Contents

Grafos

grafo

es un par ordenado G = (V, E) donde

V es un conjunto cualquiera.

En esta materia siempre supondremos V finito.

E es un subconjunto del conjunto de subconjuntos de 2 elementos de V.

es decir $E \subseteq \{A \subseteq V : |A| = 2\}$

Notaciones

elementos de V
se llaman **vértices** o nodos. Usaremos preferentemente el primer nombre.

elementos de E
se llaman **lados** o aristas. Usaremos preferentemente el primer nombre.

cantidad de elementos de V,
salvo que digamos otra cosa, se denotará por default como n.

cantidad de elementos de E,
salvo que digamos otra cosa, se denotará por default como m.

Un elemento {x, y} $\in$ E
será abreviado como xy.

x e y se llamarán los extremos del lado xy.

Subgrafos

Dado un grafo G = (V, E), un **subgrafo** de G es un **grafo** H = (W, F) tal que W $\subseteq$ V y F $\subseteq$ E.

Observemos que pedimos que H sea en si mismo un grafo. No cualquier par (W, F) con W $\subseteq$ V y F $\subseteq$ E será un subgrafo

Vecinos de un vértice

Dado x $\in$ V, los vértices que forman un lado con x se llaman los **vécinos** $\in$ de x.

El conjunto de vécinos se llama el

"vecindario"
y se denota por Γ (x).

Es decir Γ (x) = {y $\in$ V : xy $\in$ E}

Grado de un vértice

La cardinalidad de Γ (x) se llama el **grado** de x, y la denotaremos por d(x) (o dG(x)

WARNING:
en algunos libros se denota usando la letra griega delta: δ (x)

δ **y** Δ

El menor de todos los grados
de un grafo lo denotaremos por δ y al

mayor de todos los grados
por Δ .

$\delta = \text{Min}\{\text{d(x)} : \text{x} \in \text{V}\}$ $\text{Min}\{\text{d(x)} \in \text{V}\}$ $\Delta = \text{Max}\{\text{d(x)} : \text{x} \in \text{V}\}$

Un grafo que tenga $\delta = \Delta$ (es decir, todos los grados iguales) se llamará un

grafo regular.

o Δ -regular si queremos especificar el grado común a todos los vértices.

Cíclicos y completos

grafo cíclico
en n vértices, (n > 3) denotado por Cn, es el grafo:

$\{x_1, ..., x_n\}$ y lados $\{x_1x_2, x_2x_3, ..., x_{n-1}x_n, x_nx_1\}$).

grafo completo
en n vértices, denotado por Kn, es el grafo:

$\{x_1, ..., x_n\}$ y lados $\{x_ix_j : i, j \in \{1, 2, ..., n\}, i < j\}$)

Cn y Kn tienen ambos n vértices, pero Cn tiene n lados mientras que Kn tiene

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

lados.

Cn se llaman cíclicos porque su representación gráfica es un ciclo de n puntos.

$$d_{C_n}(x) = 2$$
$$d_{K_n}(x) = n - 1$$

para todo vértice de Cn, mientras que para todo vértice de Kn.

Por lo tanto ambos son grafos regulares.

es 2-regular y Kn es (n − 1)-regular).

camino

$$x_1, \ldots, x_r$$
$$x_1 = x$$
$$x_r = y$$
$$x_i x_{i+1} \in E \; \forall i \in \{1, 2, ..., r-1\}$$

“x ∼ y sii existe un camino entre x e y”

es una relación de equivalencia.

Por

lo tanto el grafo G se parte en clases de equivalencia de esa relación de equivalencia.

Esas partes se llaman las componentes conexas de G.

componentes conexas

Grafos conexos

Un grafo se dice conexo si tiene una sola componente conexa.

Cn y Kn son conexos.

arbol

es un grafo conexo sin ciclos.

Determinación de las componentes conexas

El algoritmo básico de DFS o BFS lo que hace es, dado un vértice x, encontrar todos los vértices de la componente conexa de x.

algoritmo

(abajo en vez de BFS puede usarse DFS)

Tomar W = $\emptyset$, i = 1.

Tomar un vértice cualquiera x de V.

Correr BFS(x).

LLamarle Ci a la componente conexa que encuentra BFS(x).

Hacer W = W $\cup$ (vértices de Ci).

Si W = V, return C1, C2, ..., Ci.

Si no, hacer i = i + 1, tomar un vértice x / $\in$ W y repetir [3].

DFS y BFS

breve repaso

a partir de un vértice raiz, los algoritmos van buscando nuevos vértices, buscando vecinos de vértices que ya han sido agregados. DFS agrega de a un vécino por vez y usa una pila.

BFS agrega todos los vecinos juntos y usa una cola.

BFS(x):

Crear una cola con x como único elemento.

Tomar C = {x}. WHILE (la cola no sea vacia)

Tomar p=el primer elemento de la cola. Borrar p de la cola. IF existen vértices de Γ (p) que no esten en C:

Agregar todos los elementos de Γ (p) que no estén en C a la cola y a C.

ENDWHILE

return C.

DFS(x):

Crear una pila con x como único elemento.

Tomar C = {x}. WHILE (la pila no sea vacia)

Tomar p=el primer elemento de la pila. IF existe algún vértice de Γ (p) que no esté en C:

Tomar un $q \in \Gamma$ (p) $-$ C. $\in -$ Hacer C = C $\cup$ {q}. $\cup$ {q}. Agregar q a la pila.

ELSE:

Borrar p de la pila.

ENDWHILE

return C.

Complejidad

la complejidad tanto de DFS como de BFS es O(m).

Coloreos propios

Un coloreo (de los vértices) es una función cualquiera $c : V \to S$ donde S es un conjunto finito.

Un coloreo es propio si $xy \in E \Rightarrow c(x) \neq c(y)$ (extremos con distinto color)

Si la cardinalidad de S es k diremos que el coloreo tiene k colores. En general usaremos $S = \{0, 1, \ldots, k-1\}$ para denotar los colores.

Un grafo que tiene un coloreo propio con k colores se dice k-coloreable.

número cromático

χ (G) = min{k : $\exists$ un coloreo propio con k colores de G}

Calculando χ (G)

Si uno dice que χ (G) = k, por la definición misma de este número, hay que hacer dos cosas para probarlo:

1 Dar un coloreo propio de G con k colores. (y obviamente probar que es propio).

Esto prueba la parte del " $\exists$ un coloreo propio con k colores de G"

2 Probar que no existe ningún coloreo propio con k − 1 colores de G.

Esto prueba que k es el mínimo.

ayuda útil para probar [2]

Si H es un subgrafo de G, entonces χ (H) $\leq$ χ (G).

Entonces si encontramos un subgrafo H de G para el cual sepamos que χ (H) = k habremos probado [2].

prueba por contradicción:
se asume que existe un coloreo propio con k − 1 colores y deduciendo cosas, se llega a un absurdo.

Hay 2 problemas

1 Llegar al absurdo puede ser bastante dificil, teniendo que contemplar varios casos, pej.

2 Para poder hacer la prueba por contradicción, hay que asumir que existe un coloreo propio con k − 1 colores.

− Eso significa que uds. NO TIENEN CONTROL sobre ese coloreo. Sólo saben que hay uno, y deben deducir cosas sobre ese coloreo a partir de la estructura del grafo.

χ (G) para algunos grafos

En general, dado que para cualquier grafo G podemos darle un color distinto a todos los vértices, tenemos la desigualdad χ (G) $\leq$ n. χ (Kn) = n si quieren probar que r $\leq \chi$ (G) basta con ver que existe un Kr subgrafo de G. χ (G) = 1 si y solo si E = $\emptyset$ asi que para cualquier grafo que tenga al menos un lado, χ (G) $\emptyset \geq 2$.

$$\chi(C_{2r}) = 2$$

pues podemos colorear c(i) = (i mod 2)

$$\chi(C_{2r+1})$$

con tendriamos que 2r + 1 y 1 tendrían color 1, absurdo pues forman lado. Podemos colorear: c(i) = (i mod 2) si i < 2r + 1 y c(2r + 1) = 2.

los ciclos impares tienen número cromático igual a 3.

cualquier grafo que tenga como subgrafo a un ciclo impar debe tener número cromático mayor o igual que 3.

Algoritmo de fuerza bruta

simplemente tomar todos los coloreos posibles con los colores {0, 1, ..., n − 1} y calcular cuales {0, − 1} de esos coloreo son propios, y ver de entre esos quien tiene la menor cantidad de colores.

Este algoritmo calcula χ (G) pero:

Hay nn posibles coloreos. Chequear que un coloreo es propio es O(m).

el algoritmo tiene complejidad O(nnm) asi que no es útil salvo para n muy chicos.

Algoritmo Greedy

El algoritmo Greedy requiere como input no sólo un grafo G sino un **orden** de los vértices.

para extraer el mayor beneficio posible de Greedy conviene poder llamarlo varias veces cambiando el orden.

Idea de Greedy

La idea de Greedy consiste de dos partes:

1 Ir coloreando los vértices de G uno por uno, en el orden dado, manteniendo siempre el invariante que el coloreo parcial que se va obteniendo es propio.

2 Darle a cada vértice al momento de colorearlo el menor color posible que se le pueda dar manteniendo el invariante de que el coloreo es propio.

Greedy

Input: Grafo G y orden de los vértices

$$x_1, x_2, \ldots, x_n.$$

$$c(x_1) = 0$$

Para i > 1, asumiendo que los vértices

$$x_1, x_2, \ldots, x_{i-1}$$

ya han sido coloreados, colorear xi con:

$$c(x_i) = \min\{k \geq 0 : k \notin c(\{x_1, \ldots, x_{i-1}\} \cap \Gamma(x_i))\}$$

estamos usando la notación usual de c(A) = {c(a) : a ∈ A}.

Es decir, xi recibe el menor color que sea distinto del color de todos los vecinos anteriores a xi.

Complejidad de Greedy

la complejidad de Greedy es

$$O(d(x_1) + d(x_2) + \cdots + d(x_n)).$$

Por el lema del apretón de manos que vieron en Discreta I, la suma de todos los grados es igual a 2m.

Por lo tanto ## Cotas para Greedy

χ (G) $\leq \Delta + 1$

Teorema de Brooks

Si G es conexo, entonces χ (G) $\leq \Delta$, a menos que G sea un ciclo impar o un grafo completo.

Propiedad

Si G es conexo, entonces existe un ordenamiento de los vértices tal que Greeedy colorea todos los vértices, salvo uno, con Δ colores o menos.

VIT

Very Important Theorem

Sea G = (V, E) un grafo cuyos vértices estan coloreados con un coloreo propio c con r colores {0, 1, ..., r − 1}. {0, − 1}. Sea π una permutación de los números 0, 1, ..., r − 1, es decir, − π : {0, 1, ..., r − 1} → {0, 1, ..., r − 1} es una biyección. {0, Sea Vi = − 1} → {0, − 1} {x ∈ V : c(x) = i}, i = 0, 1, ..., r − 1. {x ∈ i}, − Ordenemos los vértices poniendo primero los vértices de V π (0), luego los de V π (1),etc, hasta V π (r − 1). (el orden interno de los vértices dentro de cada V π (i) es irrelevante)

Entonces Greedy en ese orden coloreará G con r colores o menos.

Corolario

Existe un ordenamiento de los vértices de G tal que Greedy colorea G con χ (G) colores.

Consecuencia

si no podemos obtener χ (G) polinomialmente, usaremos el VIT para tratar de obtener una aproximación a χ (G).

No siempre se puede, pero en la practica suele funcionar bastante bien, dependiendo de cuales permutaciones π se usen.

Grafos bipartitos

Un grafo se dice bipartito si χ (G) = 2.

Es decir, si G = (V, E) entonces existen X, Y $\subseteq$ V tales que:

1 V = X $\cup$ Y. $\cup$ 2 X $\cap$ Y = $\emptyset$

$\cap$ $\emptyset$ 3 wv $\in$ E $\Rightarrow$ (w $\in$ X, v $\in$ Y) $\vee$ (w $\in$ Y, v $\in$ X)

El problema 2COLOR

Dado un grafo G, ¿es χ (G) $\leq$ 2?

Teorema

2COLOR es polinomial

Algoritmo 2COLOR para G conexo.

Elegir un vértice x cualquiera.

Correr BFS(x), creando un arbol.

Para cada vértice z, sea N(z) el nivel de z en el arbol BFS(x).

Colorear c(z) = (N(z) mod 2).

Chequear si el colorario dado en [4] es propio.

Si lo es, retornar " χ (G) $\leq$ 2"

Si no lo es, retornar " χ (G) $>$ 2"

Complejidad

la complejidad total es O(m) + O(m) = O(m).

Corolario

Sea G un grafo con χ (G) $\geq$ 3.

Como χ (G) $\geq$ 3, el coloreo de 2 colores dado en el algoritmo no $\geq$ puede ser propio.

Conclusión:

Flujos Y Networks.

Grafos Dirigidos

Definición:

Un Grafo dirigido es un par G = (V, E) donde V es un conjunto cualquiera (finito para nosotros) y E $\subseteq$ V $\times$ V

diferencia con un grafo no dirigido

E $\subseteq$ V $\times$ V

ahora los lados son pares ordenados en vez de conjuntos.

no es lo mismo (x, y) que (y, x)

Notación:

Denotaremos el lado (x, y) como

Vecinos

Pero ahora como podemos tener lados tanto (x, y) como (y, x) deberiamos diferenciar entre “vécinos hacia adelante” y “vécinos hacia atras”

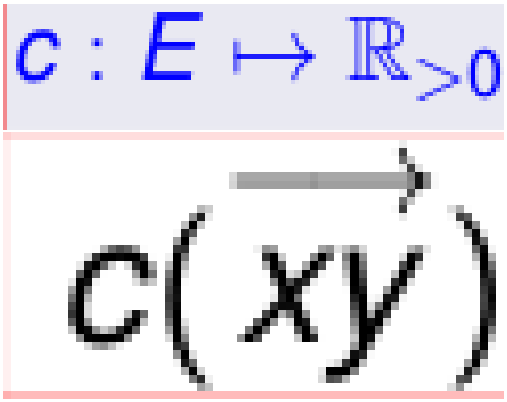
Notación:

$$\Gamma^{+}(x) = \{y \in V | \overrightarrow{xy} \in E\}$$
$$\Gamma^{-}(x) = \{y \in V | \overrightarrow{yx} \in E\}$$

Network

Definición:

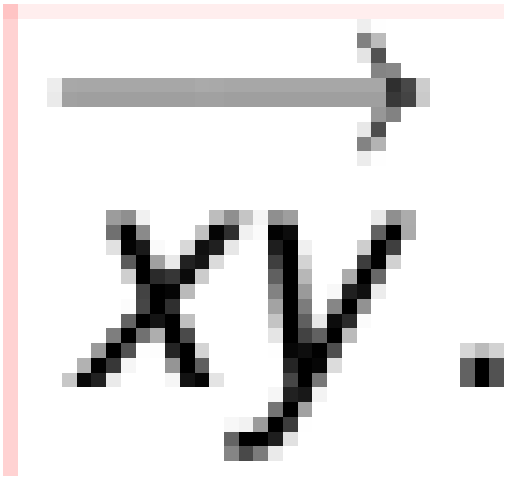
Un Network es un grafo dirigido con pesos positivos en los lados, es decir, un triple (V, E, c) donde (V, E) es un grafo dirigido y



En este contexto, se llamará la

“capacidad”

del lado



Flujos

Notación para agilizar lecturas de sumatorias

P
Si P es una propiedad que puede ser verdadera o falsa, P denota el número 1 si P es verdadera, y 0 si P es falsa.

Supongamos que tenemos una variable x, y queremos sumar una función f(x) sobre todos los x que satisfagan una propiedad P(x)

podemos simplemente escribir

$$\sum_x f(x)[P(x)]$$

o incluso

$$\sum f(x)[P(x)]$$

si queda claro que sumamos sobre x.

Notación para funciones sobre lados

Si g es una función definida en los lados y A y B son subconjuntos de vertices, entonces g(A, B) denotará la suma:

$$g(A,B) = \sum_{x,y} [x \in A][y \in B][\overrightarrow{xy} \in E] g(\overrightarrow{xy})$$

in y out

Dada una función g sobre lados y un vértice x, definimos:

outg(x) es todo lo que "sale" de x por medio de g.

ing(x) es todo lo que "entra" a x por medio de g.

$$out_g(x) = \sum_y [y \in \Gamma^+(x)] g(\overrightarrow{xy}) = g(\{x\}, \Gamma^+(x))$$
$$in_g(x) = \sum_y [y \in \Gamma^-(x)] g(\overrightarrow{yx}) = g(\Gamma^-(x), \{x\})$$

Definición

Dado un network (V, E, c), y un par de vertices s, t ∈ V, un ∈ flujo de s a t es una función f : E → R con las siguientes

propiedades:

$$0 \leq f(\overrightarrow{xy}) \leq c(\overrightarrow{xy}) \quad \forall \overrightarrow{xy} \in E.$$

("feasability")

inf(x) = outf(x) $\forall$ x $\in$ V $-$ {s, t}. ("conservación")

$$out_f(s) \geq in_f(s).$$

(s es productor)

(t es consumidor)

$$in_f(t) \geq out_f(t).$$

Explicación

la primera propiedad dice que no vamos a transportar una cantidad negativa de un bien

ni vamos a tranportar por encima de la capacidad de transporte de un lado.

La segunda propiedad dice que el network no tiene "pérdidas" .

La tercera especifica que s es un vértice donde hay una producción neta de bienes, pues produce mas de lo que consume.

y la cuarta que t es un vértice donde se consumen los bienes pues consume mas de lo que produce.

En algunos libros en vez de 3) se pide directamente

$$in_f(s) = 0$$

y en vez de 4) se pide

$$out_f(t) = 0.$$

en todos los ejemplos que usaremos,

$$\Gamma^-(s) = \Gamma^+(t) = \emptyset$$

s se llama tradicionalmente la “fuente”(source)

y t el “resumidero”(sink).

Valor de un flujo

Definición

Dado un network (V, E, c) el **valor** de un flujo f de s a t es:

$$v(f) = out_f(s) - in_f(s)$$

el valor de un flujo es la cantidad neta de bienes producidos.

Flujos maximales

Definición

Dado un network N y vertices s, t, **un flujo maximal de s a t** (o “Max flow”) es un flujo f de s a t tal que v(g) $\leq$ v(f) para todo flujo g de s a t.

Propiedad

Flujos: Greedy.

notación g(A, B)

g es una función sobre los lados y A, B $\subseteq$ V

$$g(A, B) = \sum_{x,y}[x \in A][y \in B][\overrightarrow{xy} \in E]g(\overrightarrow{xy})$$

Propiedad:

Sean f, g funciones sobre los lados tales que

$$g(\overrightarrow{xy}) \leq f(\overrightarrow{xy}) \quad \forall \overrightarrow{xy} \in E$$

Entonces

g(A, B) $\leq$ f(A, B) $\forall$ A, B $\subseteq$ V

Criterio simple para maximalidad

Propiedad:

Sea f flujo en un network N tal que v(f) = c({s}, V). Entonces f es maximal.

Existencia

de la definición no es claro que EXISTA un flujo maximal.

flujo sea "entero",

es decir que las capacidades y el flujo en cada lado deben ser números enteros,

entonces,

como hay una cantidad finita de flujos enteros, es claro que existe un flujo entero maximal.

Greedy

Algoritmo

$$f(\overrightarrow{xy}) = 0 \forall \overrightarrow{xy} \in E$$

Comenzar con f = 0 (es decir,

Buscar un camino dirigido s = x0, x1, ..., xr = t, con

$$\overrightarrow{x_i x_{i+1}} \in E$$

tal que

$$f(\overrightarrow{x_i x_{i+1}}) < c(\overrightarrow{x_i x_{i+1}})$$

para todo $\in$ i = 0, ..., r − 1.

(llamaremos a un tal camino un camino dirigido “no saturado” .)

Calcular

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

Aumentar f a lo largo del camino de 2. en ϵ , como se explicó antes.

Repetir 2 hasta que no se puedan hallar mas caminos con esas condiciones.

Conclusiones sobre Greedy

este Greedy no necesariamente va a encontrar un flujo maximal.

eligiendo inteligentemente los caminos encontramos un flujo maximal.

el Greedy de caminos puede ser modificado para encontrar un flujo maximal en tiempo polinomial

Not Greedy

En el caso de flujos, se puede construir un algoritmo que corre Greedy y cuando llega a un cierto punto, “SE DA CUENTA” que se equivocó en la elección de los caminos

y CORREGIR los errores.

Definición de Corte

Un Corte es un subconjunto de los vertices que tiene a s pero no tiene a t.

Capacidad de un Corte

La capacidad de un corte es cap(S) = c(S, S), donde S = V − S

Definición:

Ford-Fulkerson

Complejidad de Greedy

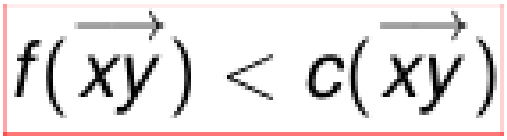
Como en Greedy los lados nunca se des-saturan, entonces Greedy puede hacer a lo sumo O(m) incrementos de flujo antes de que forzosamente deba terminar si o si.

Encontrar un camino dirigido no saturado es O(m)

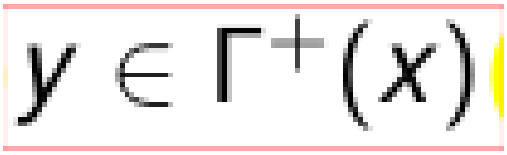
la complejidad total de Greedy es O(m2).

FF

idea

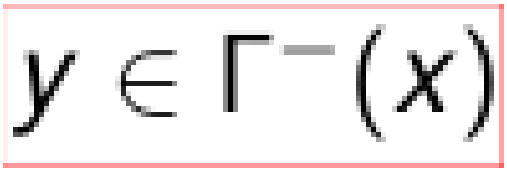


en vez de limitar la busqueda a



con

permiten ademas buscar



con

$$f(\overrightarrow{yx}) > 0$$

Camino aumentante

Un camino aumentante (o f-camino aumentante si necesitamos especificar f) o camino de Ford-Fulkerson, es una sucesión de vértices x0, x1, ..., xr tales que:

x0 = s, xr = t. Para cada i = 0, ..., r − 1 ocurre una de las dos cosas siguientes:

$$x_i\overrightarrow{x_{i+1}} \in E \text{ y } f(x_i\overrightarrow{x_{i+1}}) < c(x_i\overrightarrow{x_{i+1}})$$

1

$$\overrightarrow{x_{i+1}x_i} \in E \text{ y } f(\overrightarrow{x_{i+1}x_i}) > 0.$$

2

Si en vez de comenzar en s y terminar t el camino es como arriba pero con x0 = x,xr = z diremos que es un camino aumentante **desde x a z**

Lados forward y backward

A los lados en 1) los llamaremos "lados de tipo I" o **"lados forward"**

A los lados en 2) los llamaremos "lados de tipo II" o **"lados backward"**

Algoritmo de Ford-Fulkerson

$$f(\overrightarrow{xy}) = 0 \forall \overrightarrow{xy} \in E$$

Comenzar con f = 0 (es decir,

Buscar un f-camino aumentante s = x0, x1, ..., xr = t.

Definir ϵ i de la siguiente manera:

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

en los lados forward.

$$\varepsilon_i = f(\overleftarrow{x_{i+1} x_i})$$

en los lados backward.

Calcular $\epsilon = \min\{ \epsilon \text{ i}\}$.

Cambiar f a lo largo del camino de [2] en ϵ , de la siguiente forma:

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

en los lados forward.

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

en los lados backwards.

Repetir [2] hasta que no se puedan hallar mas caminos aumentantes.

FordFulkerson mantiene "flujicidad"

Si f es un flujo de valor v y aumentamos f con un f-camino aumentante con ϵ calculado como se explica en el algoritmo de Ford-Fulkerson, entonces lo que queda sigue siendo flujo y el valor del nuevo flujo es v + ϵ

Complejidad de Ford-Fulkerson

NO ES polinomial:

Max Flow Min Cut

Teorema

$$v(f) = f(S, \overline{S}) - f(\overline{S}, S)$$

A
Si f es un flujo y S es un corte, entonces

B
El valor de todo flujo es menor o igual que la capacidad de todo corte.

C
Si f es un flujo, las siguientes afirmaciones son equivalentes:

1 Existe un corte S tal que v(f) = cap(S). 2 f es maximal. 3 No existen f-caminos aumentantes.

Corolario

Si el algoritmo de Ford-Fulkerson termina, termina con un flujo maximal

Teorema de la Integralidad

Teorema de la integralidad.

En un network con capacidades enteras, todo flujo entero maximal es un flujo maximal.

Teorema

Ford-Fulkerson con DFS

1 Creamos una pila con s.

2 Si la pila es vacia, terminamos, no hay camino. Si no es vacia, tomamos x =el primer elemento de la pila y buscamos algún vécino de x que satisfaga las condiciones de Ford-Fulkerson.

3 Si no hay, sacamos a x de la pila y repetimos 2). 4 Si hay tal vécino, tomamos z uno de ellos. 5 Si z = t encontramos nuestro camino.

6 Si no, agregamos z a la pila y repetimos 2).

ventaja

DFS es O(m) asi que la búsqueda de caminos es polinomial.

desventaja

con DFS Ford-Fulkerson puede no terminar nunca,

Edmonds y Karp

propusieron estas dos alternativas.

aumentar eligiendo caminos de longitud mínima, y aumentar eligiendo caminos de aumento máximo.

Algunos libros lo llaman "heurística"

porque no es un nuevo algoritmo, sino que es Ford-Fulkerson con la especificación de usar BFS para la búsqueda.

buena forma de recordarlo

es que EK=FF+BFS.

Otra cosa que tienen que hacer

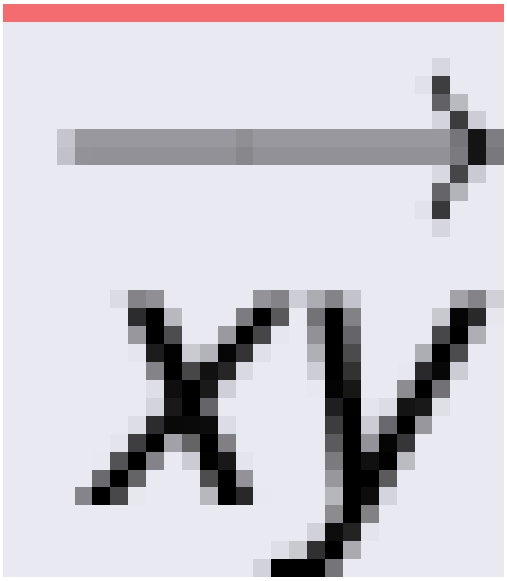
Complejidad de Edmonds-Karp

Teorema de Edmonds-Karp

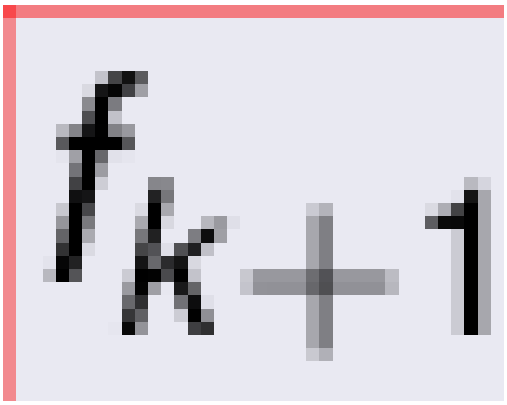
La complejidad del algoritmo de Edmonds-Karp es O(nm2)

Lados críticos

Definición



Diremos que un llado **se vuelve crítico** durante la construcción de uno de los flujos intermedios (digamos, fk+1) si para la construcción de



pasa una de las dos cosas siguientes:

1 Se usa el lado en forma forward, saturandolo (es decir

$$f_k(\overrightarrow{xy}) < c(\overrightarrow{xy})$$
$$f_{k+1}(\overrightarrow{xy}) = c(\overrightarrow{xy})$$

pero luego

2 O se usa el lado en forma backward, vaciandolo (es decir

$$f_k(\overrightarrow{xy}) > 0$$
$$f_{k+1}(\overrightarrow{xy}) = 0$$

pero

distancias

Definición
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. **La denotaremos como df(x, z).**

Notación
Dado un vértice x denotamos

$$d_k(x) = d_{f_k}(s, x)$$

y

$$b_k(x) = d_{f_k}(x, t).$$

Es decir,
dk(x) es la longitud del menor fk-camino aumentante entre s y x y bk(x) es la longitud del menor fk-camino aumentante entre x y t.

Definición

Dado un flujo f y un vértice x, diremos que un vértice z es un vécino fFF de x si pasa alguna de las siguientes condiciones:

$\overrightarrow{xz} \in E$ y $f(\overrightarrow{xz}) < c(\overrightarrow{xz})$ o:
$\overrightarrow{zx} \in E$ y $f(\overrightarrow{zx}) > 0$.

Observación trivial:

Si z es un fkFF vécino de x, entonces dk(z) $\leq$ dk(x) + 1

Lema de las distancias

Las distancias definidas anteriormente no disminuyen a medida que k crece.

$d_k(x) \leq d_{k+1}(x)$ y $b_k(x) \leq b_{k+1}(x) \forall x$

Es decir,

Existencia de flujos maximales

Dado que hemos probado que Edmonds-Karp siempre termina, y dado que produce un flujo maximal,

entonces tambien hemos probado que # El algoritmo de Dinitz

idea básica de Dinitz

"guardar" todos los posibles caminos aumentantes de la misma longitud (mínima) en una estructura auxiliar.

esta primera parte se hace, al igual que con Edmonds-Karp, con BFS, pero guardamos toda la información y no sólo la necesaria para construir un camino.

Esquema básico de Dinitz

1 Construir un network auxiliar (usando BFS). 2 Correr Greedy con DFS en el network auxiliar hasta no poder seguir.

3 Usar el flujo obtenido en el network auxiliar para modificar el flujo en el network original.

4 Repetir [1] con el nuevo flujo, hasta que, al querer construir un network auxiliar, no llegamos a t.

En el network auxiliar, como se usa Greedy, nunca se des-satura un lado. los lados siguen pudiendo des-saturarse, es sólo en el network auxiliar que no se des-saturan.

Flujos bloqueantes

Definición:
Llamaremos a un flujo en un network si todo camino DIRIGIDO desde s a t tiene al menos un lado

$$c(\overrightarrow{xy}) = f(\overrightarrow{xy})$$

saturado. (es decir con

En otras palabras, si cuando queremos usar Greedy en el network, no llegamos a t.

Algoritmos tipo Dinic

1 Construir un network auxiliar (usando BFS).

2 Encontrar un flujo bloqueante en el network auxiliar. 3 Usar ese flujo bloqueante del network auxiliar para modificar el flujo en el network original.

4 Repetir [1] con el nuevo flujo, hasta que, al querer construir un network auxiliar, no llegamos a t.

Layered Networks

network "por niveles".

Definición:
Un Network por niveles es un network tal que el conjunto de vértices esta dividido en subconjuntos Vi (los "niveles") tales que sólo existen lados entre un nivel y el siguiente.

$$\overrightarrow{xy} \in E \Rightarrow \exists i : x \in V_i,\ y \in V_{i+1}$$

Es decir,

Network auxiliar,

vértices

$$V = \cup_{i=0}^{r} V_i$$

el conjunto de vértices es donde los Vi son:

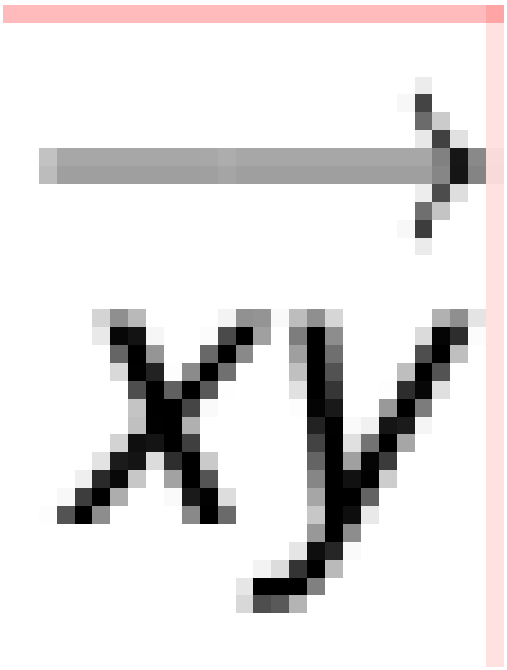
Sea r = df(s, t) donde df es la función definida en la prueba de Edmonds-Karp.

Es decir, r es la distancia entre s y t usando caminos aumentantes.

Para i = 0, 1, ..., r − 1, definimos Vi = {x : df(s, x) = i}.

Observar que entonces V0 = {s}.

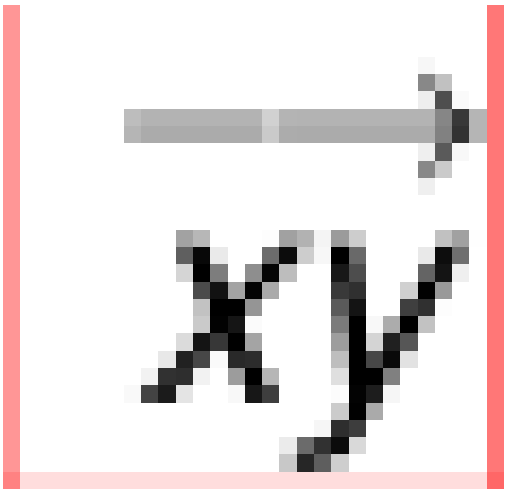
Definimos Vr = {t}



Lados y capacidades:

es un lado del network auxiliar si:

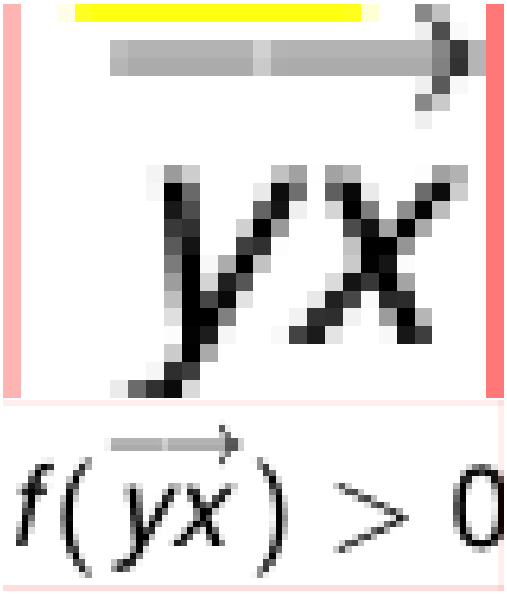
x ∈ Vi, y ∈ Vi+1 y:



es un lado del network original con

1

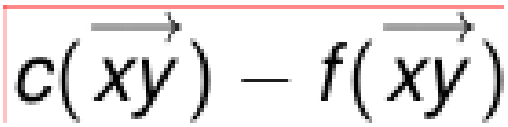
o:



es un lado del network original con

2

En el caso de [1], la capacidad de en el network



auxiliar será y en el caso de [2], la

capacidad del lado en el network auxiliar será

Otra forma de pensar esto

es que construimos primero un “network residual”

Cuyos lados son:

los lados originales, con capacidad igual a c − f − Y los reversos de los lados originales, con capacidad f.

Y luego, de ese network residual nos quedamos con los lados que unan vertices de distancia i con vértices de distancia i + 1.

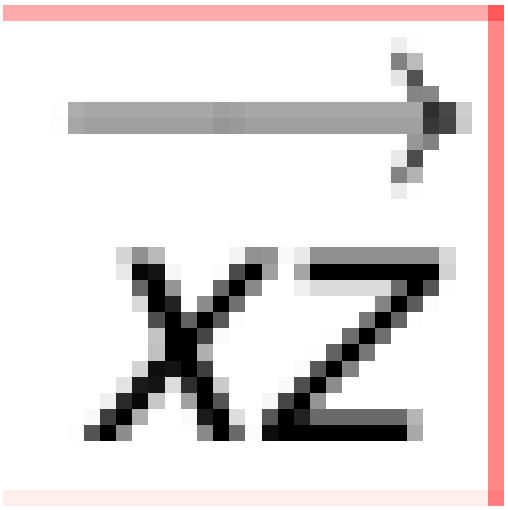
Construcción

la forma de construirla es tomar como V0 a {s}.

Y luego ir construyendo una cola a partir de s al estilo Edmonds-Karp.

Y si x agrega a z y x está en Vi, entonces z está en Vi+1.

si z ya está agregado, si bien z no se



vuelve a agregar, el lado si se agrega al network auxilir, siempre y cuando la distancia de z a s sea uno mas que la distancia de x a s.

Si en algún momento llegamos a t, no paramos inmediatamente, pues podria haber mas lados que lleguen a t.

Pero borramos todos los vértices que ya hubieramos incluido en el mismo Vr en el cual estamos poniendo a t

Y de ahi en mas no agregamos mas vértices, sólo lados entre vértices de

y t.

Observaciones

Como el network auxiliar es un network por niveles, **todos** los caminos de un mismo network auxiliar deben tener **la misma longitud.**

Complejidad “naive”de Dinitz

depende de cuantos networks auxiliares tengamos que construir.

La construcción de cada uno es O(m).

complejidad total de hallar un flujo bloqueante en un network auxiliar igual a O(m2).

la complejidad total de Dinica seria n.(O(m) + O(m2)) = O(nm2),

idea de la implementación de Ever:

cuando corremos DFS, si llegamos a un vértice x que no tiene vecinos, debemos hacer un backtrack, borrando a x de la pila y usando el vértice anterior a x en el camino para seguir buscando. Esa información de que es inútil seguir buscando por x **no deberiamos perderla** y hay que “guardarla” para futuras corridas de DFS. La forma que tiene Ever de “guardar”esa información es simplemente borrar x, o bien, si hacemos backtrack desde x a z,

borrar el lado

Diferencia entre la version rusa y la occidental de Dinitz

La diferencia entre Dinitz y Dinic-Even es en cómo y cuando se actualiza el network a medida que encontramos nuevos caminos.

Ever

borra varios lados (o vértices) extras mientras corre DFS, cada vez que tiene que hacer un backtrack.

Dinitz,

el network auxiliar se construye de forma tal que **DFS nunca tenga que hacer backtrack.**

Y cada vez que se encuentra un camino entre s y t y se cambia el flujo, tambien se cambia el network auxiliar para seguir teniendo esta propiedad.

Es decir,

en el original se usa un poco mas de tiempo actualizando el network auxiliar luego de cada camino,

mientras que en la versión de Even, no se pierde tanto tiempo actualizando el network auxiliar **entre** caminos, pero las busquedas DFS no demoran todas igual

Ever es mas “lazy” y sólo borra lados si los encuentra y se da cuenta que no los necesita, mientras que la versión original de Dinitz es mas proactiva y borra todos los lados que sabe que son inútiles aún si luego nunca los