PDF Demostraciones Discreta 2 by Willytp

Teoremas Networks:

Comp-EK

¿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 EK. Ud. puede usar esto sin necesidad de probarlo)

Enunciado

Sea n la cantidad de vértices y m la cantidad de lados, E-K tiene complejidad $O(nm^2)$

Estructura de la prueba

Supondremos que en el network no existen arcos paralelos xy e yx. Esta propiedad no es restrictiva, ya que para cualquier network N que presente arcos paralelos, podemos dar un N' tal que sea como N PERO con vértices "intermedios" adicionales en medio de cada arco paralelo.

Primero queremos ver que los flujos parciales $f_0, f_1, f_2, \ldots$ son finitos y daremos una cota para ellos.

Como la búsqueda y construcción de cada flujo parcial se hace con BFS, cada incremento tiene complejidad $O(m)$ (máximo vas a tener que ver todos los lados hasta llegar a t)

Si probamos que solo puede haber $O(nm)$ búsquedas, entonces habremos probado la complejidad.

Estructura:

- Damos lema y definiciones
- Acotamos la cantidad de veces que se puede volver crítico un lado ($O(n)$)
- Concluímos que como todo lado puede volverse crítico $O(n)$ veces, y siempre se vuelve crítico un lado en un nuevo paso de E-K, la cantidad de pasos es $O(nm)$
- La complejidad total es entonces $O(nm^2)$

Las definiciones están despúés de la prueba.

Prueba

Vemos que se puede hacer crítico un lado (n-1)/2 veces viendo que entre dos pasos k y r la f-distancia entre s y t debe haber aumentado en al menos 2.

Supongamos que en el paso k **se satura xy** para obtener el flujo f_{k+1}, por lo tanto existe un camino de longitud mínima s ... xy ... t. Entonces

$$d_k(y) = d_k(x) + 1 \text{(i)}$$

Luego para que vuelva a hacerse crítico xy en un paso r tiene que haber un paso intermedio l tal que $r \geq l > k$ en el que se vacíe al menos un poco. **Ya sea para que se vuelva a saturar, o bien para que luego (o en esa misma instancia, si r = l) se vacíe por completo.** Tenemos entonces un camino s ... ybackx ... t y por lo tanto

$$d_l(x) = d_l(y) + 1 \text{ (ii)}$$

Ahora hacemos cuentas para ver cuanto tiene que aumentar la distancia a t entre cada evento de criticalidad sobre un lado xy:

$$d_l(t) = d_l(x) + b_l(x)$$

$$d_l(t) = d_l(y) + 1 + b_l(x) \text{ por (ii)}$$

$$d_l(t) \geq d_k(y) + 1 + b_k(x) \text{ por lema distancias}$$

$$d_l(t) \geq d_k(x) + 1 + 1 + b_k(x) \text{ por (i)}$$

$$d_l(t) \geq d_k(t) + 2$$

Ahora veamos el caso en el que se **vacía xy** para obtener el flujo f_{k+1}:
Mismo planteo con $r \geq l > k$ pero en este caso tenemos que en el paso k tenemos el camino s ... ybackx ... t y por lo tanto

$$d_k(x) = d_k(y) + 1 \text{ (iii)}$$

Luego en el paso l tenemos que se debe saturar del todo o un poco el lado xy. Tenemos:

$$d_l(y) = d_l(x) + 1 \text{ (iv)}$$

Hacemos las cuentas otra vez:

$$d_l(t) = d_l(y) + b_l(y)$$

$$d_l(t) = d_l(x) + 1 + b_l(y) \text{ por (iv)}$$

$$d_l(t) \geq d_k(x) + 1 + b_k(y) \text{ por lema}$$

$$d_l(t) \geq d_k(y) + 1 + 1 + b_k(y) \text{ por (iii)}$$

$$d_l(t) \geq d_k(t) + 2$$

En ambos casos vemos que $d_r(t) \geq d_l(t) => d_r(t) \geq d_k(t) + 2$

Como la máxima distancia hasta t es n-1, tenemos que un lado puede hacerse crítico $\frac{n-1}{2}$ veces: $O(n)$.

Por lo tanto la complejidad total es $O(flujos * BFS) = O(m * n * m) = O(nm^2)$

Definiciones / lemas internos

Lema de las distancias (no se pide la prueba)

las distancias $d_k(x)$ y $b_k(x)$ son <= a $d_{k+1}(x)$ y $b_{k+1}(x)$ respectivamente.

Lado crítico

Un lado xy se vuelve crítico si:

- se satura completamente
- se vacía completamente

Distancias

$$d_k(x) = \text{ longitud del menor } f_k\text{-camino aumentante de s a x}$$

$$b_k(x) = \text{ longitud del menor } f_k\text{-camino aumentante de x a t}$$

Propiedad de las distancias

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 df (x, z), y definimos dk(x) = dfk (s, x), donde fk es el k-ésimo flujo en una corrida de Edmonds-Karp, entonces dk(x) ≤ dk+1(x).

Suposicion: Asumimos para esta prueba que si existe el lado xz entonces no existe el lado zx. Esto ya que por lo que vimos en el práctico, de cualquier network podemos definir uno nuevo en el que ponemos en cada arco paralelo "un vértice intermedio" y así tener un problema sin esta propiedad que tiene la misma solución max-flow.

Definición: fFF Vecino

decimos que z es fFF vecino de x si desde x podemos enviar flujo a z o devolver flujo a z.

Estructura

- Asumimos que $A = \{y : d_{k+1}(y) < d_k(y)\} \neq \emptyset$
- Tomando algún y en A
- Puntualmente tomaremos al x tal que d_{k+1} sea mínimo.
- Probamos que existe un z anterior a x desde el que antes no podías llegar a x pero ahora si.
  - Vemos que dada esa situación, $z \notin A$ y por lo tanto $d_k(x) > d_k(z) + 1$ ESTA ES LA PARTE DIFICIL
- Luego resta ver que para que esto haya ocurrido, antes tomamos un camino s ... x z ... t y por lo tanto $d_k(z) = d_k(x) + 1$
- Usando las conclusiones de los últimos dos pasos llegamos a que 0 > 2

Prueba

Por hipótesis tenemos que $d_{k+1}(x) < d_k(x) \leq \infty$ y por lo tanto existe camino f_{k+1} aumentante de menor longitud hasta x.

Observación: s no pertenece a A, pues la distancia a si mismo no puede bajar de 0. Por lo tanto, en el camino que consideramos existe un z directamente anterior a x.

Tenemos entonces que $d_{k+1}(z) = d_{k+1}(x) - 1$, pues x es fFF vecino de z.
Entonces $d_{k+1}(z) < d_{k+1}(x)$ y por lo tanto $z \notin A$

Por la definicón de A, $d_k(z) \leq d_{k+1}(z)$

Entonces tenemos:

$$d_k(x) > d_{k+1}(x) \text{ porque } x \in A = d_{k+1}(z) + 1 \text{ porque x es fFF vecino de z } \geq d_k(z) + 1$$

$$d_k(x) > d_k(z) + 1$$

Por lo tanto x no es fFF vecino de z en el paso k.

Esto **solo puede pasar** si el camino que usamos para pasar de f_k a f_{k+1} incluye xz o xbackz.

Veamos cada caso:

- $\overrightarrow{xz}$
  Tenemos que $f_k(\overrightarrow{xz}) = 0$ y $f_{k+1}(\overrightarrow{xz}) > 0$ por lo que vimos de la vecindad de z a x en cada paso. Por lo tanto de k a k+1 ENVIAMOS flujo de x a z.
- $\overrightarrow{zx}$
  Tenemos que $f_k(\overrightarrow{zx}) = c(\overrightarrow{xz})$ y $f_{k+1}(\overrightarrow{xz}) < c(\overrightarrow{xz})$ por lo que vimos de la vecindad de z a x en cada paso. Por lo tanto de k a k+1 DEVOLVEMOS flujo de x a z.

En **ambos casos** la longitud del camino que usamos es mímina y por lo tanto

$$d_k(z) = d_k(x) + 1$$

Pero entonces

$$d_k(z) = d_k(x) + 1 > (d_k(z) + 1) + 1$$

$$0 > 2$$

Absurdo. Lo único que asumismos hasta ahora fue que A no era vacío, y por lo tanto lo es.

Notas

Por qué $d_k(z) = d_k(x) + 1$?

- Porque si bien tendríamos sólo que $d_k(z) \leq d_k(x) + 1$ si tuvieramos que z es f_kFF vecino de x, sabemos también en ese punto de la demostración que en el paso k ELEGIMOS un camino s ... xz ... t y por lo tanto la longitud de s ... xz es la mínima posible.

Comp-Dinic

¿Cual 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 auxiliares sucesivos aumenta).

Estructura

- Enunciamos que la longitud de los sucesivos networks auxiliares crece.
- Primero mostramos los aspectos generales de la complejidad de algoritmos "tipo Dinic"
- Luego vemos cuánto cuesta conseguir un flujo bloqueante en cada implementación
- Dinitz: Vemos cuanto cuesta ver que borrar en cada poda(n) + cuanto cuesta borrar en general(2m).
- even: Planteamos que la corrida son palabras A+(R|I) y vemos cuantas palabras puede haber(m) y cuanto cuesta cada palabra(O(n))
- Ambas tienen entonces costo O(mn)

Prueba

Como sabemos que los networks auxiliares crecen en longitud, tenemos que a lo sumo abrán n networks auxiliares.

Luego, también sabemos que la complejidad de construir un network auxiliar es la de BFS: O(m) por lo tanto tenemos que la complejidad de un algoritmo tipo dinic es

$$n * (O(m) + CFB) = O(n * (m + CFB))$$

Dinitz

Para ver la complejidad de la versión de dinitz es necesario entonces ver el costo CFB.

CFB en dinitz va a consistir del costo de:

- hacer una búsqueda DFS
- podar antes de cada búsqueda

La búsqueda tiene la propiedad de que nunca vamos a hacer backtracking, ya que hemos podado deadends anteriormente. Por lo tanto el costo de buscar un camino en dinitz es O(n)

Ahora debemos ver cuántos caminos pueden haber y cuánto cuesta podar.

A lo sumo pueden haber m caminos.

Para analizar el costo de podar, debemos analizar no una sola poda sino todas las podas de un CFB en conjunto. De esta forma, podemos ver las podas como dos operaciones separadas:

- revisar los vértices de niveles más altos a más bajos para ver si borrarlos(PV)
- Borrar los vértices que no tengan lado(s) de salida(B).

La compjejidad de cada PV es O(n), pues mira los vértices nada más.

La complejidad de un único B(x) es O(m) pues tiene que borrar a todos los vecinos del vértice x. Pero si lo vemos en conjunto, el costo de todos los B es $\sum_{i=1}^{n} \Gamma(x_i) = 2m$

Por lo tanto la complejidad de CFB nos queda:

$$buscarcaminos + \text{revisar por cadacamino} + \text{borrar vertices revisados}$$

$$O(nm) + O(nm) + O(2m) = O(nm)$$

Por lo tanto la complejidad total de Dinitz es

$$n * (O(m) + CFB) = O(n * (m + CFB)) = O(mn^2)$$

Even

Para analizar la complejidad de even nos conviene dar el pseudocódigo:

```
g = 0
stopflag = 1
while(stopflag)
        p = [s]
        x = s
```

```
        while(x!=t) and (stopflag)
                if R+(x) != empty: avanzar(x)
                else if (x != s): retroceder(x)
                else: stopflag = 0
        if(x == t): incrementar()
return g
```

Una vez encontrado un camino, **incrementar() es O(n)**.
la complejidad de **avanzar() y retroceder() es O(1)**

Si denotamos avanzar como A, retroceder como R e incrementar(e inicializar el la siguiente iteración) como I, entonces cada ejecución de CFB es una palabra que contiene As, Bs e Is.

Cada corrida podemos separarla en segmentos de la forma AAAA...X, donde X es R o I.

En ambos casos (X = R ó I) se borra un lado al final de la palabra, por lo tanto hay a lo sumo m palabras.

La complejidad de cada palabra está dada por la cantidad de As del principio y la complejidad de X.

Como máximo, podemos tener n As

Si X = R, entonces tenemos $O(n)$ por cada palabra
Si X = I, entonces tenemos $O(n+n) = O(n)$ por cada palabra

Como por cada corrida hay a lo sumo m palabras y el costo de cada una es O(n), tenemos que la complejidad de CFB en dinic even es O(mn)

Por lo tanto la complejidad de dinic-even es $O(mn^2)$

Comp-Wave

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

Estructura

- Dar pseudocódigo(IMPORTANTE SE BORRAN VERTICES DESPUES DE ENVAR/DEVOLVER EN CIERTOS CASOS)
- Acotar cantidad de olas: **siempre bloqueamos a alguien en la ida**
- Separar en casos de envío y devolución (vértice a vértice)
- Acotar ocurrencias de cada caso
- Concluir la complejidad

Prueba

Primero doy el pseudocódigo:

```
g = 0

# Inicializacion
for x in E:
        D(x) = 0 # D(x) es in(x) - out(x) "Desbalanceo hacia adelante"
        B(x) = 0 # B(x) = "esta bloqueado?"
for x in R+(s):
        g(sx) = c(sx)
        D(x) = c(sx)
        D(s) = -c(sx)
        M(x) = {s} # Registramos a s como alguien que me envia

# Bucle principal

while (D(t) + D(s) != 0)
        # Forward wave
        for x in range(s+1, t-1)
                if(D(x) > 0): FB(x)

        # Backward wave
        for x in range(s+1, t-1)
                if(D(x) > 0 and B(x)): BB(x)


# Forward Balance "Enviar flujo"
def FB(x):
        while(D(x) > 0) and (R+(x).notEmpty())
                y = R+(x).get()
                if(B(y)): R+(x) -= y # Si esta bloqueado ya no le mandamos
                else:
                        A = min(D(x),  c(xy) - g(xy)) # Cantidad a enviar,
toda la que pueda dentro de mi desbalanceo
                        # Actualizar flujo y desbalanceo
                        g(xy) += A
                        D(x) -= A
                        D(y) += A
                        M(y) = M(y) + x
                        # Quitar de vecinos si esta lleno el lado
                        if(g(xy) == c(xy)): R+(x) -= y
        if(D(x) > 0): B(x) = 1 # si no pudimos balancear, estamos bloqueados
```

```
# Backward Balance "Devolver flujo"
def BB(x):
        while(D(x) > 0)
                y = M(x).get()
                A = min(D(x),  g(yx)) # Cantidad a devolver, toda la que
pueda dentro de mi desbalanceo
                g(xy) -= A
                D(x) -= A
                D(y) += A
                if(g(yx) == 0): M(x) -= y # Si le devolvi todo, ya no me
manda
```

La complejidad va a estar dada por:

$$S + P + V + Q$$

donde
$S =$ cantidad de veces que enviamos de un x a un y y lo saturamos
$P =$ cantidad de veces que enviamos de un x a un y y no lo saturamos
$V =$ cantidad de veces que devolvemos de un x a un y y lo vaciamos
$Q =$ cantidad de veces que devolvemos de un x a un y y no lo vaciamos

Cantidad de olas

- Cuando vamos hacia adelante, siempre bloqueamos a un vértice. De no ser el caso, sería la última ola(pues si quedan todos balanceados, terminamos). Por lo tanto hay $O(n)$ olas.
- Solo podemos ir hacia atras luego de ir hacia adelante, así que también hay $O(n)$ olas hacia atrás

S

Para S, sabemos que FB elimina a y luego de enviarle flujo si lo hemos saturado. Esto es correcto ya que no volvemos a envar flujo por ahí(está lleno el lado). Si en algún momento se vaciara por un BB desde y, entonces y estaría bloqueado y tampoco habría que enviarle nada por ese lado. **Por lo tanto** $S \leq m$

V

Al igual que para S, cuando devolvemos por un lado y este lado se vacía, entonces borramos a y de $M(x)$. En principio podríamos ver a y devuelta en $M(x)$ si este le volviera a envar; pero esto no es posible ya que y no le enviaría nada a un vértice bloqueado como. **Por lo tanto** $V \leq m$

P

Cuando hacemos FB, solo el último lado que vemos es capaz de no ser saturado y terminar de balancear D(x), **hay a lo sumo 1 P por Forward Balance**. Por lo tanto $P \leq FB$. Cuál es la cantidad de FB? A lo sumo $(n-2)$ por ola. P es entonces $(n-2 = *O(n) = O(n^2)$

Q

Cuando hacemos BB, solo un lado de los que miramos es capaz de no ser vaciado y por lo tanto hay uno solo(a lo sumo) por ola hacia atras por vértice. Q es entonces
$(n-2 = *O(n) = O(n^2)$

Complejidad total CFB:

$$S + V + P + Q = O(2m + 2n^2) = O(n^2)$$

Complejidad total Wave:

$$O(n) * O(n^2) = O(n^3)$$

Networks Auxiliares aumentan

Dados dos networks auxiliares sucesivos, NA y NA', ya tenemos por la prueba de Edmonds-Karp que $d_f(s,x) \leq d_{f'}(s,x)$. Ahora queremos probar que ahí vale siempre un $<$.

Estructura

- Asumir que en ambos networks llegamos a t, sino es trivial que la distancia aumenta.
- Hay un camino en NA' que no está en NA, y hay dos formas de que eso pase
- Vemos para cada forma que la distancia aumenta
- Forma 1: falta un vértice
  El vértice que falta en NA tiene que tener una distancia f mayor o igual a t.
  En NA', tiene una distancia estrictamente menor a t.
  Por lo tanto crece la distancia hasta t.
- Forma 1: falta un lado
  Vemos que si falta el lado xi -> xi+1, es porque d(xi+1) <= d(xi). Luego tenemos d(xi+1) <= d(xi) <= d'(xi) = i < i+1 = d'(xi+1). Como la distancia hasta xi+1 crece estrictamente luego de actualizar el flujo, pero sigue siendo parte de un camino de menor longitud hasta t, entonces la distancia a t crece.

Prueba

Si $t \notin NA'$ terminamos.

Asumamos entonces que $t \in NA'$:

- Hay un camino dirigido $x0, x1, \ldots, x_{r-1}x_r$
- Este camino no pertenece a NA, pues sino hubiera sido saturado y no podría estar en NA'

Como el camino no está en NA, puede ser por dos razones:

- Algún vértice del camino no está
- Algún lado no está.

Caso 1: falta un vértice

Si un vértice x_i no está, entonces sabemos que $x_i \neq t$

- La única forma de que x_i no esté en NA es que $d_f(t) \leq d_f(xi)$
- Al ser NA' un layered network, tenemos que $d_{f'}(xi) = i$
- Como $x_i \neq x_r = t$, tenemos que $d_f(t) \leq d_f(xi) \leq d_{f'}(xi) = i < r$
  Por lo tanto hemos probado que en este caso que la longitud aumenta.

Caso 2: falta un lado

Digamos que el primer lado que falta es $\overrightarrow{x_i x_{i+1}}$.

El lado $\overrightarrow{x_i x_{i+1}}$ puede no estar porque el lado que representa en el network original esta saturado y es forward o vacio y es backward. En cuyo caso, tenemos que en la construcción de NA', usamos el lado $\overrightarrow{x_{x+1} x_i}$ y claramente $df(x_{i+1}) < d(x_i)$.

La otra opción que nos queda entonces es que el lado no esté porque $df(x_{i+1}) = d(x_i)$.
De todas formas vale:

$$df(x_{i+1}) \leq d(x_i)$$

Entonces tenemos

$$d_f(x_{i+1}) \leq d_f(x_i) \leq d_{f'}(x_i) = i < i + 1 = d_{f'}(x_{i+1})$$

$$d_f(x_{i+1}) < d_{f'}(x_{i+1})$$

Entonces es claro que la distancia aumenta, pues

$$d_f(t) = d_f(x_{i+1}) + b_f(x_{i+1})$$

$$\leq d_f(x_{i+1}) + b_{f'}(x_{i+1})$$

$$< d_{f'}(x_{i+1}) + b_{f'}(x_{i+1})$$

$$= d_{f'}(t)$$

Max Flow - Min Cut

Probar que el valor de todo flujo es menor o igual que la capacidad de todo corte y que si f es un flujo, entonces las siguientes afirmaciones son equivalentes:
i) Existe un corte S tal que $v(f) = cap(S) = c(S, \overline{S}))$. (y en este caso, S es minimal)
ii) f es maximal.
iii) No existen f-caminos aumentantes.
(puede usar sin necesidad de probarlo que si f es flujo y S es corte entonces
$v(f) = f(S, \overline{S}) - f(\overline{S}, S))$

Estructura

Primero probamos $v(f) \leq cap(S), \forall f, S : \text{f flujo y S corte}$ usando $v(f) = f(S, \overline{S}) - f(\overline{S}, S))$ (es cortito)

Después probamos la equivalencia de esta forma:
$1 => 2 => 3 => 1$
Y queda probada la ida y la vuelta entre todos.

- 1=>2: $v(f) = cap(S) \leq v(g) \forall g \text{ flujo}$
- 2=>3: contrarrecíproca - existiría un flujo de valor más grande si existiera un f-camino aumentante
- 3=1: Construimos S y vemos que los limítrofes están saturados y que en $\bar{S}$ los limítrofes no envían nada hacia atrás. Por lo tanto $f(S, \bar{S}) - f(\bar{S}, S) = c(S, \bar{S}) - 0 = cap(S)$

Prueba

1 => 2

Fácil, $v(f) = cap(S) \leq v(g) \forall g \text{ flujo}$

2 => 3

Por contrarrecíproca. Si existe un camino p f-aumentante, entonces habría un flujo g tal que

$$g(\overrightarrow{xy}) = \begin{cases} f(\overrightarrow{xy}) & \text{si } \overrightarrow{xy} \in p \\ f(\overrightarrow{xy}) + \epsilon & \text{caso contrario} \end{cases}$$

En particular habrían lados $\overrightarrow{sx}$ y $\overrightarrow{yt}$ en p y por lo tanto $v(g) = v(f) + epison$. Entonces no sería maximal f.

3 => 1

Si no hay caminos f-aumentantes, probaremos que existe un corte S tal que cap(S) = v(f).

Construimos S de la siguiente manera:

$$S = \{s\} \cup \{\text{x : existe camino f-aumentante de s a x}\}$$

- Como vale 3, $t \notin S$. S es corte.
- tenemos que v(f) = $f(S, \bar{S}) - f(\bar{S}, S)$
  Para los x "limítrofes" en S, todas sus salidas están saturadas, sino tendríamos caminos a un nivel más allá y no serían limítrofes.
  Los y en $\bar{S}$ no envian nada hacia ningún x en S, de lo contrario, como vimos antes, habría un camino f-aumentante hacia y y por lo tanto $y \in S$, lo que sería absurdo.

Por lo tanto $v(f) = c(S, \bar{s}) - 0 = cap(S)$

Para mayor formalidad podemos usar las sumatorias que definen a f(S) y demás; pero lo que decimos de x e y es lo mismo: x tiene todas las salidas saturadas e y no le manda nada a ningún x

2color-polinomial

Dado un grafo G, tomamos un vertice x cualquiera y lo coloreamos con 0. Luego, hacemos BFS y coloreamos cada vértice con 0 ssi su distancia a x par, 1 caso contrario. Luego vemos si este coloreo es propio. Esto hay que hacerlo con un x, y si al terminar BFS, faltan vértices por colorear, entonces hacemos lo mismo con otro x' que falte, hasta terminar.

Si al terminar, el coloreo es propio entonces G es bipartito, sino no.

BFS es O(m) -> el algoritmo es polinomial

Este algoritmo es correcto, ahora lo probamos:

si al terminar de colorear, encontramos que hay 2 vértices vecinos v y z que tienen el mismo color, tendremos que el ciclo w ... v ... z ... w(donde w es el último ancestro común de v y w en el árbol BFS) tiene distancia:

$$d(w, v) + d(v, z) + d(z, w)$$

$$d(w, v) + 1 + d(z, w)$$

si tomamos módulo 2 a esta suma:

$$d(w, v) + 1 + d(z, w) mod 2$$

$$(d(w, v) mod 2) + 1 + (d(z, w) mod 2) mod 2$$

$$(d(w,v) mod 2) + 1 + (d(w,v) mod 2) mod 2$$

$$2(d(w,v) mod 2) + 1 mod 2$$

$$1 mod 2$$

Es decir, tenemos un ciclo impar.

Resumiendo: si el coloreo es propio entonces G es bipartito, si el coloreo no es propio, entonces tenemos un ciclo impar como certificado de que no es bipartito.

Teoremas Matching:

Teorema Hall

Sea $G = (X \cup Y, E)$ un grafo bipartito,

$$|S| \leq \Gamma(S), \forall S \subset X => \exists \text{ matching completo}$$

Estructura de prueba

Vemos la contrarecíproca y usamos lo que sabemos de Edmonds-Karp.

- Separamos el corte minimal entre los de un lado y el otro
- Vemos que E-K agrega primero a algunos de la izquierda(S0) y esos agregan a sus vecinos a la derecha(T1).
- Luego vemos que esos vecinos de la derecha agregarán exáctamente la misma cantidad de vértices a la izquierda(S1). Esto se da generalmente.
- entonces hacemos las cuentas y vemos que
  $|S| = |S_0| + |S - S_0| = |S_0| + |T| = |S_0| + |\Gamma(s)| > |\Gamma(s)|$

Prueba

Asumiendo que no hay matching completo, y sea C el corte minimal que obtenemos al correr Edmonds-Karp, definimos:

$$S = C \cap X$$

$$T = C \cap Y$$

Si pensamos en la útlima cola E-K que construye el algoritmo, tenemos S_0, los vértices en S agregados por s, luego T_1, los vértices agregados por los vértices en S_0 y en general S_i agregado por T_i que a su vez es agregado por S_{i-1}.

Observación 1: todos los T_i agregan tantos vértices a la cola como vértices tenga, por lo tanto $|T_i| = |S_i|, i > 1$

Observación 2: S es la unión de los S_i T es la unión de los T_i y estos son disjuntos entre si.

Entonces tenemos:

$$|S| = |S_0 \cup S_1, \cup . . \cup S_k|$$
$$= |S_0| + (|S_1| + \ldots + |S_k|)$$
$$= |S_0| + (|T_1| + \ldots + |T_k|)$$
$$= |S_0| + (|T_1 \cup \ldots \cup T_k|)$$
$$= |S_0| + (|T|)$$
$$= |S_0| + (|\Gamma(S)|)$$
$$> |\Gamma(S)|$$

Queda probado por contrarrecíproca

Teorema Matrimonio

Estructura

- Planteamos un subconjunto cualquiera $W \subseteq X$
- Planteamos el conjunto de lados $E_W = \{zw : w \in W\}$
- Vemos que para este conjunto, $|E_W| = \sum\{d(w) : w \in W\} = \Delta * |W|$
- Vemos que todo esto se cumple tambien para $W \subseteq Y$
- Vemos que si $W = X, E_X = E$ y $|E| = \Delta * |X| = \Delta * |Y|$ y por lo tanto $|X| = |Y|$
- Ahora basta probar que existe un matching completo, y será también perfecto.
- Vemos que si tomamos $S \subseteq X$, entonces cada lado $l \in E_S$ tambien cumple $l \in E_\Gamma(S)$
- Por lo tanto $E_S \subseteq E_\Gamma(S) => |E_S| \leq |E_\Gamma(S)|$
- Por lo tanto $\Delta * |E_S| \leq \Delta * |\Gamma(S)| => |E_S| \leq |\Gamma(S)|$
- Se cumple la condición de Hall y por lo tanto hay matching perfecto.

Prueba

Sea $W \subseteq X$, y $E_W = zw : w \in W$, tenemos que $|E_W| = \sum_{w \in W} d(w)$, pues como no hay lados entre vértices de W, no estamos contando lados dos veces. Como G es regular, tenemos también que $|E_W| = \Delta|W|$

Observemos que esto **también se cumple para cualquier W subconjunto de Y**, pues vale todo lo que dijimos hasta ahora.

Si consideramos que todos los lados en E son de la forma xy, con $x \in X$, vemos que $E_X = E$ y por ende vemos que $|E| = \Delta|X|$

Devuelta, lo mismo vale para Y, entonces $|E| = \Delta|Y|$ y por lo tanto $|X| = |Y|$

Tenemos entonces que si tuvieramos un matching completo de X a Y, ese matching sería también perfecto. Basta ver que se cumple la condición de Hall.

Consideremos $S \subseteq X$, tenemos que para todo lado $l \in E_S$, este lado es de la forma xy, donde $x \in S, y \in \Gamma(S)$, y por lo tanto $l \in E_{\Gamma(S)}$. Entonces vemos que $E_S \subseteq E_{\Gamma(S)}$

Entonces tenemos:

$$|E_S| \leq |E_\Gamma(S)|$$
$$\Delta|S| \leq \Delta|\Gamma(S)|$$
$$|S| \leq |\Gamma(S)|$$

Concluimos que existe matching perfecto.

Teorema Indice Cromatico

Sea G un grafo bipartito, $\chi'(G) = \Delta$

Estructura

- Primero probamos, usando el teorema del matrimonio y por inducción, que se puede colorear con Delta colores un grafo bipartito regular.
- Luego mostramos que cualquier grafo bipartito G con cierto Delta puede incluirse dentro de un grafo bipartito H regular con el mismo Delta.
- Esto lo hacemos dando la siguiente construcción de un grafo G', que tiene $\delta(G') = \delta(G)$, y de esa forma mostramos que podemos hacer que eventualmente lleguemos a un grafo H, con $\delta(H) = \Delta(H) = \Delta(G)$ que será regular.
- Como G es subgrafo de H, se puede colorear G con Delta de G colores

"teorema-indice-cromatico 2023-06-29 11.50.17.excalidraw" is not created yet. Click to create.

Prueba

Primero probamos que un grafo bipartito regular G se puede colorear con Delta colores.

Usamos inducción sobre Delta.

Si $\Delta = 1$, entonces todos los lados no comparten vértices, pues sino para algún vértice compartido x, d(x) > 1. Entonces podemos pintar a todos con el mismo color.

Ahora para el caso inductivo, como G es un grafo bipartito regular, tenemos que existe un matching perfecto. Si tomamos los lados de un matching perfecto y los quitamos, entonces tenemos un grafo bipartito regular con Delta = Delta(G) - 1.

Tenemos por hipótesis inductiva, que este grafo reducido se puede colorear con Delta(G)-1 colores. Si ahora usamos ese coloreo para colorear G, y coloreamos los lados del matching con un nuevo color, entonces tenemos un coloreo de lados propio, pues los lados del matching no comparten vértices. Queda probado el caso inductivo.

Ahora para el caso regular, basta con probar que todo grafo bipartito G se puede incluir en un grafo bipartito G' con $\delta(G') = \delta(G) + 1$. Iterando esta construcción, podemos llegar a un grafo $H : \delta(H) = \Delta(H) = \Delta(G)$, es decir un grafo bipartito regular con igual Delta que G, con G subgrafo de este. De esta forma quedaría probado que se puede colorear G con Delta colores.

Sea G grafo tal que $G = (X \cup Y, E)$, podemos definir G' como:

$$G' = (X' \cup Y', E')$$

donde

$$X' = X \cup Y^*, Y' = Y \cup X^*$$

$$E' = E \cup \{x'y' : xy \in E\} \cup \{xx' : x \in X \wedge d(x) < \Delta(G)\} \cup \{yy' : y \in Y \wedge d(y) < \Delta(G)\}$$

donde X *e Y* son "copias" de X e Y(notese también que x' es la copia del vértice x en X)

Este nuevo grafo G' tiene como subgrafo a G. Pero tamién tiene que para cada vértice de G cuyo grado era menor a Delta, ahora su grado es aumentado en 1. También, como no hay lados entre X' e Y', tenemos que G es bipartito.

Concluimos que existe H regular con Delta igual a Delta de G y con G subgrafo de este. Por lo tanto se puede colorear G con Delta(G) colores.

Comp-Hungaro

Probar la complejidad $O(n^4)$ del algoritmo Hungaro y dar una idea de como se la puede reducir a $O(n^3)$.

Prueba

Idea global del costo

Con el método que hacemos en papel, usamos E-K sobre la matriz para ir aumentando el matching inicial. Esto lo vamos a hacer a los sumo n veces(pues hay n lados en el matching perfecto que buscamos.)

Como el matching inicial cuesta $O(n^2)$(que es lo que cuests restarle el mínimo a cada fila y luego el mínimo a cada columna y luego obtener el matching inicial de ceros.) y **luego** haremos $O(n)$ extensiones a este matching, el costo del húngaro es a priori $O(n^2) + O(n)$, si ignoramos el costo de extender el matching(incluyendo los cambios de matriz).

¿Cual es la complejidad de extender en un lado el matching? Si solo pensamos en la búsqueda de un camino aumentante de tamaño mínimo, tenemos que el costo es $O(m) = O(n^2)$(costo de BFS).

Hasta ahora, la complejidad total del húngaro sería $O(n^2) + O(n) * O(n^2) = O(n^3)$

Pero todavía tenemos que tener en cuenta que para cada vez que extendemos en 1 el matching, también es posible que tengamos que agregar un lado haciendo una serie de cambios de matriz.

Sea CM el costo de un cambio de matriz y T la máxima cantidad de cambios de matriz necesarios para agregar un lado, **el costo total del algoritmo es**:

$$O(n^2) + O(n) * (O(n^2) + CM * T)$$

CM

CM consiste en

- Calcular la constante m
- Restar m de S
- Sumarlo m a $\Gamma(S)$

El cálculo de m consiste de calcular el mínimo de potencialmente todos los elementos de la matriz, así que su complejidad es $O(n^2)$

Restarlo de las filas es $O(n) * \text{filas etiquetadas} = O(n^2)$

Sumarlo a $\Gamma(s)$ también es $O(n^2)$

Por lo tanto la complejidad de CM es $O(n^2)$

T

Ahora veamos de acotar T.

Propiedad clave: después de un cambio de matriz, o crecen las filas etiquetadas o extendemos el matching(o ambas).

Eso lo podemos ver si analizamos que pasa cuando calculamos que $m = C_{x,v}$: tendremos un nuevo 0 en la entrada $C_{x,v}$ y por lo tanto cuando revisemos las filas, x etiquetará a v. Si v está libre, extendemos el matching. Si v no está libre, entonces agrega a alguna fila z y quizá extendemos el matching. Si terminamos no pudiendo extender el matching, entonces tenemos que S crece en al menos 1(el z que agregamos).

Con esta propiedad tenemos que $T \leq n$

Conclusión

Entonces, reemplazando:

$$O(n^2) + O(n) * (O(n^2) + CM * T)$$

$$O(n^2) + O(n) * (O(n^2) + O(n^2) * O(n))$$

$$O(n^2) + O(n^3) + O(n^4) = O(n^4)$$

Complejidad mejorada: n^3

Dada la cuenta de arriba, vemos que si CM costara O(n), entonces tendríamos una complejidad de n^3. Para lograr esto, podemos hacer que el cálculo de m sea O(n) y también lograr que restar a las filas S y sumar a las $\overline{\Gamma(S)}$ sea O(n).

Para lo primero, podemos tener pre-calculado el mínimo de cada fila de S, por ejemplo, esto se podría hacer cuando se está revisando una fila en la búsqueda anterior al cambio de matriz.

Por otro lado podemos en lugar de restar/sumar a todas las entradas, mantener un array con los valores que habría que sumarle/restarle a cada fila y columna. Esto nos da un costo O(n). Luego al buscar 0's, lugar de la guarda $C_{x,v} == 0$ tendríamos
$C_{x,v} - ArrayFilas[x] + ArrayColumnas[v] == 0$

La complejidad queda:

$$O(n^2) + O(n) * (O(n^2) + CM * T)$$

$$O(n^2) + O(n) * (O(n^2) + O(n) * O(n))$$

$$O(n^2) + O(n^3) + O(n^3) = O(n^3)$$

Problemon inconcluso

Según las filminas del profe, la cuenta de la complejidad total del húngaro es:

$$O(n^2) + O(n) * (O(n^2) + CM * T)$$

Pero, en la última parte, estamos considerando que hacemos una única búsqueda $O(n^2)$. Esto no es cierto y yo creo que la cuenta debería ser:

$$O(n^2) + O(n) * ((O(n^2) + CM) * T)$$

con esta cuenta estamos contando una búsqueda luego de cada cambio de matriz, que es lo que hacemos en realidad.

El problema no está en la complejidad n^4, nos da el mismo resultado. **El problema está en la complejidad mejorada** que al aplicar esta nueva cuenta, no nos da de todas formas

Reemplazando CM por $O(n)$ y todas las otras igual, incluso con nuestras mejoras la cuenta queda:

$$O(n^2) + O(n) * ((O(n^2) + O(n^2)) * O(n))$$

$$O(n^2) + ((O(n^4) + O(n^3))) = O(n^4)$$

La complejidad no mejora.

Teoremas Codigos:

Delta-Columnas-H

Probar que si H es matriz de chequeo de C, entonces;

$$\delta(C) = Min\{j : \exists \text{ un conjunto de j columnas LD de H}\}$$

Sea H matriz de chequeo, entonces C = Nu(H), y por lo tanto si $H \cdot w^t = 0 => w\ in C$

Sea M un conjunto de columnas de H de cardinal mínimo tal que M es LD.

Observemos primero que al ser de tamaño mínimo, hay una única combinación lineal no trivial de las columnas en M que es 0: donde todos los coeficientes son 1. Si suponemos que existe una combinación lineal con algún coeficiente nulo, digamos el i-ésimo, entonces tenemos que $M - H^{(i)}$ también es LD, lo que es absurdo pues es de cardinal más chico que M.

Sea w la palabra con un uno en cada entrada i-ésima sólo si $H^{(i)} \in M$, tenemos por lo observación de arriba que $H \cdot w^t = \sum M = 0$

Tenemos que $|M| = |w|$ y $w \in C$. Sabemos que $\delta(C) \leq |w|$Asumamos que $\delta < |w|$. Si este fuera el caso, existiría una palabra w' en C con menor cantidad de 1s, y tendríamos entonces que existe un subconjunto M' de columnas de H tal que $H \cdot w^t = \sum M' = 0$ y $|M'| = |w'|$. Tendríamos entonces que existe un subcojunto de columnas LD de cardinal menor a M.

Concluyo por contradicción que

$$\delta = |w| = |M| = Min\{j : \exists \text{ un conjunto de j columnas LD de H}\}$$

Teorema Ciclicos

Sea C un código cíclico de dimensión k y longitud n y sea $g(x)$ su polinomio generador. Probar que:
i) C esta formado por los multiplos de g(x) de grado menor que n:

$$C = p(x) : gr(p) < n \wedge g(x)|p(x)$$

ii) $C = \{v(x) \odot g(x) : \text{v es un polinomio cualquiera}\}$

iii) $gr(g(x)) = n - k$
iv) $g(x)|1 + x^n$

Estructura

Probar primero que el conjunto dado en ii) esta en C, luego que C esta en el conjunto dado en i) y luego que este último está en el de ii). Ayuda memoria en cada caso:

Prueba

i) y ii)

sean
$C1 = p(x) : gr(p) < n \wedge g(x)|p(x)$
$C2 = \{v(x) \odot g(x) : \text{v es un polinomio cualquiera}\}$

Tenemos la propiedad:

$$w \in C => w \odot v \in C$$

por lo tanto, tenemos que $C2 \subseteq C$

Veamos $C \subseteq C1$

Sea $p(x) \in C$, tenemos que $gr(p(x)) < n$.

Aplicando el algoritmo de división de polinomios, tenemos:

$p(x) = q(x)g(x) + r(x), gr(r) < gr(q) < n$
por lo tanto
$r(x) = p(x) + q(x)g(x)$

Como gr(p) < n , $p(x) = p(x) mod (1 + x^n)$
Como gr(r) < n, $r(x) = r(x) mod (1 + x^n)$

Entonces
$r(x) = r(x) mod (1 + x^n)$
$r(x) = p(x) + q(x)g(x) mod (1 + x^n)$
$r(x) = p(x) mod (1 + x^n) + q(x)g(x) mod (1 + x^n)$
$r(x) = p(x) mod (1 + x^n) + q(x) \odot g(x)$
$r(x) = p(x) + q(x) \odot g(x)$

como ambos términos son parte del código, el resto es parte del código.

Pero si es parte del código y tenemos que $gr(r) < gr(g)$ entonces $gr(r) = 0$, pues g es el polinomio no nulo de menor grado en C.

Concluimos entonces que $C \subseteq C1$

Ahora resta ver que $C1 \subseteq C2$

Digamos que $p(x) \in C1$, entonces
$p(x) = p(x) mod (1 + x^n) = q(x)g(x) mod (1 + x^n) = q(x) \odot g(x)$
Por lo tanto $C1 \subseteq C2$

iii)

Sea p cualquier palabra del codigo, tenemos por i) que
$gr(p) = gr(q) + gr(g), \text{para algún polinomio q}$ y también $gr(q) + gr(g) < n$

la cantidad de posibles p van a ser entonces la cantidad de q que cumplan $gr(q) < n - gr(g)$

Esta cantidad es $2^{n-gr(g)}$. Y como C es lineal, entonces la dimensión es $n - gr(g)$ y por lo tanto:

$$gr(g(x)) = n - k$$

iv)

Aplicamos el algoritmo de división y despejamos el resto r(x):

$$1 + x^n = q(x)g(x) + r(x)$$

$$r(x) = q(x)g(x) + (1 + x^n)$$

Si tomamos mod (1+x^n) tenemos

$$r(x) mod (1 + x^n) = q(x)g(x) mod (1 + x^n)$$

$$r(x) = q(x) \odot g(x)$$

por lo tanto tenemos que r está en el código y que $gr(r) < gr(g)$, y por lo tanto r es 0.

Idea de la propiedad del principio

No hace falta probar esto en esa demostración, pero una intucición es que si rotamos una palabra en un código cíclico entonces la palabra rotada también es parte del código. Multiplicar modularmente por una palabra no es más que por cada término no nulo de la palabra por la que multiplicamos, rotar cierta cantidad de veces y luego sumar; al ser nuestro código lineal, la suma también está en el código.

Teoremas P-NP:

3SAT-NP-Completo

Primero probamos que (≤ 3) SAT se reduce a 3-SAT, luego mostramos que SAT se reduce a (≤ 3) SAT y por lo tanto transitividad, que 3-SAT es np-completo.

Primero reducimos ≤ 3 SAT a 3-SAT. Para esto por cada disyunción D en una instancia de <= 3 sat, debemos conseguir una disyunción o serie de disyunciones que se satisfacen en el mismo universo o asignación de variables. Para el caso de que D tenga 3 literales, esto es trivial. Basta ver que hacer con los casos de 1 y 2 literales.

Si D tiene 2 literales

Entonces damos una expresión

$$E = (D \vee x_D) \wedge (D \vee \overline{x_D})$$

donde x_D es es un literal nuevo no presente en B.

Veamos que si E es satisfacible, entonces D también lo es.

$$E = (D \vee x_D) \wedge (D \vee \overline{x_D})$$

$$\overset{Distributividad}{=} D \vee (x_D \wedge \overline{x_D})$$

$$\overset{Nocontardiccion}{=} D \vee (False)$$

$$\overset{Neutrodeladisyuncion}{=} D$$

Si D tiene 1 literal

Entonces construimos

$$E = (D \vee x_D \vee y_D) \wedge (D \vee x_D \vee \overline{y_D}) \wedge (D \vee \overline{x_D} \vee y_D) \wedge (D \vee \overline{x_D} \vee \overline{y_D})$$

Veamos que es equivalente a D:

$$E = (D \vee x_D \vee y_D) \wedge (D \vee x_D \vee \overline{y_D}) \wedge (D \vee \overline{x_D} \vee y_D) \wedge (D \vee \overline{x_D} \vee \overline{y_D})$$

$$\overset{Distributividad}{=} D \vee ((x_D \vee y_D) \wedge (x_D \vee \overline{y_D}) \wedge (\overline{x_D} \vee y_D) \wedge (\overline{x_D} \vee \overline{y_D}))$$

$$\overset{AsociatividadyDistributividad}{=} D \vee ((x_D \vee (y_D \wedge \overline{y_D})) \wedge (\overline{x_D} \vee (y_D \wedge \overline{y_D}))$$

$$\overset{Nocontradiccion+neutrodeladisyuncion}{=} D \vee ((x_D) \wedge (\overline{x_D})$$

$$\overset{Idem}{=} D$$

Listo, sabemos entonces que si tenemos una expresión B en forma (<= 3)CNF entonces podemos dar una expresión equivalente B' en forma 3CNF.

Sea n la cantidad de D_is en B, tenemos que el algoritmo para pasar de B a B' es $O(n)$

Queda probado que $(\leq 3) - SAT \leq_p 3 - SAT$

Ahora basta probar que

$$SAT \leq_p (\leq 3) - SAT$$

Hacemos un procedimiento similar, usando variables nuevas para construir expresiones E por cada expresión D en B, para así construir $B' = E_1 \wedge E_2 \wedge .. \wedge E_m$
Si en D estan las variables $l_1, l_2, .., l_k$, agregaremos las variables $y_1, y_2, .., y_{k-3}$

si $k \leq 3$ tomamos simplemente

$$E = D$$

pero si $k \geq 4$ tomamos

$$E = (l_1 \vee l_2 \vee y_1) \wedge (l_3)$$