en 0.
right_wings ← un arreglo de tamaño |V| iniciado en 0.
ids ← un arreglo de tamaño |V| iniciado en False.
ammount ← 0.
for e = (u, v) ∈ S do
    if ids[u] == False ∧ ids[v] == False then
        if array[v]! = 0 ∧ array[u] == 0 then
            left_wings[u] ← v.
            left_wings[v] ← u.
            ammount ← ammount + 1.
        end if
    end if
end for
if ammount ≤ δ|M| then
    return (left_wings, right_wings).
end if
```

Algoritmo 4.4 Segundo paso del algoritmo 4.2.

```
for e = (u, v) ∈ S do
    if ids[u] == False ∧ ids[v] == False  then
        if array[u]! = 0 then
            z ← array[u].
            if left_wings[z]! = 0 ∧ array[v] == 0 then
                right_wings[u] ← v.
                right_wings[v] ← u.
            end if
        end if
    end if
end for
```

Para observar que son maximales se puede hacer una prueba muy similar a la realizada para el algoritmo 4.1. □

Finalmente, para el tercer paso del algoritmo 4.2 se debe cambiar el valor del arreglo `ids` de aquellos vértices que cumplan con alguna de las condiciones mencionadas, estas se identifican en cada uno de los 3 ciclos presentes en el algoritmo 4.5.

Una vez que se encontró un conjunto de caminos 3-aumentables ajenos por vértices se puede proceder a reemplazar aristas en el emparejamiento original. Hay que recordar que por cada camino 3-aumentable hallado se puede reemplazar la arista de en medio por ambas alas en el emparejamiento, eso se debe a que estos caminos son ajenos por vértices.

El pseudocódigo mostrado en 4.6 es el algoritmo principal que encuentra una aproximación a un emparejamiento máximo para una gráfica bipartita G. Este algoritmo, recibe, además de la gráfica, un parámetro de aproximación ϵ que está acotado por 0 y 1/3. Este algoritmo

Algoritmo 4.5 Tercer paso del algoritmo 4.2.

for $e = (u, v) \in S$ **do**
  if $left\_wings[u]! = 0 \wedge array[u] == v$ **then**
    $ids[u] \leftarrow True.$
    $ids[v] \leftarrow True.$
  end if
  if $left\_wings[u]! = 0 \wedge array[u] == v \wedge right\_wings[v]! = 0$ **then**
    $x \leftarrow left\_wings[u].$
    $y \leftarrow right\_wings[v].$
    $ids[x] \leftarrow True.$
    $ids[y] \leftarrow True.$
  end if
  if $array[u] == v \wedge (left\_wings[u] == 0 \vee right\_wings[v] == 0)$ **then**
    $ids[u] \leftarrow True.$
    $ids[v] \leftarrow True.$
  end if
end for

primero encuentra un emparejamiento maximal, después realiza varias etapas en donde, en cada una, se ejecuta el algoritmo 4.2 y se intercambian las aristas encontradas. Al concluir las etapas en el arreglo `array` se tendrá la aproximación del emparejamiento máximo

El algoritmo 4.6 va modificando el emparejamiento obtenido hasta el momento en cada ronda y para ello utiliza un arreglo auxiliar, además se considera que, como en el arreglo `array` se codifica dos veces el emparejamiento, entonces solo se procesen las aristas de este una sola vez.

En los párrafos siguientes se argumenta que este algoritmo efectivamente entrega una aproximación de la solución al problema y se establece que cumple con las restricciones del modelo.

4.3.1. Propiedades del Algoritmo de Flujos

Para determinar las propiedades del algoritmo se demostrarán primero unos lemas auxiliares. El primero establece una relación entre el tamaño de un conjunto maximal ajeno por vértices de caminos 3-aumentables con respecto al tamaño de uno máximo. El segundo, cómo influye el agregar las aristas de un conjunto maximal de caminos 3-aumentables ajenos por vértices a un emparejamiento con respecto al tamaño de un emparejamiento máximo.

Después, se prueba un lema que establece qué tan grandes son los conjuntos de caminos 3-aumentables hallados por el algoritmo 4.2 y finalmente, se prueba un teorema que establece las propiedades del algoritmo 4.6.

Lema 4.6 (Relación entre conjuntos 3-aumentables maximales y máximos)**.** *Sea M un emparejamiento para una gráfica G. Sea C un conjunto de caminos 3-aumentables ajenos por*

Algoritmo 4.6 Algoritmo para encontrar una aproximación al problema del emparejamiento máximo en gráficas bipartitas.

Entrada Una gráfica bipartita $G = (L \cup R, E)$ y un parámetro de aproximación $0 < \epsilon < \frac{1}{3}$
Salida Un arreglo que codifica una aproximación de un emparejamiento maximo para G.

Hallar la bipartición de G.
Hallar un emparejamiento maximal de G utulizando el algoritmo 4.1 y codificarlo en un arreglo $array$.
for $k \in \{1, \ldots, \left\lceil \frac{\log 6\epsilon}{log(8/9)} \right\rceil\}$ **do**
Ejecutar el algoritmo 4.2 con argumentos G, $array$ y $\delta = \frac{\epsilon}{2-3\epsilon}$. La salida de este algoritmo se encuentra en los arreglos $left\_wings$ y $right\_wings$.
$array\_aux \leftarrow$ un arreglo de tamaño $|V(G)|$ iniciado en 0.
for $i \in \{1, \ldots, |V(G)|\}$ **do** //Se recorren los vértices de G
if $array[i]! = 0$ **then** //Si está emparejado
$j \leftarrow array[i]$.
if $left\_wings[i]! = 0 \wedge right\_wings[j]! = 0 \wedge i < j$ **then**
$x \leftarrow left\_wings[i]$. //Si hay camino 3-aumentable
$y \leftarrow right\_wings[j]$. //Se saca la arista de enmedio y
$array\_aux[i] \leftarrow x$. //Se meten las alas
$array\_aux[j] \leftarrow y$.
$array\_aux[y] \leftarrow j$.
end if
end if
end for
$array \leftarrow array\_aux$. //Se actualiza el nuevo emparejamiento
end for
return $array$.

vértices maximal para M, sea O un conjunto de caminos 3-aumentables ajenos por vértices máximo para M. Entonces $|C| \geq \frac{|O|}{3}$.

Demostración. Dado un emparejamiento M. Considere que, como C es maximal, entonces por cada camino 3-aumentable (x, u, v, y) en O, C debe de usar al menos uno de los extremos x o y o bien, la arista (u, v), que es del emparejamiento. Por lo tanto, cada camino 3-aumentable en C destruye a lo más 3 caminos en O, así que $|C| \geq \frac{|O|}{3}$. □

Lema 4.7. *Sea X un conjunto ajeno por vértices de caminos 3-aumentables máximo para un emparejamiento maximal M. Sea $\alpha = \frac{|X|}{|M|}$ y sea OPT un emparejamiento máximo. Entonces se cumple que: $|M|(1 + \alpha) \geq \frac{2}{3}|OPT|$.*

Demostración. Considere $M' = OPT \triangle M$, la diferencia simétrica entre OPT y M, es decir:

$$M' = (OPT \cup M) \setminus (OPT \cap M)$$

Proposición 4.8. *En toda componente conexa de M' se tiene a lo más una arista de OPT que de M.*

Demostración. Considere cualquier componente conexa de M'. Las aristas de esta componente forman parte de OPT o de M, más aún, un camino en esta componente se debe ver como una secuencia alternada de aristas de OPT y M pues ambos son emparejamientos maximales. Además como OPT es óptimo, entonces $|OPT| \geq |M|$, así que solo puede haber a lo más una arista más de OPT que de M en la componente ya que si hubiera dos (o más), entonces el camino que las une debería tener dos aristas de OPT seguidas violando la idea de un emparejamiento. □

Considere además que si una componente conexa de M' tiene solo una arista, esta debería ser de M, pues si fuera de OPT entonces no tendría aristas de M como vecinos, en particular los extremos de esa aristas están libres con respecto a M, pero M es maximal, lo que evita que esto pase. En estas componentes la razón de aristas de M con respecto a las aristas de X es 1.

Ahora, considere las componentes de M' que tienen exactamente una arista de M y dos de OPT, como deben estar alternadas, entonces, observe que esta componente es un camino 3-aumentable con respecto a M, pues los extremos de las aristas de OPT están libres con respecto a M y la arista de en medio forma parte de M. Si se considera el número de estas componentes en M', este está acotado por arriba por $|X|$ pues X es un conjunto máximo de estos caminos con respecto a M. Aquí la razón es $\frac{1}{2}$.

Las componentes restantes de M' tienen al menos dos aristas de M, sea n esta cantidad, entonces por la observación, el número de aristas de OPT es a lo más $n+1$, por lo tanto si se considera la razón entre estas dos cantidades, tenemos una razón mínima de la forma $\frac{n}{n+1}$.

Es claro que $|M|$ se puede contar con todas las aristas en M' más las de la intersección. Las primeras se dividen en los tipos de componentes descritos anteriormente y si se compara el tamaño de M con respecto al de OPT se obtiene que en la intersección es igual, pero en las componentes de M':

1. Si hay solo una arista de M, entonces no hay aristas de OPT.
2. Si hay una arista de M y hay más aristas, debe haber solo dos de OPT. Pero este número de componentes está acotado por X.
3. Si hay más de dos aristas de M, entonces hay al menos $\frac{2}{3}$ de aristas de M en toda la componente.

Por todo lo anterior el tamaño de M es siempre mayor o igual a $\frac{2}{3}$ del tamaño de OPT, salvo en las componentes de tipo 2, pero estas son a lo más $|X|$. Entonces:

$$|M| \geq \frac{2}{3}|OPT| - |X| \therefore$$

$$|M| + |X| = |M|(1+\alpha) \geq \frac{2}{3}|OPT|$$

□

Ahora se enuncia un lema que habla acerca del rendimiento del algoritmo 4.2.

Lema 4.9. *El algoritmo 4.2 encuentra* $(\alpha|M|-2\delta|M|)/3$ *caminos 3-aumentables simultáneos en* $\frac{3}{\delta}$ *pasadas al flujo.* $\alpha = \frac{|X|}{|M|}$*, donde* X *es un conjunto máximo de caminos 3-aumentables simultáneos.*

Demostración. Recuerde que el algoritmo utiliza un emparejamiento maximal M y un factor de decisión δ. El algoritmo encuentra un conjunto maximal de caminos 3-aumentables ajenos por vértices.

Sea $L(M)$ el subconjunto de vértices de L que son extremos de una arista en M y sea $V_L(M) = \{v \in R | v \text{ está libre en } M \wedge \exists u \in L(M), (u, v) \in E(G)\}$.

A cada repetición de los pasos $1-4$ del algoritmo se le conocerá como una *fase*. Para que haya una fase completa, en el primer paso se deben hallar al menos $\delta|M|$ alas izquierdas y todos estos vértices se ignoran en las siguientes, por lo tanto el número de aristas ignoradas de M en cada fase es al menos $\delta|M|$, por lo tanto el algoritmo puede tener a lo más $1/\delta$ fases. Además en cada fase se realizan 3 pasadas al flujo, por lo que el número total de pasadas es a lo más 3δ.

Cuando el algoritmo termina, hay a lo más $\delta|M|$ alas izquierdas halladas. El conjunto encontrado en el primer paso, es un emparejamiento maximal entre los vértices no ignorados de $L(M)$ y $V_L(M)$. Como es maximal, entonces hay a lo más $2\delta|M|$ alas izquierdas disponibles en los vértices no ignorados de G. Como son alas, entonces hay a lo más $2\delta|M|$ caminos 3-aumentables ajenos por vértices restantes.

Sea G' la subgráfica de G compuesta por los vértices y aristas no ignorados aún por el algoritmo. Sea $G'' = G \setminus G'$. Observe que un conjunto máximo de caminos 3-aumentables simultáneos en G'' tiene tamaño al menos $|X| - 2\delta|M|$, ya que al máximo posible hay que restarle los que quedan en G'. Además el número de caminos 3-aumentables n hallados hasta el momento es maximal, por lo que por el lema 4.6:

$$n \geq \frac{|X| - 2\delta|M|}{3} = \frac{a|M| - 2\delta|M|}{3}$$

□

Finalmente, en el siguiente teorema se establecen las propiedades del algoritmo 4.6 que determinan la calidad de aproximación del emparejamiento obtenido, así como la eficiencia del algoritmo.

Teorema 4.10. *Sea* $0 < \epsilon < \frac{1}{3}$ *y sea una gráfica bipartita* G*. El algoritmo 4.6 encuentra una* $(\frac{2}{3} - \epsilon)$*-aproximación de un emparejamiento máximo en* G *utilizando* $O(\log(\frac{1/\epsilon}{\epsilon}))$ *pasadas al flujo. El algoritmo procesa cada arista en tiempo constante, salvo en la primera pasada, en donde se utiliza un tiempo de* $O(\log(n))$*. El espacio adicional utilizado por el algoritmo tiene orden* $O(n\log(n))$*.*

Demostración.
Tiempo de procesamiento por arista
Observe que la primera pasada al flujo consiste en hallar la bipartición, pero para ello se puede

utilizar el algoritmo 2.2, el cual realiza este trabajo utilizando un tiempo de procesamiento de $O(\log(n))$ por arista, en el peor caso.

Después de hallar la bipartición se debe encontrar un emparejamiento maximal; utilizando el algoritmo 4.1 se puede lograr esto utilizando tiempo constante para procesar cada arista. Con el emparejamiento maximal se procede a utilizar el algoritmo 4.2 un número dado de etapas, en cada una de estas el algoritmo 4.2 realiza tres pasadas al flujo, pero en cada una de ellas el tiempo de procesamiento por arista es constante. Así en cada pasada que el algoritmo 4.6 realiza, después de la primera, procesa cada arista del flujo en tiempo constante.

Número de pasadas al flujo
Se utiliza una pasada para hallar la bipartición y una más para hallar un emparejamiento maximal.

Después, se realizan $\left\lceil \frac{\log(6\epsilon)}{\log(\frac{8}{9})} \right\rceil$ etapas. En cada una de estas, se ejecuta el algoritmo 4.2 con parámetro $\delta = \frac{\epsilon}{2-3\epsilon}$. En la demostración del lema 4.9, se establece que el algoritmo 4.2 realiza $3/\delta$ pasadas, es decir, $\frac{6-9\epsilon}{\epsilon}$ pasadas. Así que el total de pasadas realizadas es:

$$2 + \left\lceil \frac{\log(6\epsilon)}{\log(\frac{8}{9})} \right\rceil \frac{6-9\epsilon}{\epsilon} \leq 2 + \left(\frac{\log(6\epsilon)}{\log(\frac{8}{9})} \right) \frac{6-9\epsilon}{\epsilon} + \frac{6-9\epsilon}{\epsilon}$$

Sea $c_1 = \frac{1}{\log(\frac{8}{9})}$. Observe que $c_1 < 0$ pues el argumento de la función log es menor a 1. Con esto, la desigualdad derecha queda de la siguiente manera:

$$2 + c_1 \log(6\epsilon) \frac{6-9\epsilon}{\epsilon} + \frac{6-9\epsilon}{\epsilon} = 2 + \frac{6c_1 \log(6\epsilon)}{\epsilon} - \frac{c_1 9\epsilon \log(6\epsilon)}{\epsilon} + \frac{6-9\epsilon}{\epsilon} =$$

$$2 + \frac{6c_1 \log(6\epsilon)}{\epsilon} + -c_1 9 \log(6\epsilon) + \frac{6-9\epsilon}{\epsilon} = 2 + \frac{6 \log(6\epsilon)^{c_1}}{\epsilon} + -c_1 9 \log(6\epsilon) + \frac{6-9\epsilon}{\epsilon} =$$

$$2 + \frac{6 \log(\frac{6}{\epsilon^{-c_1}})}{\epsilon} + -c_1 9 \log(6\epsilon) + \frac{6-9\epsilon}{\epsilon}$$

Considerando el orden de los factores anteriores tenemos: $O(\frac{\log(\frac{1}{\epsilon})}{\epsilon} + \log(\epsilon))$ pero como $\epsilon < 1$ lo anterior tiene orden $O(\frac{\log(\frac{1}{\epsilon})}{\epsilon})$

Espacio de almacenamiento
El algoritmo necesita almacenar varios arreglos extra: para la bipartición, para el emparejamiento maximal y otro para el nuevo emparejamiento (estos últimos se intercambian). Hay que recordar que estos arreglos tienen n entradas, por lo que ocupan $O(n \log(n))$ bits de espacio.

Corrección
Sea OPT un emparejamiento máximo para G. En la ronda i sea M_i el emparejamiento encontrado por el algoritmo y X_i un conjunto máximo de caminos 3-aumentables ajenos por vértices para el emparejamiento M_i. Sea $\alpha_i = \frac{|X_i|}{|M_i|}$ y $s_i = \frac{|M_i|}{|OPT|}$.

Dados los posibles valores de α_i se tienen dos casos:

1. Existe una ronda i en donde $\alpha_i \leq \frac{3\epsilon}{2-3\epsilon}$:
   En este caso se tiene que: $1+\alpha_i \leq 1+\frac{3\epsilon}{2-3\epsilon} = \frac{2}{2-3\epsilon}$, entonces $\frac{1}{1+\alpha_i} \geq \frac{2-3\epsilon}{2}$.

   Por el lema 4.7:

$$|M_i|(1+\alpha_i) \geq \frac{2}{3}|OPT| \Rightarrow$$

$$|M_i| \geq \frac{2}{3}\left(\frac{1}{1+\alpha_i}\right)|OPT| \geq \frac{2}{3}\left(\frac{2-3\epsilon}{2}\right)|OPT| = \left(\frac{2}{3}-\epsilon\right)|OPT|$$

   Dado que el tamaño del emparejamiento encontrado hasta el momento nunca disminuye, entonces el tamaño del emparejamiento de la última ronda es al menos tan grande como M_i, pero $|M_i|$ es una $\left(\frac{2}{3}-\epsilon\right)$-aproximación.

2. $\forall$ ronda i, $\alpha_i > \frac{3\epsilon}{2-3\epsilon}$:
   Recuerde que $\delta = \frac{\epsilon}{2-3\epsilon}$. Observe que $\frac{\alpha_i}{3} > \frac{1}{3}\frac{3\epsilon}{2-3\epsilon} = \frac{\delta}{3}$, por lo que $\delta \leq \frac{\alpha_i}{3}$, $\forall i$. Ahora, con esta δ, se tiene, por el lema 4.9, el algoritmo 4.2 halla $\frac{\alpha_i|M_i|-2\delta|M_i|}{3} = \frac{|M_i|(\alpha_i-2\delta)}{3}$.

   Como $\delta \leq \frac{\alpha_i}{3}$, entonces $2\delta \leq \frac{2}{3}\alpha_i \Rightarrow -2\delta \geq -\frac{2}{3}\alpha_i \Rightarrow \alpha_i - 2\delta \geq \alpha_i - \frac{2}{3}\alpha_i = \frac{\alpha_i}{3}$ $\therefore$

$$\frac{|M_i|(\alpha_i-2\delta)}{3} \geq \frac{|M_i|\alpha_i(\frac{1}{3})}{3} = \frac{|M_i|\alpha_i}{9}$$

   Ahora, como se inicia con un emparejamiento M_0 que es maximal, se tiene que $s_0 = \frac{|M_0|}{|OPT|} > \frac{1}{2}$. Ahora, por el lema 4.7, en cualquier etapa se tiene que $|M_i| + \alpha_i|M_i| \geq \frac{2}{3}|OPT|$. Por lo que

$$s_i + \alpha_i s_i \geq \frac{2}{3} \tag{4.1}$$

   Para calcular el tamaño del siguiente emparejamiento en la etapa $i+1$ se debe agregar al tamaño actual el aumento hallado, pero el lema 4.9 nos garantiza el tamaño de ese aumento, por lo que se tiene

$$|M_{i+1}| = |M_i| + |M_i|\frac{(\alpha_i-2\delta)}{3} \geq |M_i| + |M_i|\frac{\alpha_i}{9} = |M_i|(1+\frac{\alpha_i}{9})$$

   Por lo tanto, $s_{i+1} = \frac{|M_{i+1}|}{|OPT|} \geq \frac{|M_i|}{|OPT|}\left(1+\frac{\alpha_i}{9}\right) = s_i\left(1+\frac{\alpha_i}{9}\right)$ Con esto se crea la siguiente ecuación:

$$s_{i+1} \geq s_i\left(1+\frac{\alpha_i}{9}\right) \tag{4.2}$$

   Entonces, usando las ecuaciones 4.1 y 4.2:

$$s_{i+1} \geq \frac{8}{9}s_1 + \frac{1}{9}s_1 + \frac{\alpha_i s_i}{9} = \frac{8}{9}s_i + \frac{s_i + \alpha_i s_i}{9} \geq \frac{8}{9}s_i + \frac{1}{9}\left(\frac{2}{3}\right) = \frac{8}{9}s_i + \frac{2}{27}$$

   Con lo anterior se crea una recurrencia de la forma $s_{i+1} = \frac{8}{9}s_i + \frac{2}{27}$. En [11] se establece que al resolver esta recurrencia se obtiene que

$$s_i \geq \frac{2}{3} - \frac{1}{6}\left(\frac{8}{9}\right)^i$$

En la última ronda $k = \left\lceil \frac{\log(6\epsilon)}{\log(\frac{8}{9})} \right\rceil$, se tiene que

$$s_k \geq \frac{2}{3} - \frac{1}{6}\left(\frac{8}{9}\right)^{\frac{\log(6\epsilon)}{\log(\frac{8}{9})}}$$

Aplicando un cambio de base se obtiene:

$$s_k \geq \frac{2}{3} - \frac{1}{6}\left(\frac{8}{9}\right)^{\log_{(8/9)}(6\epsilon)}$$

Por definición de logaritmo:

$$s_k \geq \frac{2}{3} - \frac{1}{6}(6\epsilon) = \frac{2}{3} - \epsilon \therefore$$

$|M_k| \geq (\frac{2}{3} - \epsilon)|OPT|$. Lo cual es una $\left(\frac{2}{3} - \epsilon\right)$-aproximación.

Así, en cualquier caso, se tiene el resultado. □

Con esto se demuestra que el algoritmo presentado obtiene una aproximación buena de un emparejamiento máximo a partir de una gráfica bipartita que recibe como entrada en el flujo. Esta idea de obtener una solución base de manera simple y luego irla mejorando es una técnica que puede servir para crear aproximaciones para otros problemas en el modelo de semiflujo.

5. Muestreo en Flujos

En este capítulo se explora una técnica muy útil en situaciones donde no conviene o no se puede almacenar toda la entrada de un algoritmo, tal como es el caso de los algoritmos de flujos. La técnica de *muestreo* (*sampling*) es una técnica que permite almacenar solo una proporción de la entrada, tratando de preservar las propiedades relevantes al considerar trabajar con solo un subconjunto de datos de entrada.

El *muestreo* está presente en otras áreas de las matemáticas, en particular es parte fundamental de la Estadística. En computación, a lo largo de los años, se han estudiado y diseñado algoritmos que obtengan propiedades de un conjunto de datos por medio de esta técnica. En particular, en este capítulo se estudia esta técnica como método de diseño de algoritmos de flujos.

En el capítulo se diseña un algoritmo de flujos que actúa sobre una gráfica y que calcula un muestreo de las aristas de esta (recuerde que el flujo consiste en la secuencia de aristas de la gráfica de entrada). De este modo, el algoritmo obtiene una subgráfica que se pretende sea representativa con respecto a la gráfica original, de modo que al calcular las propiedades relevantes en la subgráfica muestreada se obtenga una buena estimación de estas propiedades en la gráfica original.

El muestreo se refiere a tomar elementos del flujo con cierta probabilidad, es decir, dada una probabilidad $0 < p < 1$, los elementos del flujo se consideran en la subgráfica muestreada con probabilidad p de estar y con probabilidad $1 - p$ de no estar. Hay varias maneras para considerar estas probabilidades: puede ser la misma para cada arista (muestreo uniforme), o una distinta para cada arista (muestreo no-uniforme), incluso puede que la probabilidad de que una arista particular e sea considerada o no, dependa de las elecciones previas realizadas por el algoritmo, en este caso el muestreo no es *independiente*.

La diferencia entre muestreo *independiente* contra muestreo *no independiente* obedece a la independencia o no de las variables probabilísticas involucradas en el muestreo.

5.1. Muestreo Uniforme

Para poder tomar una muestra de los datos del flujo es necesario procesarlos y decidir si se incluirán o no en la muestra. La manera más sencilla de hacer eso es determinar una probabilidad de éxito de muestreo p. Este es un número positivo fijo menor a 1 y se recibe como entrada del algoritmo (se puede suponer que es un parámetro ajeno al flujo).

Dada la probabilidad de muestreo p, se procede a leer uno a uno los elementos del flujo y determinar si será ingresado o no el elemento en cuestión s, esto se puede hacer simplemente al generar un número aleatorio g y comparar g con p para determinar si el elemento s formará parte de la muestra. A este muestreo se le conoce como*muestreo independiente uniforme* ya que todos los elementos tienen la misma probabilidad de ser elegidos en la y la inclusión o exclusión de uno es independiente con las decisiones tomadas para los demás elementos (se

espera que el lector esté familiarizado con eventos independientes de variables probabilistas, de lo contrario consultar [21]).

En 5.1 se encuentra el pseudocódigo para ejecutar un muestreo independiente y uniforme a un flujo de datos para obtener una muestra H a partir de los elementos del flujo S.

Algoritmo 5.1 Algoritmo que obtiene una muestra independiente y con probabilidad uniforme a partir de un flujo de datos.

Entrada S un flujo de elementos, p número real positivo menor que 1.
Salida H un subconjunto de S que representa una muestra de él.

```
H ← ∅.
for s ∈ S do
    g ← número aleatorio en el intervalo (0, 1).
    if g ≤ p then
        H ← H ∪ {s}.
    end if
end for
return H.
```

En el caso cuando los elementos del flujo son aristas de una gráfica G. El resultado del muestreo es un subconjunto de aristas $F \subseteq E(G)$. Este subconjunto induce una subgráfica $\tilde{G}$ de la gráfica original recibida en el flujo. La idea detrás del muestreo es que la gráfica $\tilde{G}$ sea más pequeña que la gráfica original y además sería bueno que preservara ciertas propiedades relevantes para el problema en cuestión. Esto último es deseado pues se puede esperar que la subgráfica muestreada sea lo suficientemente pequeña y significativa para poder realizar un algoritmo convencional sobre ella y al obtener una respuesta se obtenga una respuesta relevante de la propiedad buscada en la gráfica original.

5.1.1. El Problema de encontrar un Corte Mínimo en Gráficas

En estos párrafos se presenta un problema bastante estudiado en teoría de las gráficas y se plantea cómo abordarlo en el modelo de flujos.

Suponga que se tiene una gráfica conexa G y que se quiere determinar aquellas aristas que son más relevantes en el sentido de la conexidad, es decir a aquellas aristas que al removerlas, la gráfica se desconecte en dos subgráficas ajenas por vértices, en particular si existe una arista de este estilo a esta se le llama *puente*. Por ejemplo en un árbol, todas las aristas son puentes pues al remover cualquiera, el árbol se desconecta y se forman dos árboles ajenos por vértices.

Hay gráficas para las cuales no existen dichas aristas puente, pero observe que si se quitan todas las aristas, la subgráfica resultante es disconexa, por definición. De este modo es claro que al ir removiendo aristas la gráfica se desconecta eventualmente. Lo interesante es encontrar el menor número de aristas que es necesario remover para que esto ocurra. A este número se le conoce como el *número de corte* de una gráfica G. Y a cualquier conjunto de aristas que al removerlas logren que la subgráfica resultante sea disconexa se le conoce como

conjunto de corte. El problema del *corte mínimo* consiste en hallar un conjunto de corte de tamaño exactamente el número de corte de la gráfica, es decir, un conjunto de corte de menor tamaño posible.

El problema del corte mínimo, es entonces un problema de optimización, pues se quiere minimizar la cardinalidad de un conjunto que cumpla con la propiedad de ser de corte. Este problema se puede generalizar si la gráfica incluye pesos pues ahora se puede encontrar un conjunto de corte de menor peso posible. Esta variación toma gran relevancia debido a que es equivalente a hallar el flujo máximo en una red de flujos (un tipo especial de gráfica) como lo muestra el teorema del flujo máximo-corte mínimo presentado en [8, captítulo 26]. Este último problema es muy importante en problemas de investigación de operaciones, logística, optimización de recursos, etc y el teorema de flujo máximo-corte mínimo es muy importante pues relaciona dos conceptos en principio ajenos.

Debido a la equivalencia entre los problemas del corte mínimo y el flujo máximo, ambos se han estudiado extensamente a lo largo de la historia y en particular se han hallado diversos algoritmos que los encuentren de manera exacta y otros que los aproximen (en particular que aproximen el tamaño del corte mínimo).

El problema con todos los algoritmos para hallar el corte mínimo (o el flujo máximo) de manera exacta es que todos requieren tener a toda la gráfica en memoria para verificar conexidad y determinar la vecindad (conjunto de aristas incidentes) de un vértice dado. Por este motivo no se ha ideado un algoritmo de flujos que calcule el corte mínimo de manera exacta. De hecho en [25] los autores demuestran que con una sola pasada al flujo es imposible hallarlo utilizando $o(n^2)$ bits de memoria, donde n es el número de vértices de la gráfica. Esto deja en claro que este problema es muy complicado de tratar en el modelo de flujos.

A pesar de que hallar el corte mínimo de manera exacta sea imposible en una pasada, se puede utilizar la técnica de muestreo descrita anteriormente para hallar una aproximación a la solución. De nuevo, el objetivo de esta técnica es encontrar un subconjunto del conjunto de datos del flujo que pueda preservar ciertas propiedades para que al ejecutar algoritmos sobre el subconjunto se encuentren aproximaciones o relaciones con el conjunto de datos original.

En el caso del problema del corte mínimo se puede muestrear la gráfica recibida en el flujo para hallar una subgráfica con menos aristas que la original y sobre esta calcular de manera exacta el corte mínimo.

Debido a que se espera que la subgráfica muestreada tenga aristas suficientes para que quepan en memoria, entonces se puede proceder a ejecutar un algoritmo convencional sobre estas y así se tenga el tamaño de un corte mínimo para la gráfica muestreada, a partir de esto se puede aproximar el tamaño del corte mínimo en la gráfica original.

En la siguiente subsección se habla más del algoritmo de flujo en detalle y de ciertos requisitos que se deben pedir para garantizar ciertos errores en las aproximaciones pedidas.

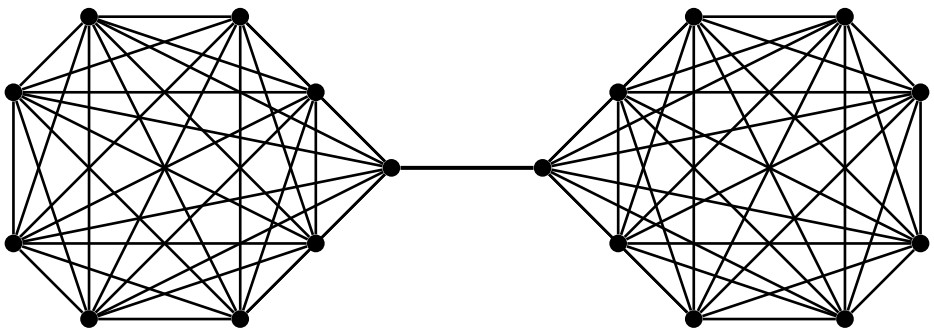
5.1.2. Algoritmo de Flujos para Aproximar el Corte Mínimo

Como ya se mencionó anteriormente, el esquema de muestreo presentado en 5.1 funciona para obtener una subgráfica H cuando el flujo S consta de las aristas de una gráfica G. Este muestreo se hace tomando en cuenta una probabilidad dada p. Sin embargo, si el muestreo se hace con una probabilidad arbitraria p no se garantiza mucho acerca de las condiciones de H. Por ejemplo, si p es muy pequeña, se puede esperar que H tenga pocas aristas y por el contrario; si p es cercana a 1, entonces H puede tener casi tantas aristas como G.

En este sentido, y de acuerdo a la probabilidad p, el orden esperado de H es el orden de G por la probabilidad. Es decir, la esperanza o valor esperado de aristas que tendrá la gráfica muestreada es el producto del número de aristas de la gráfica original recibida en el flujo, por la probabilidad utilizada para el muestreo.

A pesar de que el orden de la muestra es proporcional al orden original, no se pueden establecer garantías en cuanto a que en la muestra se preserve el tamaño de los cortes de la gráfica original, pues por ejemplo, si se considera una gráfica "mancuerna" como la de la figura 5.1, entonces, si la probabilidad es $\frac{1}{2}$, el valor esperado es casi el tamaño de uno de los "discos", sin embargo, si la arista puente no se encuentra en el muestreo, el corte mínimo allí podría ser mucho más grande que el corte mínimo original.

Figura 5.1: Gráfica Mancuerna que al ser muestreada puede no preservar una relación entre los tamaños de los cortes mínimos con la muestra.



El algoritmo de flujos es presentado en 5.2 y recibe un flujo compuesto por las aristas de la gráfica y una probabilidad fija p.

Considerando que el muestreo del algoritmo 5.2 no ofrece ninguna garantía en cuanto a la relación entre el tamaño de los conjuntos de corte de una gráfica y de su muestra, el número k puede ser tan cercano o tan lejano al buscado por igual.

Para corregir esta situación se presentan a continuación algunos resultados ya existentes sobre muestreo en gráficas para el problema del corte mínimo, aunque el autor no consideraba el modelo de flujos, la manera en la que se realiza el muestreo es totalmente aplicable a este modelo por lo que sus resultados son válidos.

Algoritmo 5.2 Algoritmo que obtiene una aproximación al problema del corte mínimo en una gráfica a partir del muestreo uniforme e independiente de la misma.

Entrada S un flujo de elementos que representan las aristas de una gráfica G, p número real positivo menor que 1.
Salida k un número que es una aproximación a la cardinalidad del corte mínimo en G.

```
H ← ∅.
for e ∈ S do      //Inicia el muestreo
    g ← número aleatorio en el intervalo (0, 1).
    if g ≤ p then
        H ← H ∪ {e}.
    end if
end for
k ← el tamaño del corte mínimo en H.      //Se estima el corte en el muestreo.
return k · p
```

Muestreo Uniforme con Error Acotado

Las ideas que a continuación se describen fueron presentadas por David R. Karger en 1999, en el artículo [13]. En estas se describe qué condiciones debe cumplir la probabilidad de muestreo para poder estimar más claramente qué tan cercano será el tamaño de un corte mínimo en el muestreo con respecto a uno en la gráfica original.

Al muestrear una gráfica G con probabilidad p se puede considerar una gráfica $\tilde{G}$ con el mismo conjunto de vértices y con las mismas aristas solo que a cada arista se le asigna un peso que es la probabilidad p. Si se considera un corte X en $\tilde{G}$, este tiene un peso asociado, que es $p|X|$. Este valor es justamente el valor esperado del corte X en la muestra H del algoritmo, es decir. Por cada corte en G, el valor de ese corte en $\tilde{G}$ es el valor esperado de ese corte en el muestreo.

Con esta gráfica en mente, si se considera un corte mínimo $\tilde{c}$ en $\tilde{G}$, este tendrá un valor esperado $p \cdot c$, donde c es el valor de un corte mínimo en G. En [13] demuestran lo siguiente:

Teorema 5.1. *Si $\tilde{c} \geq \ln(n)$, entonces sea $\epsilon = \sqrt{3(3)(\ln n/\tilde{c})}$, con probabilidad $1 - O(1/n)$, todo corte en el muestreo H es una ϵ-aproximación de su valor esperado.*

Para demostrar el teorema utilizan los siguientes resultados y definiciones:

Definición 5.2 ($\alpha-$corte mínimo)**.** Un *$\alpha-$corte mínimo* es un corte cuyo valor es a lo más α veces el valor de un corte mínimo global.

Lema 5.3. *En una gráfica no dirigida, el número de $\alpha-$cortes mínimos es menor a $n^{2\alpha}$.*

Lema 5.4 (Cota de Chernoff)**.** *Sea X la suma de variables aleatorias independientes boolea-*

nas (Bernoulli) con probabilidad de éxito p y valor esperado $\mu = \sum p$. *Entonces para* $\epsilon \leq 1$*:*

$$Pr[|X - \mu| > \epsilon\mu] \leq 2e^{-\epsilon^2\mu/3}$$

Con esto se puede demostrar el teorema.

Demostración del teorema 5.1. Sea $r = 2^n - 2$ el número de cortes en una gráfica. Este número resulta de la manera en que se pueden partir a los vértices en exactamente dos conjuntos ajenos. Sean $c_1, c_2, \ldots, c_r$ los valores de los cortes esperados en el muestreo ordenados ascendentemente. De modo que $\tilde{c} = c_1 \leq c_2 \leq \cdots \leq c_r$.

Sea p_k la probabilidad de que el corte k-ésimo diverja de su valor esperado por más de ϵ. Entonces la probabilidad de que algún corte diverja de su valor esperado por más de ϵ es $\sum_{k=1}^{r} p_k$. Se quiere acotar por arriba esta suma para establecer un orden asintótico.

Observe que el valor esperado de la muestra es una suma de variables booleanas (Bernoulli) por lo que por la cota de Chernoff se tiene que

$$p_k \leq 2e^{-\epsilon^2 c_k/3}$$

En la ecuación anterior p_k es justo la probabilidad izquierda en la cota. Recuerde que $\epsilon = \sqrt{3(3)(\ln n/\tilde{c})}$, por lo que para el valor del corte mínimo muestreado $\tilde{c}$ se tiene que:

$$e^{-\epsilon^2\tilde{c}/3} = e^{-3\ln n} = n^{-3}$$

Ahora se considere a los primeros n^2 cortes más pequeños. Todos ellos cumplen que $c_k \geq \tilde{c}$, y por lo tanto $p_k \leq 2n^{-3}$, esto porque a medida de que $c_k \geq \tilde{c}$ se tiene que $e^{-\epsilon^2 c_k/3} \leq e^{-\epsilon^2\tilde{c}/3}$, por lo tanto se cumple que:

$$\sum_{k \leq n^2} p_k \leq (n^2)(2n^{-3}) = 2n^{-1} \in O\Big(\frac{1}{n}\Big)$$

Considere ahora a los cortes más grandes. De acuerdo al lema 5.3, hay menos de $n^{2\alpha}$ cortes cuyo valor esperado sea menor a $\alpha\tilde{c}$. Como se han enumerado los cortes de modo ascendente, se tiene que $c_{n^{2\alpha}} \geq \alpha\tilde{c}$. Por lo que para el corte $k = n^{2\alpha}$ se observa que: $\alpha = \frac{1}{2}\log_n(k) = \frac{1}{2}(\frac{\ln k}{\ln n})$, por el cambio de base. Por lo anterior

$$c_k \geq \frac{\ln k}{2\ln n} \cdot \tilde{c}$$

Y así se concluye que para toda $k > n^{2\alpha}$:

$$p_k \leq 2k^{-3/2}$$

Finalmente, si se considera a $\alpha = 1$ para determinar el número de cortes que se alejan del mínimo se concluye que:

$$\sum_{k > n^2} p_k \leq \sum_{k > n^2} 2k^{-3/2}$$

Y la última suma es menor o igual que

$$\int_{n^2}^{\infty} 2k^{-3/2} = 4k^{-1/2}\Big|_{n^2}^{\infty}$$

Esta evaluación está en el orden de $O(\frac{1}{n})$

Por lo que todas las probabilidades p_k sean mayores o menores a n^2 están en este orden, dando por terminada la demostración. □

El teorema 5.1 da un error acotado cuando el tamaño esperado del corte mínimo $\tilde{c}$ es mayor a $9\ln(n)$. Recordando que $\tilde{c} = p \cdot c$, donde p es la probabilidad de muestreo y c el tamaño del corte mínimo, si se considera a $p \geq \frac{9\ln(n)}{\epsilon^2 c}$, se tiene que $\tilde{c} \geq 9\ln n$, por lo que se puede utilizar el teorema 5.1 para estimar con alta probabilidad que al calcular el corte mínimo en el muestreo, este será una $\epsilon-$aproximación de su valor esperado, por lo que si este tamaño se divide por la probabilidad p, con alta probabilidad se tiene una $\epsilon-$aproximación del corte mínimo para la gráfica original.

El detalle con este algoritmo de muestreo es que para poder acotar el error parece ser necesario conocer el tamaño del corte mínimo c. Sin embargo basta con poder aproximarlo, por ejemplo si se tiene una $3-$aproximación de c, se puede considerar la probabilidad de muestreo con este valor, y observe que la probabilidad de muestreo será mayor a la necesaria, por lo que el teorema sigue siendo válido. Además, esta probabilidad modificada solo es mayor por un factor constante, por lo que el tamaño de las aristas de la gráfica muestreada sigue en el mismo orden. Por ejemplo se puede utilizar el algoritmo de Matula presentado en [19] para hallar dicha aproximación a c.

Con todo esto se puede observar que la técnica de muestreo es sencilla, pero poderosa cuando se ponen condiciones a la probabilidad de muestreo, en particular en este caso, si la probabilidad es alta se puede tener una muy buena aproximación para un problema muy estudiado. El muestreo uniforme independiente es el tipo más sencillo de muestreo y por supuesto, sirva para cualquier tipo de flujo, no solo para aristas. De hecho, se puede utilizar muestreo casi directamente en cualquier flujo e incluso, si el tamaño de la muestra se asegura es pequeño (que quepa en memoria), se puede pensar en utilizar un algoritmo convencional sobre ella para obtener datos o propiedades del flujo.

6. Momentos de Frecuencia y un Problema de Conteo

El fundamento de este capítulo es presentar una técnica de diseño basada en el uso de los llamados *momentos de frecuencia* de una secuencia. Estos dictan estadísticas y propiedades de los datos de entrada y son muy utilizados en estadística y otras áreas.

En los últimos años se han desarrollado varios algoritmos de flujos que intentan calcular eficientemente estos momentos y también se utilizan estos algoritmos como bloques para generar algoritmos de flujos que resuelvan otros problemas.

Cabe mencionar que este capítulo es el más complicado de este trabajo, pues se presentan y analizan conceptos que no son de dominio común en un curso convencional de análisis de algoritmos.

En la primera sección se introducen los momentos de frecuencia de una secuencia, su utilidad y diversas aplicaciones. Se presentan también referencias en donde se pueden conocer más a detalle cómo se implementan estos algoritmos. Al final de la sección se presenta una técnica de reducción de algoritmos en el modelo de flujos que consiste en utilizar el cálculo de los momentos de frecuencia para crear nuevos algoritmos de flujos.

En la segunda sección se presenta una aplicación de los momentos de frecuencia y la técnica de reducción al crear un algoritmo que aproxime el cálculo del número de triángulos de una gráfica, así como las demostraciones que sustentan la eficiencia, también se presenta una manera de transformar eficientemente el flujo real en uno virtual para que el algoritmo sea eficiente.

6.1. Momentos de Frecuencia y Reducciones

En esta sección se define un concepto que es muy utilizado para el desarrollo de algoritmos de flujos: los momentos de frecuencia de una secuencia. Al final de la sección se describe una manera de utilizar estos momentos para crear nuevos algoritmos de flujos.

Los momentos de frecuencia son utilizados en muchos otros problemas (véase [1]), en particular se consideran de gran utilidad en los sistemas de bases de datos concurrentes como se observa en [9]. Desde que Alon et al. establecieron cómo aproximar el cálculo de dichos momentos en el modelo de flujos, ha habido otros autores interesados en mejorar estos algoritmos y en utilizarlos como bloques fundamentales para el procesamiento masivo de datos. Avrim Blum, John Hopcroft y Ravindran Kannan lo consideran así al incluir estos algoritmos en su libro *Foundations of Data Science*, el cual se encuentra en proceso de creación y se puede consultar en [5].

Los momentos de frecuencia de una secuencia de datos se pueden definir de la siguiente manera:

Sea A un conjunto finito, no vacío. $A = \{a_1, \ldots, a_m\}$. Sea $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ una secuencia de elementos (posiblemente repetidos) de A. Es decir, Σ es un multiconjunto de elementos de A.

Definición 6.1 (Frecuencia de un elemento en una secuencia de datos)**.** Dado $a_j \in A$, se define $f_j = |\{i \in [n] | \sigma_i = a_j\}|$, es decir, f_j es el número de apariciones de a_j en la secuencia Σ.

Definición 6.2 (Momentos de frecuencia de una secuencia)**.** Los *momentos de frecuencia* de una secuencia son una familia de funciones $F_k :: \Sigma \mapsto \mathbb{R}\ \forall k \in \mathbb{Z}, k \geq 0$ que se definen como:

$$F_k = \sum_{j=1}^{m} f_j^k$$

En particular, dada k, a F_k se le llama el *k-ésimo* momento de la secuencia.

Por ejemplo el momento F_0 es el número de elementos distintos de la secuencia, pues todos los $f_j > 0$ valen 1 y son positivos solo si aparecen al menos una vez en la secuencia. El momento F_1 es simplemente contar el tamaño del flujo pues cada f_j aporta tantos valores como repeticiones tenga en él. Los demás momentos aportan valores estadísticos a la frecuencia y en este trabajo solo se utilizarán: F_0, F_1 y F_2.

Calcular estos momentos de manera exacta es costoso en memoria [1] por lo que el cálculo de estos se aproximará. En la siguiente sección se presenta uno de los primeros algoritmos que existieron para aproximar el momento F_0

6.1.1. Algoritmo de Flajolet y Martin para estimar F_0

En 1985 Philippe Flajolet y G. Nigel Martin presentaron un algoritmo el cual aproxima el cálculo de F_0. Este algoritmo se encuentra en [12] y sirvió como base para que se investigará más el cálculo de estos momentos, sobretodo en el modelo de semiflujo en [1]. En los siguientes párrafos se da la idea general del funcionamiento del algoritmo, sin embargo para más detalles el lector puede consultar el artículo original.

El algoritmo estima F_0 de un conjunto de datos que recibe como entrada, esto lo hace por medio de una función *hash* que mapea estos datos a una cadena binaria de longitud L de manera uniforme. Es decir, que recibe un dato de la secuencia y lo transforma a una cadena binaria de longitud fija de modo que cualquier cadena binaria tiene la misma probabilidad de ser imagen de algún dato.

Además de la función *hash* se define a la función $bit(y, k)$ como la asignación que obtiene el k-ésimo bit en la cadena binaria y. También se define la función $\rho(y)$ que representa la posición del bit 1 menos significativo en cadena binaria y, para definir bien esta función se debe adaptar una convención para $\rho(0)$ pues la cadena binaria compuesta por puros ceros no

tiene un 1. La función se define así:

$$\rho(y) = \begin{cases} \text{mín}(bit(y,k) = 1) & \text{si } y > 0 \\ L & \text{si } y = 0 \end{cases}$$

Recuerde que L es la longitud de las cadenas binarias usadas por la función *hash*.

Con esto se puede definir el pseudocódigo para un algoritmo que estime la cantidad de elementos distintos de un conjunto de datos M, es decir, el algoritmo estimará F_0. El pseudocódigo se encuentra en 6.1.

Algoritmo 6.1 Algoritmo de Flajolet y Martin para estimar F_0 de un conjunto de datos

Entrada *Bitmap* es un arreglo de tamaño L, la función *hash* mapea registros a cadenas binarias de longitud L de manera uniforme, el conjunto M tiene n registros de los cuales se estimará cuántos de ellos son distintos.

Salida Un número que estima cuántos valores distintos tiene el conjunto M de entrada.

```
for i = 1 to L − 1 do
    Bitmap[i] ← 0.
end for
for x ∈ M do
    index ← ρ(hash(x)).
    Bitmap[index] ← 1.
end for
i ← 0.
while Bitmap[i] > 0 do
    i ← i + 1.
end while
return 2^i.
```

Para entender cómo funciona el algoritmo 6.1 se debe observar lo siguiente: como la función *hash* mapea los valores de manera uniforme, entonces la probabilidad de que la representación de un elemento x tenga un 1 en el bit menos significativo es $\frac{1}{2}$, por lo que la función $\rho(hash(x))$ será 0 la mitad de las veces (recuerde que la función ρ da una posición de la cadena binaria). Esta función ρ dará como resultado 1 un cuarto de veces ($\frac{n}{4}$) y así sucesivamente, por lo que al final el arreglo *Bitmap* tendrá grandes posibilidades de tener sus primeras entradas en 1 y sus últimas entradas en 0.

Las entradas i del arreglo *bitmap* que probablemente sean 1 son todas aquellas que cumplan que $i << \log_2(n)$ ya que en estos casos al dividir n entre 2^{i+1} el resultado será mayor a 1 si i tiende a cero y muy cercano a 1 si i tiende a $\log_2(n)$. Análogamente, si $i >> \log_2(n)$, entonces el resultado de la división será muy cercano a 0 por lo que probablemente esas entradas efectivamente sean 0. Observe que en la zona en que $i \approx \log_2(n)$ las divisiones son cercanas a 1 por lo que esas posiciones se accedan tal vez una o dos veces, por lo que se tendrán 0 y 1 intercalados.

Con lo anterior, al considerar la primera posición que sea cero del arreglo de *Bitmap*: k y devolver 2^k, probablemente $k \approx \log_2(n)$ lo que hace hace que se devuelva algo que es

cercano al número de elementos distintos de M, ya que los elementos repetidos hacen que se manipulen las mismas entradas de *Bitmap*.

El algoritmo de Flajolet y Martin es muy básico en comparación con los algoritmos creados después por Alon et al [1] y por Ziv Bar-Yossef et al [3]. Aquí se expone a modo de ejemplo para familiarizar al lector con algoritmos que estimen estos momentos, pero los algoritmos utilizados en la realidad son mucho más complejos y complicados y escapan al alcance de este trabajo.

6.1.2. Reducciones en Algoritmos de Flujos

El concepto de *reducción* en análisis y diseño de algoritmos es muy conocido y consiste en transformar la entrada de un problema A en una entrada para un problema B, de modo que al resolver B, esta solución se pueda usar como subrutina para hallar una solución para A.

Esta idea es muy utilizada, sobretodo en la teoría de la complejidad para establecer cotas inferiores o probar NP-Completud. En [3] se define una técnica de reducción de problemas de flujos y está enfocada en la creación de nuevos algoritmos a partir de la aproximación de los momentos de frecuencia.

El concepto fundamental es la mismo: Se tiene un problema de flujos A para el cual se quiere crear un algoritmo eficiente de flujos y se tienen algoritmos eficientes para los problemas $B_1, \ldots, B_k$. La técnica consiste en tomar un elemento del flujo de A y transformarlo en una serie de elementos para cada uno de los flujos de los problemas $B_i, 1 \leq i \leq k$. Así, se utilizan estas entradas en cada uno de los algoritmos de flujos y se obtienen $S_i, \ldots, S_k$ salidas, una para cada problema B_i y se juntan estas salidas para generar una salida para A.

Esta reducción fue ideada por Ziv Bar-Yossef, Ravi Kumar, D. Sivakumar en [3] y utiliza a los momentos de frecuencia F_k como los problemas "básicos" B_i ya que allí se describen varios algoritmos eficientes en el modelo de flujos para aproximar estos momentos. Así mismo se pone como ejemplo de uso de esta técnica el conteo de triángulos en una gráfica, el cual se analizará en la siguiente sección.

Observe que al transformar un elemento del flujo para el problema A en varios elementos para los flujos de los problemas B_i se podrían estar generando muchos elementos en cada flujo para B_i. Estos elementos podrían ser tantos que se podría necesitar un orden lineal de operaciones, con respecto al tamaño del flujo para A, para generarlos. Esto haría imposible la reducción en el modelo, pues cada elemento del flujo A tomaría tiempo $O(n)$ en ser procesado.

Con la observación anterior en mente en [3] definen lo que es que un algoritmo de flujos sea *eficiente en rangos* y la idea de esto es que esta transformación del flujo para la reducción se pueda hacer de manera eficiente y simple mediante una representación sucinta (abreviada). En la última parte de la siguiente sección se habla más acerca de esta representación del flujo virtual en el caso del conteo de triángulos.

Cabe mencionar que el combinar los momentos de frecuencia y las reducciones de algoritmos de flujos utilizando estos momentos, son técnicas generales de diseño presentadas por primera

vez en [3] y se consideran suficientemente generales para resolver problemas de flujos; en este ambiente, una aplicación de estas técnicas es el conteo de triángulos en una gráfica y aunque este problema es interesante de por sí, en el artículo solo es una ejemplificación de las técnicas.

Sin embargo, para este trabajo es más valioso el entender la técnica de diseño que desarrollar con formalidad tanto los algoritmos que aproximan los momentos F_0 y F_2, ya que el desarrollo formal de estos temas excede el alcance de este trabajo, por ello se invita al lector interesado en estos temas en consultar [1] y [3] para más detalles de estos algoritmos.

En relación a las aproximaciones de estos algoritmos en [3] se demuestran el siguiente teorema:

Teorema 6.3. *Existe un algoritmo de flujos que produce una* (ϵ, δ)*-aproximaciónpara* F_0 *utilizando* $O(\frac{1}{\epsilon^3} \log \frac{1}{\delta} \log m)$ *espacio y tiempo de procesamiento por elemento del flujo.*

Además en [1] se establece el siguiente teorema:

Teorema 6.4. *Existe un algoritmo de flujos que produce una* (ϵ, δ)*-aproximaciónpara* F_2 *utilizando* $O(\frac{1}{\epsilon^2 \log(\frac{1}{\delta}}(\log m + \log n)))$ *espacio y tiempo de procesamiento por elemento del flujo.*

Recuerde que en los teoremas anteriores: n es el tamaño del flujo y m el tamaño del conjunto universo de posibles elementos del flujo.

6.2. Conteo de Triángulos en una Gráfica

En esta sección se presenta una aplicación de la técnica de reducción al estimar los momentos de frecuencia para estimar una propiedad de una gráfica. La aplicación consiste en diseñar un algoritmo de flujos que actúe sobre una gráfica y cuente (aproxime) el número de triángulos que hay en ella de acuerdo a la siguiente definición:

Definición 6.5 (Número de triángulos de una gráfica G)**.** Sea $G = (V, E)$ una gráfica, sea $\Gamma = \{(x, y, z) | x, y, z \in V(G)\}$ el conjunto de todas las posibles ternas de vértices distintos en $V(G)$. Se definen los conjuntos $\Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3$ de la siguiente manera:

$\Gamma_i = \{(x, y, z) \in \Gamma |$ hay i aristas en G entre $x, y, z\}$.

Sean $T_0 = |\Gamma_0|, T_1 = |\Gamma_1|, T_2 = |\Gamma_2|, T_3 = |\Gamma_3|$. El *número de triángulos* de G es T_3.

Este problema es interesante, ya que saber cuántos triángulos tiene una gráfica dada se relaciona con calcular el coeficiente de agrupamiento (*clustering*) que se utiliza para estudiar estructuras y comportamientos en redes sociales [3]. El problema de contar triángulos y, en general, ciclos de longitud pequeña en una gráfica ha sido muy estudiado en el modelo convencional como se muestra en [2].

La idea de aproximar la cantidad de triángulos de la gráfica en lugar de obtener una cantidad exacta es debido a que hacer lo segundo en el modelo de semiflujo es imposible. En [7] se demuestra que con una sola pasada al flujo se necesita $\Omega(n^2)$ espacio, donde $n = |V(G)|$ y si se permite un número constante de pasadas (independiente en n) se necesita $\Omega(\frac{n^2}{T_3})$ espacio.

En ambos casos esta cantidad de memoria adicional no está permitida en el modelo, por lo que lo mejor que queda es aproximar esta cantidad.

Para realizar la aproximación del número de triángulos de la gráfica se utilizarán las aproximaciones de los momentos de frecuencia F_0 y F_2 de un flujo virtual, este nuevo flujo surge como transformación al flujo original y esta transformación se describe en los siguientes párrafos.

Generando el Flujo Virtual

Para poder aproximar los momentos F_0 y F_2 es necesario transformar el flujo de aristas a un flujo virtual apropiado para que estos algoritmos funcionen. Para la transformación solo es necesario tener el número de vértices totales de la gráfica: n. Los vértices se representan con los primeros n enteros y una arista $e = (u, v)$ consta de los enteros $u \leq n$ y $v \leq n$. Cuando en el flujo original se presenta la arista e, en el flujo virtual se crean una serie de ternas que constan de todas las posibles que tienen a u y v como elementos, por supuesto $u \neq v$ y se puede suponer que $u < v$. A este conjunto de ternas se le llama Γ_e. Observe que $|\Gamma_e| = n - 2$ ya que hay n vértices en la gráfica y el primer elemento es u, el segundo es v y el tercer elemento puede ser cualquiera de los $n - 2$ restantes. Cada una de las ternas de Γ_e es un objeto del nuevo flujo S'. Claramente $|S'| = (n - 2) \cdot |E|$.

A partir de la gráfica representada en la figura 6.1 se considera el flujo

$$S = \{(1, 2), (3, 4), (1, 3), (4, 5), (2, 3)\}.$$

Con este flujo S se puede obtener el siguiente flujo virtual:
$\{(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (2, 3, 4), (3, 4, 5), (1, 2, 3), (1, 3, 4),$
$(1, 3, 5), (1, 4, 5), (2, 4, 5), (3, 4, 5), (1, 2, 3), (2, 3, 4), (2, 3, 5)\}$

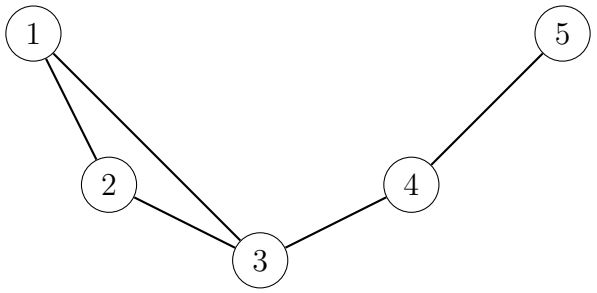


Figura 6.1: Una gráfica G, donde sus aristas se presentan en el flujo $S = \{(1, 2), (3, 4), (1, 3), (4, 5), (2, 3)\}$ del cual se obtendrá un flujo virtual.

Con lo anterior parecería que el flujo virtual es mucho más grande que el original, sin embargo este conjunto Γ_e se puede representar de una manera *compacta* de la siguiente manera: se consideran tres listas de ternas:

1. Las ternas que tienen a un vértice $k < u < v := [1, .., u - 1]$.
2. Las ternas que tienen a un vértice entre u y entre v: $[u + 1, .., v - 1]$

3. Las ternas que tienen a un vértices después de v: $[v+1, ..n]$.

Con estas tres listas se generan tres ternas descritas por un rango de valores:

$$([1, .., u-1], u, v), (u, [u+1, .., v-1], v), (u, v, [v+1, ..n])$$

Estas tres ternas son las que reciben los algoritmos que aproximan a F_0 y a F_2 y se pueden generar muy fácilmente a partir de cada arista del flujo original y conociendo el número total de vértices.

Una observación que hay que notar es la siguiente: Suponga que en G los vértices $(1, 2, 3)$ forman un triángulo (como en la figura 6.1. Entonces, cuando en el flujo aparezca la arista $(1, 2)$ en el flujo virtual se generará la terna $(1, 2, 3)$ por la tercera condición. Cuando en el flujo aparezca la arista $(1, 3)$, se volverá a generar una terna $(1, 2, 3)$ por la condición dos y finalmente, cuando en el flujo aparezca la arista $(2, 3)$ se generará de nuevo la terna $(1, 2, 3)$ por la primera condición. Con esto se puede concluir que en el flujo virtual, una terna aparece tantas veces como aristas haya entre ella en el flujo original, lo cual sustenta que cada elemento de Γ_i tiene i ternas en el flujo virtual.

Relación con los Momentos de Frecuencia

Como ya se mencionó se utilizarán los momentos de frecuencia y la técnica de reducción descrita anteriormente para aproximar este conteo, sin embargo se debe establecer una relación entre el número de triángulos que tiene una gráfica y entre los momentos de frecuencia de un flujo; esta relación se describe a continuación.

La relación de los momentos de frecuencia de este nuevo flujo $F_k(S')$ con T_1, T_2 y T_3 (este último es el que se quiere aproximar) se da por la siguiente igualdad:

$$F_k = F_k(S') = T_1 \cdot 1^k + T_2 \cdot 2^k + T_3 \cdot 3^k \tag{6.1}$$

Esto es cierto por la siguiente razón: Para calcular el momento de frecuencia F_k hay que calcular, por cada elemento que aparece en el flujo, sus repeticiones y elevar estas repeticiones a la k. Pero en la ecuación anterior lo que se está sumando son las ternas que tienen solo una arista entre ellas más las que tienen 2 más las que tienen 3. Observe que si una terna tiene i aristas entre ellas, entonces por cada una de estas i aristas, al momento de generar el flujo virtual, se genera la misma terna, por lo que toda terna tiene tantas repeticiones en el flujo virtual como aristas haya entre ellos en la gráfica. Así que para calcular el momento de frecuencia F_k basta con contar las ternas que tienen 1 arista (T_1) y sumar las que tienen 2, por el número de repeticiones a la k ($T_2 \cdot 2^k$) y finalmente las que tienen tres aristas y su número de repeticiones a la k ($T_3 \cdot 3^k$). De este modo la observación es correcta y con ella se pueden formar una serie de ecuaciones para los momentos F_0, F_1 y F_2:

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{pmatrix} \cdot \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix}$$

Al invertir el sistema anterior para obtener el valor de T_3 se tiene que:

$$T_3 = F_0 - 1.5F_1 + 0.5F_2$$

Claramente F_1 es el número de elementos del flujo por lo que se puede obtener de manera exacta así que solo se necesita aproximar F_0 y F_2. Estos dos momentos se pueden aproximar con parámetros (ϵ, δ) utilizando los algoritmos descritos en [3] y sustentados por los teoremas 6.3 y 6.4. Al obtener las aproximaciones de estos momentos se puede aproximar el número de triángulos.

Sean $\tilde{F_0}$ y $\tilde{F_2}$ las aproximaciones para F_0 y F_2 respectivamente. Estas son obtenidas por los algoritmos descritos en [3] con los parámetros $(\epsilon', \frac{\delta}{2})$, donde $\epsilon' = \frac{\epsilon}{8mn}$, con $m = |E(G)|$ y $n = |V(G)|$.

Cabe mencionar que Ziv Bar-Yossef et ál. en [3] crean algoritmos especiales para trabajar con el flujo virtual descrito anteriormente. Estos algoritmos son mejoras a los propuestos por Alon et al. en [1] y es con estos algoritmos con los que se obtienen las estimaciones mencionadas en el párrafo anterior. Para más detalles se invita al lector a consultar las referencias.

Sea $\tilde{T_3} = \tilde{F_0} - 1.5F_1 + 0.5\tilde{F_2}$ la aproximación obtenida a partir de las aproximaciones de F_0 y F_2 con los parámetros descritos anteriormente. A continuación se demuestra que esta es una ϵ-aproximación.

Lema 6.6. *La aproximación $\tilde{T_3}$ es una (ϵ, δ)-aproximación de T_3*

Demostración.

$$|T_3 - \tilde{T_3}| = |F_0 - 1.5F_1 + 0.5F_2 - (\tilde{F_0} - 1.5F_1 + 0.5\tilde{F_2})| = |(F_0 - \tilde{F_0}) + 0.5(F_2 - \tilde{F_2})| \leq$$

$$|(F_0 - \tilde{F_0})| + 0.5|(F_2 - \tilde{F_2})| \leq \epsilon' + 0.5\epsilon' = 1.5\epsilon'$$

Pero como se supone que la gráfica contenida en el flujo no es vacía, entonces: $F_1 \geq 1$, $F_0 > 0$ y $F_2 \geq 0$ $\therefore$

$$1.5\epsilon' < \epsilon'(F_0 + 1.5F_1 + 0.5F_2)$$

Por otro lado, por la igualdad (4.1), se tiene que:

$$T_1 + T_2 + T_3 = F_0 \leq F_1 = T_1 + 2T_2 + 3T_3$$

Y también:

$$\frac{F_2}{9} = \frac{T_1}{9} + \frac{4T_2}{9} + T_3 \leq T_1 + 2T_2 + 3T_3 = F_1 \therefore$$

$$\text{máx}(F_0, \frac{F_2}{9}) \leq F_1 = m(n-2)$$

Finalmente:

$$(F_0 + 1.5F_1 + 0.5F_2) \leq F_1 + 1.5F_1 + 4.5F_1 = 7F_1 = 7(m(n-2)) \leq 8mn$$

Con todo lo anterior se concluye que:

$$|T_3 - \tilde{T_3}| \leq 1.5\epsilon' < \epsilon'(F_0 + 1.5F_1 + 0.5F_2) \leq 8\epsilon' mn = \epsilon$$

Así que $\tilde{T_3}$ es una (ϵ)-aproximación de T_3.

Además, esto se logra con una probabilidad de fallo $\frac{\delta}{2} + \frac{\delta}{2} = \delta$ ya que cada una de las aproximaciones de F_0 o de F_2 pueden fallar con probabilidad $\frac{\delta}{2}$ (debido a los teoremas 6.3 y 6.4). Por lo tanto se puede generar un algoritmo que aproxime a T_3 con factores (ϵ, δ).

□

Con esto se ha demostrado que es posible aproximar el número de triángulos de una gráfica teniendo como bloques básicos las aproximaciones de los momentos de frecuencia. Los cálculos de estos se relacionan con el conteo de triángulos mediante la transformación de las aristas de la gráfica (flujo original) a una serie de ternas de vértices compactadas en formas de listas (flujo virtual).

Los momentos de frecuencia actúan sobre este flujo virtual y con su salida (la estimación) se puede crear una aproximación para el conteo de triángulos. Tome en cuenta que estas aproximaciones son probabilistas, por lo que dependen de un factor de probabilidad.

Así concluye la discusión de esta técnica de diseño que resulta ser la más complicada y es tal vez, la menos familiar al lector, pues incluye tanto conceptos teóricos como algoritmos de aproximación probabilistas y reducciones algorítmicas como también conceptos particulares de estadística como los momentos de frecuencia de una secuencia.

Conclusiones y Trabajo Futuro

A lo largo de este trabajo se presentaron diversas técnicas de diseño de algoritmos de flujos, ejemplificando cada técnica al intentar resolver un problema de teoría de gráficas. Sin embargo, estas técnicas no se limitan a resolver un problema dado, por el contrario, son lo suficientemente generales para atacar una gran variedad de problemas.

Este trabajo se puede apreciar como un catálogo de técnicas de diseño algoritmos de flujos teniendo una técnica diferente en cada uno de los capítulos. Estas técnicas se organizaron de acuerdo a su complicación, siendo las últimas aquellas que requieren conocimientos que no todo alumno de una licenciatura o maestría en computación pueda conocer. Las primeras técnicas son más fáciles de comprender debido a que utilizan conceptos o técnicas básicas de algoritmos y estructuras de datos.

Estas técnicas fueron consideradas tras analizar diversos artículos en donde se presentaban algoritmos de flujos, en particular del compendio realizado por Andrew McGregor en [20]. A partir de este trabajo se pensó en juntar y exponer varias técnicas de diseño de algoritmos de flujos a modo que introdujeran al lector a este modelo de algoritmos.

En la actualidad hay varios compendios como el de McGregor e incluso John Hopcroft aborda el tema en uno de los capítulos de su nuevo libro llamado "Foundations of Data Science" [5] que está en proceso de culminación. Hopcroft y otros científicos piensan que los algoritmos de flujos serán una herramienta básica en la creación de nuevos programas y la resolución de diversos problemas dentro de las ciencias de datos y en lo personal, comparto su opinión.

Debido a que se comparte la motivación e inspiración de Hopcroft se decidió presentar este trabajo de esta manera para ayudar a los estudiantes y a los profesores a interesarse e introducirse en el tema de un modo más explicado y liviano a lo presentado en un compendio a modo de artículo de revista o congreso y con mayor profundidad y serenidad a lo presentado en un libro donde se tratan otros modelos y temas.

Las aportaciones forman parte de los objetivos de este trabajo por lo que nunca se perdió de vista ese objetivo, en particular, la primera aportación es la presentación clara y concisa de una estructura de datos para conjuntos disjuntos multicoloreados. Es decir, se quiere que cada conjunto tenga a su vez una $k-$partición por colores y en el apéndice A.1 se presenta esta idea detallada. Aunque se presenta para conjuntos bicoloreados, la idea se puede generalizar a un número arbitrario de colores manteniendo las complejidades, siempre y cuando el conjunto de colores sea constante con respecto al número total de elementos involucrados.

La aportación más trascendente es la presentada en el capítulo 3 que involucra a los matroides. Lo que se hizo fue establecer un esquema de algoritmo *greedy* para matroides que fuera óptimo en el modelo de flujos. Algo equivalente a lo que ya se sabía para el modelo convencional. El esquema presentado aquí resulta ser óptimo para cualquier matroide, sin embargo la eficiencia del algoritmo queda sujeta a la manera en la que se represente el matroide en la memoria, en particular a la forma en la que se pueda determinar si un conjunto de elementos es independiente o no y a como se obtenga un elemento más pesado en el circuito. A pesar de esto, se tiene un esquema greedy general que puede funcionar en muchos casos y que promueve un método genérico para diseñar algoritmos de flujos y resolver problemas de

optimización.

Los algoritmos de flujos deben ser considerados como un modelo moderno y deben estar presentes en la formación de los científicos de datos y en la de los científicos de la computación. Este tipo de algoritmos poseen diversas técnicas de diseño lo que permite que se pueda abordar un problema de flujos de diversas maneras y haya más probabilidad de poder resolverlo o aproximarlo con alguna de las técnicas disponibles.

Cabe mencionar que este trabajo no necesariamente abarca todas las técnicas empleadas o disponibles para este tipo de algoritmos, sino solo algunas que han sido exitosas para diversos problemas y ejemplifican de buen modo la diversidad que existe en este modelo de algoritmos.

El trabajo futuro en el campo consiste en seguir resolviendo problemas o mejorando soluciones ya creadas en este modelo de flujos, para tener un abanico aún mayor de técnicas y recursos disponibles para las nuevas generaciones de investigadores en el área. Así mismo se puede empezar a crear literatura más específica en el tema que presente este tipo de algoritmos de manera sucinta y simple para los estudiantes de licenciatura.

En este trabajo se pretendió hacer lo último pero no es suficiente, se debe dar mayor difusión a estas técnicas y modelos novedosos que permiten lidiar con restricciones impuestas por la obtención y generación de información masiva de hoy día. Se considera que este tipo de algoritmos debe formar parte del conocimiento básico de todo licenciado en ciencias de la computación y de todo investigador en el área de algoritmos.

Este escrito puede mejorarse al incluir otras técnicas de diseño o al diseñar un manual de ejemplos donde se utilicen estas técnicas para resolver problemas en diversas áreas del conocimiento y no solo en la teoría de gráficas. También se puede diseñar un plan de estudios para una curso semestral o cuatrimestral para licenciatura o maestría basado en el contenido de este trabajo. Incluso se pueden proponer talleres y cursos para docentes en el área, de modo que se familiaricen con el tema y puedan llevarlo a sus aulas de manera simple y bien explicada.

A. Estructuras de Datos

En este apéndice se describen más en detalle las distintas estructuras de datos que se utilizan en los algoritmos presentados en este trabajo y que o bien no forman parte de un curso básico de estructuras de datos y algoritmos o bien fueron modificadas y adecuadas para trabajar con los algoritmos aquí presentados.

A.1. Conjuntos Disjuntos

Esta sección detalla la estructura de datos empleada en el algoritmo para hallar la bipartición de una gráfica. La estructura básica es presentada en más detalle en [8, capítulo 21].

La estructura de datos *conjuntos disjuntos* mantiene una serie de colecciones de elementos ajenas entre sí. Las operaciones de la estructura permiten crear un conjunto a partir de un elemento (`make_set`), unir dos conjuntos en uno solo (`union`) e identificar a qué conjunto pertenece un elemento (`find`).

Hay varias maneras de implementar esta estructura, en particular aquí se discute la opción de hacerlo mediante *bosques disjuntos*. Esta idea almacena una serie de nodos en memoria, donde cada nodo representa a uno de los elementos a almacenar en la estructura. Estos nodos se conectan entre sí por medio de un apuntador a otro nodo, este apuntador mantiene la relación de *parentesco*, es decir, si un nodo u tiene un apuntador a v, entonces se dice que v es padre de u. De este modo se mantiene una serie de árboles ajenos (un bosque de nodos) en la memoria. Con esta representación se explican a continuación el funcionamiento de las operaciones.

make_set
Esta operación crea un nuevo árbol con un elemento. El único nodo de este árbol tiene como apuntador a su padre a él mismo y está aislado de los demás. Este árbol representa a un conjunto unitario formado por un solo elemento.

union
Esta operación junta dos árboles en uno solo, haciendo que la raíz de uno apunte a la raíz del otro. Observe que la raíz de un árbol siempre apunta a sí misma, así que en esta operación se modifica para hacer que apunte a la raíz del otro.

Para que esta operación sea eficiente se utiliza una heurística que siempre une al conjunto más pequeño con el más grande, en otras palabras, el grande absorbe al pequeño. Esto se hace agregando un atributo a cada nodo llamado su rango y cuando se unen dos árboles se compara el rango de sus raíces. El que tenga rango menor se une al que tenga rango mayor; en caso de empate se escoge que el primero se una al grande. El rango de un nodo es la longitud del camino más largo desde la raíz hasta una hoja.

Cuando se unen dos árboles, el tamaño de la unión es al menos el doble que el del uniendo más pequeño debido a la heurística de la unión por rango. Esto implica que la altura de los árboles es de $O(\log(n))$.

Es importante notar que la unión une dos elementos, pero comparara raíces de árboles por lo que primero se debe saber quién es la raíz de un árbol. Esto se hace mediante la operación `find`.

find
Esta operación identifica a qué árbol pertenece un elemento siguiendo el camino de apuntadores del nodo del elemento hasta la raíz. La raíz de un árbol es el representante de todos sus elementos.

Como los árboles tienen altura logarítmica, esta operación puede tardar ese tiempo, sin embargo, si se utiliza otra heurística conocida como *compresión de camino*, las operaciones subsecuentes de `find` para varios vértices del árbol tomarán menor tiempo. Esta heurística consiste en que se actualicen los apuntadores de padres de ciertos nodos de modo que apunten directamente a la raíz del árbol. Así el hecho de preguntar por el representante nuevamente tomará $O(1)$. Los apuntadores que se actualizan son todos aquellos que forman parte del recorrido del nodo original hasta la raíz. Es decir, se actualizan los apuntadores de todos los descendientes del nodo incluido, él.

Ya que se sabe cómo funcionan la operaciones de la estructura se debe argumentar el tiempo de ejecución. Aquí no se demostrarán las complejidades pues requieren de un análisis detallado que escapa del enfoque del trabajo. Las operaciones de unión y búsqueda (`find`) toman tiempo amortizado de $O(\alpha(n))$. Esta función es la inversa de una función que crece muy rápido, por lo que esta, crece muy lento. Para que $\alpha(n)$ valga 5 n debe ser mayor que la cantidad de átomos en el universo visible. Así que este valor es muy muy pequeño aún cuando n sea muy grande.

A.1.1. Conjuntos disjuntos bicoloreados

A continuación se detalla la construcción de la estructura que se emplea en el algoritmo 2.2. La idea de la estructura es mantener conjuntos disjuntos pero que a su vez tienen una partición dentro de ellos. El tamaño de esta partición puede ser arbitrario, aunque en este caso solo se necesitan dos partes pues solo se necesita mantener 2-colores simultáneos.

Esta implementación utiliza la idea central de la implementación de los conjuntos disjuntos tradicionales utilizando las mismas ideas para realizar las operaciones. El cambio principal radica en la unión pues en este caso se puede necesitar unir con más cuidado cada una de las partes de ambos uniendos para tener el resultado deseado.

Cada elemento de la estructura se llamará *contenedor* y constará de un identificador que servirá para indicar a qué conjunto pertenece un elemento, también tendrá dos conjuntos representados por un árbol con apuntadores al padre de cada nodo, al igual que en la estructura original de conjuntos disjuntos. Por simplicidad a estos conjuntos se les llamará *rojo* y *azul*. Conceptualmente estos contenedores se ven como el mostrado en la figura A.1.

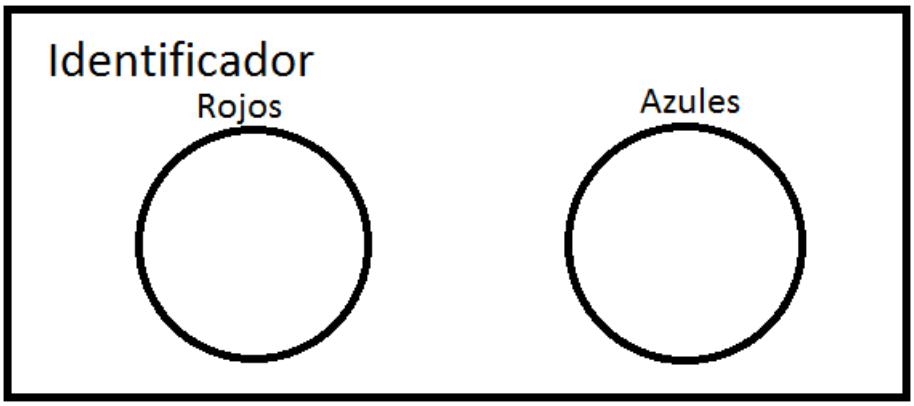


Figura A.1: Estructura contenedora para conjuntos bicoloreados.

Las operaciones definidas para esta estructura y su implementación se muestran a continuación, para mantener los identificadores de los contenedores se utiliza una variable global llamada `value` e inicializada en 0. También se utiliza un arreglo global de nodos llamado `nodes`. Este arreglo guarda en la posición i el nodo correspondiente al elemento i.

Cabe señalar que esta implementación es para propósitos ilustrativos y para enfatizar que las operaciones de esta nueva estructura están basadas en las originales y por lo tanto, mantienen su complejidad.

***make_set*:**

```
cotainer make_set(int elem){
  node n = new node(elem)
  n.set_rank(0)
  n.set_parent(n)
  container c = new container(value++, n, null)
  n.set_cointainer(c)
  nodes[elem] = n
  return c
}
```

Esta operación crea un nodo que contiene al elemento deseado y lo introduce en un nuevo contenedor, este contenedor es el resultado de la operación. El constructor del contenedor toma tres argumentos: el identificador, el nodo raíz de los rojos y el nodo raíz de los azules; en este caso, el último es `null`.

***find*:**

```
node find_set(int elem){
  n = nodes[elem]
  if n.padre != n
```

```
    n.set_parent(find_set(n.parent.elem))
        return n.parent
}
```

Si se observa el código para la operación `find` común se notará que este código es muy similar, de hecho implementa también la heurística de compresión de caminos de la misma manera, por lo que el tiempo en el peor caso es de $O(\log(n))$pero en el tiempo amortizado es $O(\alpha(n))$ pues se sigue valiendo el análisis de [8].

***union*:**
Esta es la operación más delicada y confusa por lo que se explicará con detalle antes de mostrar el código. Primero hay que considerar que se utilizará la heurística de unión por rangos, en donde se comparan los rangos de los árboles a unir y el que tiene menor rango se une al de mayor rango. En caso de empate se escoge uno de los dos y ese se une al otro, aumentando el rango del otro en 1.

Además de implementar la heurística se debe considerar que en el algoritmo se dan dos tipos de uniones: aquellas en donde los elementos que ocasionan la unión tienen colores contrarios y aquellas donde tienen el mismo color. En el primer caso, la unión debe ser entre los rojos y azules de ambos uniendos para formar el nuevo contenedor.

En el segundo, se deben unir de manera cruzada, es decir, un rojo con un azul y el otro azul con el otro rojo. Sin embargo, debido a la heurística, pudiera darse el caso de que ambos conjuntos azules (o rojos) tienen mayor rango que los otros, en este caso, se formaría un contenedor con ambos representantes del mismo color, por lo que uno de ellos debe cambiarse de color para formar un contenedor correcto. Para mantener el color actual de los representantes se utiliza otro arreglo global llamado `colors`, donde la posición i almacena el color actual del elemento i. Al estar en esta situación es necesario modificar el color de uno de los dos representantes (suponga, ambos rojos) para que cada uno quede de un color distinto. Esta modificación se hace solo en el arreglo de colores y no involucra a los nodos en sí.

Gracias al arreglo extra de colores, es posible "recolorear" componentes completas durante la unión en tiempo constante, preservando así la eficiencia de la operación.

A continuación se escribe un bosquejo con algunos de los casos de la unión, cabe mencionar que la mayoría de casos son análogos por lo que solo se muestra un caso de unión sencilla y otro de unión cruzada.

```
container union(int x, int y){
  repr_x = find_set(x)
  repr_y = find_set(y)
  container_x = representative_x.container
  container_y = representative_y.container
  rank_x_left = container_x.left.rank
  rank_y_left = container_y.left.rank
  rank_x_right = container_x.right.rank
  rank_y_right = container_y.right.rank
  //inician los distintos casos...
```

```
  if colors[repr_x.value] != colors[repr_y.value] //union sencilla
    if rank_x_left > rank_y_left
          container_y.left.parent = cointainer_x.left
          winner_left = cointainer_x.left
        else
          container_x.left.parent = cointainer_y.left
          winner_left = cointainer_y.left
          if rank_x_left == rank_y_left
            rank_y_left +=1
        .
        .//se procede a asignar el lado derecho
        .
        container c = new container(val++,winner_left,winner_right)
  else //union cruzada
    if rank_x_left > rank_y_right
      container_y.right.parent = cointainer_x.left
      winner_left = cointainer_x.left
    else
      container_x.left.parent = cointainer_y.right
      winner_left = cointainer_y.right
      if rank_x_left == rank_y_right
        rank_y_right +=1
    colors[winner_left.value] = 0
    .
    .//se procede a asignar el lado derecho
    .// y poner su color en 1
    container c = new container(val++,winner_left,winner_right)
  return c
}
```

A partir del código se observa que se invoca dos veces a la función `find_set` pero el resto de las operaciones se hacen en $O(1)$, por lo que la unión toma el mismo tiempo que la operación de búsqueda, que toma, en el peor caso, $O(\log(n))$.

Bibliografía

[1] Alon, Noga, Yossi Matias y Mario Szegedy: *The space complexity of approximating the frequency moments*. En *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, páginas 20–29. ACM, 1996.

[2] Alon, Noga, Raphael Yuster y Uri Zwick: *Finding and counting given length cycles*. Algorithmica, 17(3):209–223, 1997.

[3] Bar-Yossef, Ziv, Ravi Kumar y D Sivakumar: *Reductions in streaming algorithms, with an application to counting triangles in graphs*. En *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, páginas 623–632. Society for Industrial and Applied Mathematics, 2002.

[4] Baswana, Surender: *Streaming algorithm for graph spanners—single pass and constant processing time per edge*. Information Processing Letters, 106(3):110–114, 2008.

[5] Blum, Avrim, John Hopcroft y Ravindran Kannan: *Foundations of Data Science*. `https://www.cs.cornell.edu/jeh/book.pdf`, Accedido 17-03-2016.

[6] Bollobás, Béla: *Extremal graph theory*. Courier Corporation, 2004.

[7] Braverman, Vladimir, Rafail Ostrovsky y Dan Vilenchik: *How hard is counting triangles in the streaming model?* En *International Colloquium on Automata, Languages, and Programming*, páginas 244–254. Springer, 2013.

[8] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest y Clifford Stein: *Introduction to Algorithms*. The MIT Press, tercera edición, 2009.

[9] DeWitt, David J, Jeffrey F Naughton, Donovan A Schneider y Srinivasan Seshadri: *Practical skew handling in parallel joins*. University of Wisconsin-Madison. Computer Sciences Department, 1992.

[10] Edmonds, Jack: *Matroids and the greedy algorithm*. Mathematical programming, 1(1):127–136, 1971.

[11] Feigenbaum, Joan, Sampath Kannan, Andrew McGregor, Siddharth Suri y Jian Zhang: *On graph problems in a semi-streaming model*. Theoretical Computer Science, 348(2-3):207–216, 2005.

[12] Flajolet, Philippe y G Nigel Martin: *Probabilistic counting algorithms for data base applications*. Journal of computer and system sciences, 31(2):182–209, 1985.

[13] Karger, David R: *Random sampling in cut, flow, and network design problems*. Mathematics of Operations Research, 24(2):383–413, 1999.

[14] Kingsford, Carl: *CMSC 451: Maximum Bipartite Matching*. `http://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/matching.pdf`, Accedido 06-02-2017.

[15] Kleinberg, Jon y Éva Tardos: *Union Find Course Notes*. `http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind-2x2.pdf`, Accedido 17-11-2016.

[16] Kleinberg, Jon y Éva Tardos: *Union Find Slides*. http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf, Accedido 17-01-2017.

[17] Kleinberg, Jon y Éva Tardos: *Algorithm Design*. Addison Wesley, primera edición, 2006.

[18] Kruskal, Joseph B: *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical society, 7(1):48–50, 1956.

[19] Matula, David W.: *A Linear Time (2+ϵ)-Approximation Algorithm for Edge Connectivity*. En *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, páginas 500–504, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics, ISBN 0-89871-313-7.

[20] McGregor, Andrew: *Graph stream algorithms: a survey*. ACM SIGMOD Record, 43(1):9–20, 2014.

[21] Mendenhall, William, Robert J. Beaver, Barbara M. Beaver y Jorge Humberto Romo Muñoz: *Introducción a la probabilidad y estadística*. Australia ; México, D.F. : Cengage Learning, c2010, 2010. http://pbidi.unam.mx:8080/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat02025a&AN=lib.MX001001480713&lang=es&site=eds-live.

[22] Mucha, Marcin y Piotr Sankowski: *Maximum matchings via Gaussian elimination*. En *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, páginas 248–255. IEEE, 2004.

[23] Reinhard, Diestel: *Graph Theory*. Springer, tercera edición, 2005.

[24] Tarjan, R.: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983. http://epubs.siam.org/doi/abs/10.1137/1.9781611970265.

[25] Zelke, Mariano: *Intractability of min-and max-cut in streaming graphs*. Information Processing Letters, 111(3):145–150, 2011.