Final Matemática Discreta II
TEÓRICO
GRUPO A

Agustín M. Domínguez

Julio 2024

Índice

1 Contexto **2**
- 1.1 Introducción . . . 2
- 1.2 Enunciado . . . 2
- 1.3 Grupo . . . 2

2 Pregunta 1: Complejidad de Edmonds-Karp **3**
- 2.1 Enunciado detallado . . . 3
- 2.2 Conocimiento presupuesto para la resolución . . . 3
  - 2.2.1 Definiciones para llegar a **E-K** . . . 3
  - 2.2.2 Algoritmo de Edmonds-Karp . . . 5
  - 2.2.3 Distancia entre iteraciones de E-K . . . 5
- 2.3 Resolución . . . 6

3 Pregunta 2: Probar $d_k(x) \leq d_{k+1}(x) \quad \forall x \in V$ **8**
- 3.1 Enunciado detallado . . . 8
- 3.2 Resolución . . . 8

4 Pregunta 4: Complejidad de Dinic **11**
- 4.1 Enunciado detallado . . . 11
- 4.2 Resolución . . . 11

5 Pregunta 5: Complejidad de Wave **16**
- 5.1 Enunciado detallado . . . 16
- 5.2 Resolución . . . 16

Contexto

1.1 Introducción

Este apunte fue construido para final de la materia **Matemática Discreta II** de la **Facultad de Matemática Física y Computación** en las fechas de Julio/Agosto del año 2024.

Una nota para otros estudiantes: El objetivo de las demostraciones de **este apunte** es ayudar a entenderlas, no ayudar a memorizarlas. Están escritas con todos los pasos explícitos y estrictos, a veces al punto de redundancia y obviedad, *a propósito*. La idea es que el lector tenga que hacer la menor cantidad de saltos de lógica posibles para entender la demostración. La resolución que se haga en el final probablemente pueda tener algunos pasos menos explícitos, pero no es el marco de este apunte.

Fue declarado por la cátedra que estas preguntas van a formar el Teórico del final. A continuación se repite dicho enunciado.

1.2 Enunciado

La parte teórica del final consistirá de 3 preguntas tomadas de esta lista, mas una pregunta extra del tema de Milagro.

Para aprobar el teórico hay que obtener 40% del puntaje EN CADA pregunta (de esta lista). Estos teoremas son para Julio/Agosto 2024, excepto algunos que estan marcado que solo se tomaran a partir de Diciembre.

Ademas de estos teoremas, en Diciembre/Febrero/Marzo pueden agregarse incluso otros, pero si lo hacemos se indicara en la pagina de la materia. Si no se dice nada, estos son los que valen.

De los **tres** teoremas que se tomarán de esta lista, **uno será de los seis primeros, otro de los tres ultimos y el otro del resto**.

1.3 Grupo

Este apunte tiene las respuestas de las opciones para la **primera pregunta**, desde ahora llamadas el **Grupo A**.

Pregunta 1: Complejidad de Edmonds-Karp

2.1 Enunciado detallado

¿Cuál es la complejidad del algoritmo de Edmonds-Karp? Probarlo.

Nota: en la prueba se definen unas distancias, y se prueba que esas distancias no disminuyen en pasos sucesivos de **E-K**. Ud. puede usar esto sin necesidad de probarlo

OBSERVACION PARA LOS PRIMEROS 3 EJERCICIOS: Como explicamos en el teorico, casi toda la prueba de estos teoremas valdrian para pej Ford-Fulkerson, excepto por unas partes claves de la prueba donde se usa una propiedad especifica de Edmonds-Karp. Si ustedes escriben toda la prueba sin destacar que propiedad de Edmonds-Karp se usa y donde, estan desaprobados.

2.2 Conocimiento presupuesto para la resolución

2.2.1 Definiciones para llegar a E-K

Grafo dirigido

Es un par (V, E) con $E \subseteq V \times V$. A diferencia de la definición normal de un grafo, el orden de los vértices si importa, y E es el conjunto de estos vértices con orden.

Network

Es un triple (V, E, C) con (V, E) grafo dirigido y C es una función $C : E \to \mathbb{R}_{\geq 0}$. A esta función se le dice la *capacidad de flujo* de cada lado.

Flujo y Flujo Maximal

Dado una network $N = (V, E, C)$ y vertices s y t, un Flujo s a t es una función de los lados con:

1. $0 \leq f(\overrightarrow{xy}) \leq C(\overrightarrow{xy}) \quad \forall \overrightarrow{xy} \in E$
2. $\mathrm{IN}_f(x) = \mathrm{OUT}_f(x) \quad \forall x \neq s, t$
3. $\mathrm{IN}_f(s) = 0 = \mathrm{OUT}_f(t)$

El **valor** de un flujo es: $v(f) = \mathrm{OUT}_f(s) - \mathrm{IN}_f(s)$

En la mayoria de los flujos s no tiene vecinos hacia atras, por lo que $v(f) = \mathrm{OUT}_f(s)$

Un flujo f es **maximal** si $v(g) \leq v(f)\ \forall g$ flujo de s a t

Problema Maxflow

Dado un network, hallar *un* flujo maximal.

Camino Aumentante

Dado un flujo f en un network, un camino aumentante o f-camino aumentante es una sucesión de vértices $x_0, x_1, \ldots, x_r$ con $x_0 = s,\ x_r = t$ donde $\forall i < r$ ocurre una **y solo una** de las siguientes propiedades:

- $\overrightarrow{x_i x_{i+1}} \in E \ \wedge\ f(\overrightarrow{x_i x_{i+1}}) < C(\overrightarrow{x_i x_{i+1}})$ llamados lados *forward*
- $\overrightarrow{x_{i+1} x_i} \in E \ \wedge\ f(\overrightarrow{x_{i+1} x_i}) > 0$ llamados lados *backward*

Algoritmo de Ford-Fulkerson (F-F)

Es un algoritmo que resuelve el problema **maxflow**.

Empezar con $f = 0$ y hacer:

1. Buscar un f-camino aumentante $s = x_0, x_1, \ldots, x_r = t$ (en el algoritmo no se especifica cómo buscar este camino aumentante)
2. Definir: $\varepsilon_i = \begin{cases} C(\overrightarrow{x_i x_{i+1}}) - f(\overrightarrow{x_i x_{i+1}}) & \text{en lados forward} \\ f(\overrightarrow{x_{i+1} x_i}) & \text{en lados backward} \end{cases}$

   (es decir lo máximo que pueda enviar en los lados forward y lo máximo que puedo devolver en los lados backward)
3. Tomar $\varepsilon = \min\{\varepsilon_i \mid i = 0, \ldots, r-1\}$
4. Cambiar f a lo largo del camino:

   $f(\overrightarrow{x_i x_{i+1}}) = \overrightarrow{x_i x_{i+1}} + \varepsilon$

   $f(\overrightarrow{x_{i+1} x_i}) = \overrightarrow{x_{i+1} x_i} - \varepsilon$

Cuando no exista más camino aumentante s a t entonces el algoritmo se detiene.

La ventaja de este algoritmo es que encuentra flujos maximales pero la desventaja es que **no garantiza que el algoritmo termine**.

2.2.2 Algoritmo de Edmonds-Karp

El algoritmo de Edmonds-Karp es una variante del Algoritmo Ford-Fulkerson que encuentra **caminos aumentantes** pero de una forma que garantiza que siempre termina.

La diferencia que tiene con **FF** es su definición de cómo buscar caminos aumentantes. La forma de hacer esto es usando **BFS** *Breadth First Search* partiendo de s tratando de llegar a t. La idea es siempre buscar el camino aumentante de **menor longitud posible**, y que cuando la búsquedas no llegue a t se sepa que no existe ningún camino aumentante disponible.

En la búsqueda se consideran como vecino de cada vértice primeros los vértices que tengan lados *forward* y luego los que tengan lados *backward*.

2.2.3 Distancia entre iteraciones de E-K

Dados x, z vértices y f flujo, se define la siguiente función: $d_f(x, z)$ como la longitud del menor **f-CA** (camino aumentante) entre x y z, si es que existe tal camino, ∞ si no existe, o 0 si $x = z$.

Sean $f_0 = 0, f_1, f_2, \ldots$ los flujos producidos por el algoritmo **Edmonds-Karp**.

Denotamos $d_i(x) = d_{f_i}(s, x)$

y denotamos $b_i(x) = d_{f_i}(x, t)$

Propiedad de la función d_i:

$$d_i(t) = d_i(x) + b_i(x)$$

Para todo x que pertenezca al camino aumentante encontrado por **E-K**

Por enunciado, podemos asumir las siguientes propiedades:

$$d_i(x) \leq d_{i+1}(x) \quad \forall x \in V \quad \wedge \quad b_i(x) \leq b_{i+1}(x) \quad \forall x \in V$$

Generalizando lo anterior obtenemos:

$$i \leq j \implies d_i(x) \leq d_j(x) \quad \forall x \in V$$
$$i \leq j \implies b_i(x) \leq b_j(x) \quad \forall x \in V$$

2.3 Resolución

La complejidad de Edmonds-Karp es $\mathcal{O}(nm^2)$

Asumimos sin demostrar las siguientes propiedades:

$$i \leq j \implies d_i(x) \leq d_j(x) \quad \forall x \in V \quad \textbf{(A)}$$
$$i \leq j \implies b_i(x) \leq b_j(x) \quad \forall x \in V \quad \textbf{(B)}$$

Dem. Dados x, z vértices y f flujo

Diremos que un lado se vuelve *crítico* cuando al pasar de f_k a f_{k+1} se **satura o se vacía**.

Para probar la complejidad veremos cuántas veces un lado se puede volver crítico durante la ejecución del algoritmo.

Supongamos que un lado de vértices x y z se vuelve crítico en el paso k y vuelve a ser crítico en el paso j

$s \dots xz \dots t$ crítico en el paso $f_k \to f_{k+1}$

Como el algoritmo es **E-K**, que encuentra los caminos aumentantes más cortos:

$$\implies d_k(z) = d_k(x) + 1 \quad \text{(C)}$$

Nota: No sabemos si el lado del flujo es $\overrightarrow{xz}$ o $\overrightarrow{zx}$. Por lo que vamos a tener que considerar ambos casos.

Primer caso: Si $\overrightarrow{xz} \in E$. Eso implica que se volvió crítico porque se saturó. Entonces para volver a ser crítico en el paso j hay dos opciones: O bien se vacía, o primero devuelve parte del flujo y luego se satura.

En cualquier caso se cumple que $\exists\ l$ paso $\mid k < l \leq j$ donde z devuelve flujo a x.

Es decir que existe iteración l tal que: $s \dots \overleftarrow{zx} \dots t$ y nuevamente como el algoritmo es **E-K** $\implies d_l(x) = d_l(z) + 1$

Segundo caso: Si $\overrightarrow{zx} \in E$ entonces este lado se volvió crítico porque se vació. Por el mismo tipo de análisis que el caso anterior, para volver a ser crítico debe saturarse o haberse vuelto a vaciar, por lo cual en algun paso posterior debió haber enviado flujo:

$$\therefore \exists \text{ paso } l \mid k < l \leq j \text{ tal que } s \dots zx \dots t \implies d_l(x) = d_l(z) + 1 \quad \text{(D)}$$

En ambos casos vemos que se cumple **(D)**.

Ahora utilizaremos todas estas propiedades:

Por **(D)** tenemos que $l \leq j$

$$l \leq j \overset{(A)}{\implies} d_j(t) \geq d_l(t)$$

Por propiedad de la función d tenemos que: $d_l(t) = d_l(x) + b_l(x)$

$$\implies d_j(t) \geq d_l(x) + b_l(x) \overset{(D)}{=} d_l(z) + 1 + b_l(x) \overset{(B)}{\geq} d_k(z) + 1 + b_l(x)$$

$$\overset{(B)}{\implies} d_j(t) \geq d_k(z) + 1 + b_k(x) \overset{(C)}{=} (d_k(x) + 1) + 1 + b_k(x)$$

$$\implies d_j(t) \geq d_k(x) + b_k(x) + 2$$

Nuevamente, por propiedad de la función d, tenemos que: $d_k(t) = d_k(x) + b_k(x)$

$$\implies d_j(t) \geq d_k(t) + 2$$

Conclusión: Una vez que un lado se vuelve crítico, solo puede volver a ser crítico si la distancia entre s y t aumenta por lo menos en 2.

Recordemos que notamos n como la cantidad de vértices y m la cantidad de lados.

Como la búsqueda de camino aumentante en **E-K** itera sobre los vértices y no los repite, el camino más largo posible en una corrida de **E-K** es n. Como cada vez que un lado se vuelve crítico la distancia del camino aumenta en por lo menos 2, entonces **un lado se puede volver crítico a lo sumo** $\frac{n-1}{2} = \mathcal{O}(n)$ **veces**.

Ahora juntamos estas conclusiones para llegar al orden del algoritmo:

1. Para pasar de f_i a f_{i+1} al menos un lado se vuelve crítico
2. Hay m lados
3. Cada lado se puede volver crítico $\mathcal{O}(n)$ veces

$\therefore$ el número total de pasos (iteraciones de E-K) es $\mathcal{O}(mn)$

En cada iteración hay que encontrar un **Camino-Aumentante Mínimo**, que en **E-K** se hace con **BFS** *(Breadth First Search)*, cuya complejidad es $\mathcal{O}(m)$.

$$\therefore \quad \text{Complejidad total es:} \quad \mathcal{O}(nm^2)$$

□

Pregunta 2: Probar $d_k(x) \leq d_{k+1}(x) \quad \forall x \in V$

3.1 Enunciado detallado

Probar que si, dados vértices x, z y flujo f definimos a la distancia entre x y z relativa a f como la longitud del menor **f-camino aumentante** entre x y z, si es que existe tal camino, o infinito si no existe o 0 si $x = z$, denotandola por $d_f(x, z)$, y definimos $d_k(x) = d_{f_k}(s, x)$, donde f_k es el **k-ésimo** flujo en una corrida de Edmonds-Karp, entonces $d_k(x) \leq d_{k+1}(x)$.

OBSERVACION PARA LOS PRIMEROS 3 EJERCICIOS: Como explicamos en el teorico, casi toda la prueba de estos teoremas valdrian para pej Ford-Fulkerson, excepto por unas partes claves de la prueba donde se usa una propiedad especifica de Edmonds-Karp. Si ustedes escriben toda la prueba sin destacar que propiedad de Edmonds-Karp se usa y donde, estan desaprobados.

3.2 Resolución

Dem. Sea $A = \{y : \ d_{k+1}(y) < d_k(y)\}$

Si demonstramos que A es vacío, entonces completamos la demonstración

Suposición Inicial: $A \neq \varnothing$.

Sea x el vértice de A con menos distancia a s en la iteración $k+1$. Es decir:

Sea $x \in A \quad | \quad d_{k+1}(x) \leq d_{k+1}(y) \quad \forall \ y \in A \quad (1)$

Como $x \in A$ entonces se cumple que $d_{k+1}(x) < d_k(x)$

En particular: $d_{k+1} < \infty \implies \exists \ f_{k+1}$-CA entre s y x **(P1)**.

y por definición $d_k(s) = 0 = d_{k+1}(s) \implies s \notin A \implies s \neq x$ **(P2)**

$$\textbf{(P1)} \wedge \textbf{(P2)} \implies \exists \ z \in f_{k+1}\text{-CA tal que } z \text{ esta inmediatamente anterior a } x$$

Es decir el camino aumentante es: $s \dots zx \dots t \quad (f_{k+1} - CA)$

Nota: z puede ser s, y no sabemos si $\overrightarrow{zx} \in E$ o si $\overrightarrow{xz} \in E$

Como el camino aumentante es el resultado de una iteración de **E-K**, es de longitud mínima. Por lo tanto se cumple:

$$d_{k+1}(x) = d_{k+1}(z) + 1 \quad (2)$$

Luego:

$$(2) \implies d_{k+1}(z) < d_{k+1}(x) \overset{(1)}{\implies} z \notin A$$
$$\implies d_k(z) \leq d_{k+1}(z) \quad (3)$$

Como $d_{k+1}(z) < d_{k+1}(x) < \infty \overset{(3)}{\implies} d_k(z) < \infty$

Por definición de la función d:

$$d_k(z) < \infty \implies \exists f_k\text{-CA entre } s \text{ y } z$$

De los caminos existentes tomo el de longitud mínima: $f_k = s \dots z \quad (4)$

Como no sabemos si $\overrightarrow{zx} \in E$ o si $\overrightarrow{xz} \in E$, veo ambos casos:

Suponemos $\overrightarrow{zx} \in E$. Entonces en el camino aumentante de $k+1$ es un lado forward:

$$\implies f_{k+1}(\overrightarrow{zx}) < C(\overrightarrow{zx})$$

Hay dos opciones, que en la iteración f_k el lado $\overrightarrow{zx}$ haya estado saturado o no.

Caso 1: **Suponemos** $\overrightarrow{zx}$ estaba saturado en f_k

Como estaba saturado en f_k pero se usó *forward* en f_{k+1}, entonces en f_k se tuvo que usarse *backward*

Por lo que el $f_k - CA$ debio ser: $s \dots \overleftarrow{xz} \dots t$

Como el algoritmo utilizado es **E-K**, este camino es de longitud mínima, entonces se cumple que:

$$d_k(z) = d_k(x) + 1 \quad (5)$$
$$\implies d_k(x) = d_k(z) - 1 \overset{(3)}{\leq} d_{k+1}(z) - 1 \overset{(2)}{=} d_{k+1}(x) - 1 - 1$$
$$\implies d_k(x) \leq d_{k+1}(x) - 2 \overset{x \in A}{<} d_k(x) - 2$$
$$\implies d_k(x) < d_k(x) - 2 \implies 0 < -2 \quad \textbf{ABS}$$

Caso 2: **Suponemos** $\overrightarrow{zx}$ **no** estaba saturado en f_k

$$\implies f_k(\overrightarrow{zx}) < C(\overrightarrow{zx})$$

Por (4) tenemos un f_k-CA **mínimo** entre s y z: $s \dots z$. Y como $\overrightarrow{zx}$ no estaba saturado:

$\exists f_k$-CA : $\quad s \dots zx$ (no necesariamente mínimo)

Este camino aumentante **no es necesariamente mínimo** pero su longitud es $d_k(z) + 1 < \infty$

Luego por definición de la función d, se cumple que $d_k(x) \leq d_k(z) + 1 \quad (6)$

$$d_k(x) \leq d_k(z) + 1 \overset{(3)}{\leq} d_{k+1}(z) + 1 \overset{(2)}{=} d_{k+1}(x) \overset{x \in A}{>} d_k(x)$$
$$\implies d_k(x) < d_k(x) \implies 0 < 0 \quad \textbf{ABS}$$

En ambos casos llegamos a un **Absurdo**. Por lo que concluimos que $\overrightarrow{zx} \notin E$

$$\implies \overrightarrow{xz} \in E \implies zx \text{ era } \textit{backward} \text{ en la iteración } k+1$$

Nuevamente tenemos dos casos: En la iteración f_k el lado $\overrightarrow{xz}$ estaba vacío o no.

Caso 1: **Suponemos** $\overrightarrow{xz}$ estaba vacío en f_k

Como en el flujo f_{k+1} este lado se usó como *backward*, entonces necesariamente en la iteración k tuvo que haberse usado *forward*, resultando en: f_k-CA: $s \dots xz \dots t$

Por estar usando el algoritmo **E-K**, entonces este Camino Aumentante debe ser mínimo

$$\implies d_k(z) = d_k(x) + 1 \quad (5)$$

Volvemos a llegar a la misma expresión **(5)** del caso anterior, y con el mismo progreso llegamos a un **ABS**.

Caso 2: **Suponemos** $\overrightarrow{xz}$ **no** estaba vacío en f_k, entonces $\implies f_k(\overrightarrow{xz}) > 0$

Igual que en el anterior Caso 2, podemos extender $s \dots \overleftarrow{zx}$ como un Camino Aumentante de longitud $d_k(z) + 1$, y volviendo a aplicar la definición de la función d llegamos a:

$d_k(x) \leq d_k(z) + 1$ que es **(6)**.

Por la misma lógica llegamos a un **ABS**

En todos los casos llegamos a un **Absurdo** por lo que nuestra suposición inicial **es incorrecta**. $\implies A = \varnothing$

$$\therefore \quad d_k(x) \leq d_{k+1}(x) \quad \forall x \in V$$

□

Pregunta 4: Complejidad de Dinic

4.1 Enunciado detallado

¿Cuál es la complejidad del algoritmo de Dinic? Probarla en ambas versiones: Dinitz original y Dinic-Even. (no hace falta probar que la distancia en networks auxiliarias sucesivos aumenta)

4.2 Resolución

La complejidad del algoritmo Dinic es $\mathcal{O}(mn^2)$, tanto para la versión Dinitz original como la versión Dinic-Even

Dem. En ambos algoritmos el proceso básico es construir Networks Auxiliares que tienen los caminos aumentantes, y luego encontrar y saturar dichos caminos aumentantes de la NA en el flujo original hasta que la NA no tenga más caminos, es decir encontrar un flujo bloqueante en dicha NA.

Por lo tanto, la complejidad es igual a:

$$\textbf{NNA} \times (\textbf{CNA} + \textbf{FBLK})$$

- NNA = N° de Networks Auxiliares usados
- CNA = Complejidad de crear un Network Auxiliar
- FBLK = Complejidad de hallar un flujo bloqueante en ese Network Auxiliar

Primero encontremos el primer número probando que **en cualquier corrida *'tipo'* Dinic hay a lo sumo n Networks Auxiliares.**

Por el enunciado, podemos asumir sin demostrar que la distancia entre NA sucesivos aumenta. Dado esto, quiere decir que la distancia entre s y t aumenta en al menos 1 por cada NA. Salvo el último NA, donde la distancia es infinita, dicha distancia debe ser un número natural entre 1 y $n-1$, por lo tanto hay a lo sumo n NA.

La complejidad entonces es:

$$\mathcal{O}(n) \times (\textbf{CNA} + \textbf{FBLK})$$

Veamos ahora el segundo término. Como ambas complejidades se suman, la complejidad final va a ser esencialmente la mayor de ellas. Primero sabemos que CNA es $\mathcal{O}(m)$ ya que el método para crear el NA es **BFS**.

Luego si consideramos FBLK, para hallar un flujo bloqueante en un NA hay que revisar todos los lados para saber si el flujo es bloqueante, por lo que parece imposible que su complejidad sea *menor* a $\mathcal{O}(m)$.

Hasta que se descubra un método que permita esto, vamos a asumir que FBLK $\geq$ CNA por lo que la complejidad de CNA es 'absorbida' por FBLK, y queda:

$$\mathcal{O}(n) \times \textbf{FBLK}$$

Ahora demostraremos la complejidad de encontrar un flujo bloqueante en una NA (FBLK).

Para esto tendremos que ver las diferencias entre los algoritmos **Dinitz Original** y **Dinic-Even**, y la mejor forma de ver las diferencias es anotando sus respectivos pseudocódigos:

```
DINITZ_ORIGINAL(NA)
    g = 0
    stop = false
    WHILE (¬stop)
        p = [s]
        x = s
        WHILE (x ≠ t)
            AVANZAR(x, p, NA)
        IF (x == t)
            INCREMENTAR(p, g)
            PODAR(NA, stop)
    RETURN(g)
```

```
DINIC_EVEN(NA)
    g = 0
    stop = false
    WHILE (¬stop)
        p = [s]
        x = s
        WHILE (x ≠ t AND ¬stop)
            IF (Γ⁺(x) ≠ ∅):
                AVANZAR(x, p, NA)
            ELSE IF (x ≠ s):
                RETROCEDER(x, p)
            ELSE:
                stop = true
        IF (x == t)
            INCREMENTAR(p, g)
    RETURN(g)
```

Usando las siguientes funciones auxiliares:

```
AVANZAR(x: Vert, p: Camino, NA):
    Tomar y ∈ E | Γ⁺(x) // y vecino de x
    Agregar y al camino p
    x = y
```

```
INCREMENTAR(p: Camino, g: Num):
    Recorrer el camino p para calcular el ε
    Aumentar g en ε a lo largo de p
    Borrar los lados saturados


RETROCEDER(x: Vert, p: Camino, NA)
    y = vertice anterior de x en el camino p
    Borrar yx⃗ del NA
    Borrar x del camino
    x = y


PODAR(NA, stop: Bool)
    Itero los niveles desde t a s:
        En cada vertice sino tiene vecinos hacia adelante:
            Borro sus lados hacia atras y el vertice
```

Veamos la complejidad de **Dinic-Even**

Si denotamos AVANZAR como 'A', RETROCEDER como 'R', y luego INCREMENTAR por 'I', tomando la inicialización como parte de I, vemos que el algoritmo es una sucesión de As, R, Is. Podemos dividir esta sucesión de letras en 'palabras' de la forma $AA \dots AX$ donde X es R o I.

La complejidad del algoritmo entonces es la cantidad de palabras por la complejidad de cada palabra.

Veamos primero cuantas palabras hay.

Tenemos las siguientes propiedades:

- Cada palabra termina en R o I
- R borra el lado por el cual retrocede
- I manda flujo por un camino en el cual satura al menos un lado, y borra todos los lados saturados $\implies$ Borra al menos un lado

$\implies$ Al final de cada palabra se borra al menos un lado.

$\therefore$ La cantidad de palabras es a lo sumo m.

Veamos ahora la complejidad de las palabras:

Tanto el orden de AVANZAR como el de RETROCEDER es $\mathcal{O}(1)$ ya que no iteran sobre nada y realizan asignaciones simples.

En INCREMENTAR se recorre el camino encontrado, que como vimos puede ser como máximo largo n, por lo que su complejidad es $\mathcal{O}(n)$

Si una palabra $AA \dots AX$, tiene r 'A's, entonces su complejidad va a ser:

$$\begin{cases} \mathcal{O}(r+1) & \text{si } X = R \\ \mathcal{O}(r+n) & \text{si } X = I \end{cases}$$

Como en cada avance el camino aumenta en un nivel de la NA, y puede haber a lo sumo n niveles, entonces $r \leq n$

$$\implies \begin{cases} \mathcal{O}(n+1) = \mathcal{O}(n) & \text{si } X = R \\ \mathcal{O}(n+n) = \mathcal{O}(n) & \text{si } X = I \end{cases}$$

$\therefore$ La complejidad de la palabra es $\mathcal{O}(n)$

En resumen, **hay a lo sumo m palabras y la complejidad de cada palabra es $\mathcal{O}(n)$**

$\implies$ La complejidad de **FBLK** es $\mathcal{O}(mn)$

$\therefore$ La complejidad total de Dinic-Even es $\mathcal{O}(n) \times \mathcal{O}(mn) = \mathcal{O}(mn^2)$

Ahora veamos la complejidad de **Dinitz Original**.

El análisis es similar, excepto que las palabras no tienen 'R's, pero si tiene PODAR que denotaremos como 'P'.

Las palabra siempre tienen la forma $AA \dots IP$

Con el mismo análisis sabemos que:

1. Cada palabra tiene un I que borra al menos un lado
2. Cada 'A' del camino aumenta en uno el nivel de la NA
3. Hay un máximo de n niveles en la NA

1. $\implies$ La cantidad de palabras es $\mathcal{O}(m)$

2. $\wedge$ 3. $\implies$ Hay $\mathcal{O}(n)$ 'A's en cada palabra.

$\implies$ La complejidad de cada $AA \dots I$ es $\mathcal{O}(n+n) = \mathcal{O}(n)$

$\therefore$ La complejidad total del conjunto de $AA \dots I$ es $\mathcal{O}(nm)$

Queda ver la complejidad de los 'P'.

cada P está divido en dos partes:

1. Iterar sobre vertices. Sea esta parte PV
2. Bajo cierta condición, borra todos los lados de entrada. Sea esta parte B(x)

La complejidad de cada PV es $\mathcal{O}(n)$, ya que chequea todos los vértices.

Hay un PV por cada palabra, asíque hay a lo sumo m en total

La complejidad de todos ellos es $\mathcal{O}(nm)$.

Ahora veamos la complejidad de los B(x):

1. La complejidad de *un* B(x) es la cantidad de lados borrados.
2. Si se llama B con un vértice, no se vuelve a llamar más con el mismo vértice.

   $\implies$ Independientemente de cuantos B(x) hay luego de *un* PV, sabemos que al final del algoritmo habrá a lo sumo un B(x) por cada vértice.

1. $\wedge$ 2. $\implies$ Al final del algoritmo a lo sumo se borraron todos los lados, por lo que la complejidad de B(x) en **conjunto** es $\mathcal{O}(m)$

En resumen:

- La complejidad de los $AA \dots I$ es $\mathcal{O}(nm)$
- La complejidad de los 'PV's es $\mathcal{O}(nm)$
- La complejidad de los B(x) en conjunto es $\mathcal{O}(m)$

$$\implies \text{FBLK} = \mathcal{O}(nm) + \mathcal{O}(nm) + \mathcal{O}(m) = \mathcal{O}(nm)$$

$\therefore$ La complejidad total de Dinitz Original es $\mathcal{O}(n) \times \mathcal{O}(mn) = \mathcal{O}(mn^2)$

□

Pregunta 5: Complejidad de Wave

5.1 Enunciado detallado

¿Cuál es la complejidad del algoritmo de Wave? Probarla. (no hace falta probar que la distancia en networks auxiliares sucesivos aumenta)

5.2 Resolución

La complejidad del algoritmo de Wave es $\mathcal{O}(n^3)$

Dem. Como el algoritmo es 'tipo' Dinitz, la complejidad es: $\mathcal{O}(n) \times$ FBLK

FBLK = Complejidad de hallar flujo bloqueante

Si demostrasmos que FBLK = $\mathcal{O}(n^2)$, ya está.

Para encontrar el flujo bloqueante, Wave hace una serie de ciclos de olas hacia adelante y olas hacia atrás. En cada ola hacemos balanceos forward o backward.

Por cada balanceo sobre un vértice x revisamos una serie de lados, ya sean los $\Gamma^+(x)$ para el caso forward como el los vértices que le mandaron flujo a x para el caso backward, es decir que procesamos una serie de lados $\overrightarrow{xy}$ o $\overrightarrow{yx}$, según sea el caso.

Cada uno de estos procesamientos es $\mathcal{O}(1)$, ya que solo actualizamos el flujo y movemos un vértice de conjunto. Por lo tanto la complejidad de encontrar un flujo bloqueante con Wave es simplemente la cantidad total de estos procesamientos.

Estos procesamientos se dividen en los siguientes casos:

1. Procesamos balance forward $\overrightarrow{xy}$

   Caso A. Luego de procesarlo, el lado $\overrightarrow{xy}$ queda saturado

   Caso B. No queda saturado

2. Procesamos balance backward $\overrightarrow{yx}$

   Caso C. Luego de procesarlo, el lado $\overrightarrow{yx}$ queda vacío

Caso D. No queda vacío

Sea:

- A = Cantidad de veces que ocurre el Caso A.
- B = Cantidad de veces que ocurre el Caso B.
- C = Cantidad de veces que ocurre el Caso C.
- D = Cantidad de veces que ocurre el Caso D.

Luego **FBLK** = A + B + C + D

Una vez que un lado $\overrightarrow{xy}$ se satura, **nunca más** se vuelve a saturar, ya que para volver a saturarse y debería primero devolverle flujo a x y luego x enviarle flujo nuevamente a y, pero si y le devolvió flujo a x es porque está **bloqueado**, por lo que el algoritmo no permite que x le vuelva a mandar flujo.

De la misma forma, todo lado se puede vaciar una sola vez, ya que para vaciarse tuvo que devolver flujo, y esto solo puede pasar si el vértice que devolvió está saturado, por lo que no puede volver a recibir flujo nuevamente.

$\implies$ A y C están acotadas por la cantidad de lados es decir que son $\mathcal{O}(m)$

Ahora veamos el caso de B y D. Por cada Balanceo Forward, **a lo sumo** un procesamiento corresponde a B. Esto es por la definición del algoritmo ya que en cada balanceo la única forma que se pase a otro vecino forward de x es si el vecino anterior quedó saturado. En otras palabras: no existe el caso donde un vecino quede procesado, no saturado, y luego se pase a otro.

Por un análisis similar, en cada balanceo Backward, **a lo sumo** un lado va a ser procesado sin quedar vacío. Ese caso es cuando no hay que devolver más flujo al vecino anterior.

Es decir que B y D están ambas acotadas por la cantidad de Balanceos Forward y Backward respectivamente.

En cada ola sea forward o backward, se hace el balanceo de los vértices una vez, por lo que **en cada ola** hay a lo sumo $n-2$ Balanceos

$\therefore$ Las cantidades de B y D son $\mathcal{O}(n)$ por ola.

Falta ver la cantidad de olas:

Sabemos que si una ola forward no bloquea a ningún vértice, entonces los vértices quedan balanceados y el algoritmo termina.

$\implies$ En toda ola forward que no sea la última, al menos un vértice se bloquea.

Si un vértice se bloquea, no se vuelve a desbloquear más.

$\therefore$ La cantidad de olas forward es $\mathcal{O}(n)$

A su vez, las olas backward ocurren siempre luego de una forward, por lo que no puede haber más olas backward que forward.

$\therefore$ La cantidad de olas es $\mathcal{O}(n)$

Luego la cantidad de B y D en todo el algoritmo son $\mathcal{O}(n^2)$

$\implies$ FBLK = A + B + C + D = $\mathcal{O}(m) + \mathcal{O}(n^2) + \mathcal{O}(m) + \mathcal{O}(n^2)$

$\therefore$ FBLK = $\mathcal{O}(n^2)$

Como se dijo al inicio, esto implica que la complejidad total de Wave es $\mathcal{O}(n^3)$

□