

# Algoritmos de aproximación: Presentación

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Motivación

## Muchos problemas de uso práctico

Se han demostrado como NP-Completo

## Sin embargo, su importancia es elevada

Y no pueden dejarse de lado

# Motivación (cont.)

## Si la instancia del problema es pequeña

Se puede resolver óptimamente aun siendo no polinomiales

## Si la instancia cumple con ciertas características

Se puede encontrar un algoritmo polinómico (ej 2-SAT)

## Sino

Tratar de encontrar una solución aproximada al óptimo en tiempo polinomial

# Relación de aproximación $\rho(n)$

## Sea

Un problema de optimización P (de maximización o minimización)

Un algoritmo A que resuelve P de forma aproximada

## Diremos

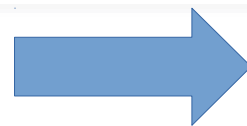
Que A tiene una relación de aproximación  $\rho(n)$

## Si para cualquier I instancia de tamaño n

La solución producida por  $C=A(I)$

## Esta dentro de un factor $\rho(n)$

de la solución óptima  $C^*$



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

# Relación de aproximación $\rho(n)$

## Llamaremos al algoritmo A

Un  $\rho(n)$ -algoritmo de aproximación

## La aproximación es siempre peor o igual al optimo

$$\rho(n) \geq 1$$

## Aunque (ojo!) algunos autores definen

$\rho(n) \leq 1$  para problemas de maximización

$\rho(n) \geq 1$  para problemas de minimización

# Esquema de aproximación

## Existen algoritmos de aproximación

Que permiten en su ejecución adicionar un parámetro adicional

### Su valor

permitirá mejorar la relación de aproximación

### A costo de

aumentar el tiempo de ejecución del mismo

# Esquema de aproximación (cont.)

## Sea

Un problema de optimización  $P$

Un algoritmo  $A$  que resuelve  $P$  de forma aproximada

Un parámetro  $\varepsilon > 0$  fijo

## Diremos

Que  $A$  tiene un esquema de aproximación  $(1+\varepsilon)$

## Si para cualquier $I$ instancia de tamaño $n$

La solución producida por  $C=A(I)$

## Esta dentro de un factor $(1+\varepsilon)$ del optimo

# Esquema de aproximación polinomial en tiempo

## Un esquema de aproximación

Es polinomial en tiempo

**Si**

Para cualquier  $\epsilon > 0$  fijo

**Se ejecuta**

En tiempo polinomial en función a  $n$

**Ejemplo**

$O(n^{2/\epsilon}) \leftarrow$  cuanto mas pequeño  $\epsilon$ , mas costosa la ejecución



# Esquema de aproximación totalmente polinomial en tiempo

## Un esquema de aproximación

Es totalmente polinomial en tiempo

**Si**

Para cualquier  $\varepsilon > 0$  fijo

**Se ejecuta**

En tiempo polinomial en función a  $n$  y de  $1/\varepsilon$

**Ejemplo**

$$O((1/\varepsilon)^2 n^3)$$

# Construcción de algoritmos de aproximación

## Existen diversas técnicas

para construir algoritmos de aproximación

## Entre ellos

Uso de algoritmos greedy

Pricing method (primal-dual technique)

Programación lineal y redondeo

Programación dinámica y redondeada de la instancia

# Desafíos

## Para cada

Algoritmo de aproximación presentado

## Deberemos probar

su relación o esquema de aproximación

Su complejidad temporal en función de los parámetros

**... sin conocer realmente cual es su optimo!**

(lo haremos con paciencia y detalle)

# Imposibilidad de aproximación

## Existen ciertos problemas

Que dada su naturaleza

## No permiten

Generar algoritmos aproximados

## Un ejemplo

Problema del viajante general (sin desigualdad triangular, ni simetría)

Solo existe un algoritmo de aproximación si  $P=NP$  !



Presentación realizada en Julio de 2020

## Balanceo de Carga

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Problema del balanceo de carga

## Tenemos:

- Un Set de  $m$  maquinas  $M_1, M_2, M_m$
- Un Set de  $n$  tareas
- Cada tarea  $j$  requiere  $T_j$  de tiempo de procesamiento.

**Objetivo: Asignar las tareas a las maquinas de tal forma que la carga quede balanceada**

(el tiempo asignado a cada maquina sea lo más parejo posible)

# Cómo medir el balanceo?

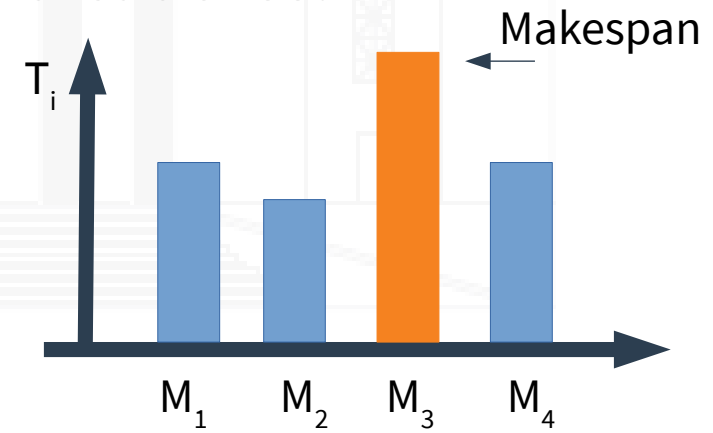
Si llamamos  $A(i)$  al conjunto de tareas asignadas a la maquina  $i$   
Podemos calcular la carga de la maquina  $i$  como:

$$T_i = \sum_{j \in A(i)} t_j$$

Podemos medir el balanceo por diferentes indicadores.

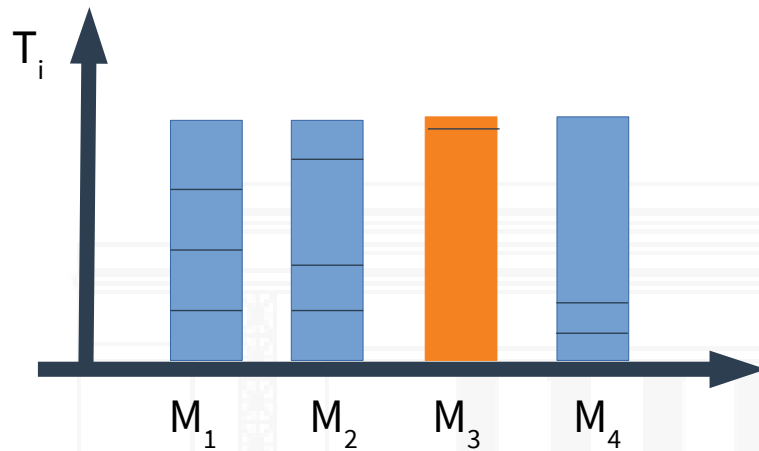
Usaremos:

Makespan :  $\max (T_i)$  para todas las maquinas

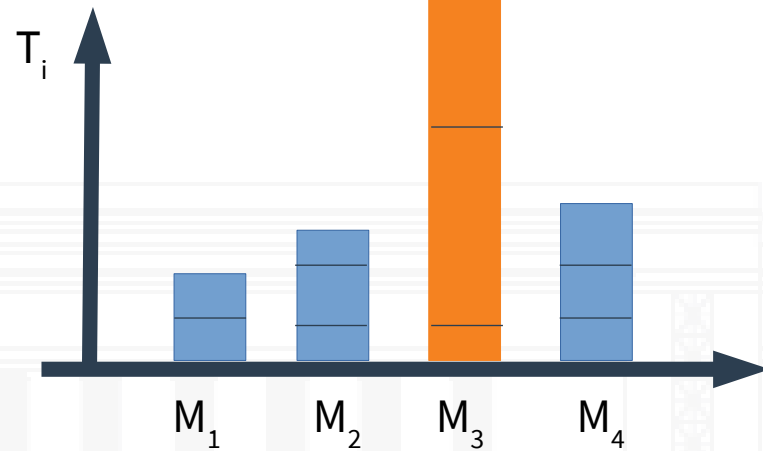




# Asignación de trabajos



Ideal



No deseado

El método para seleccionar la asignación y la naturaleza de los trabajos determinará la programación final de las tareas

Es un problema NP-HARD

# Un primer método greedy

Para cada tarea  $i$ , asignarla a la maquina  $j$  con menor carga en el momento.

Comenzar sin trabajos asignados

Definir  $T_i = 0$  y  $A(i) = \emptyset$  para todas las maquinas  $M_i$

Desde  $j = 1$  a  $n$

Sea  $M_i$  la maquina con menor  $T_k$  ( $k=1$  a  $m$ )

Asignar Tarea  $j$  a maquina  $M_i$

Establecer  $A(i) \leftarrow A(i) \cup \{j\}$

Establecer  $T_i \leftarrow T_i + t_j$

# Análisis del algoritmo

Para determinar cuanto se aleja la solución obtenida de la optima ( $T^*$ ), debemos compararlas.

Pero ... no tenemos la solución optima

Sin embargo, podemos acotarla:

$$T^* \geq \frac{1}{m} \sum_j t_j$$

El optimo es mayor o igual al tiempo promedio total

$$T^* \geq \max_j t_j$$

El optimo es mayor o igual al tiempo del trabajo mas largo

# Análisis del algoritmo (cont.)

## A.1 El algoritmo asigna los trabajos a las máquinas con un makespan $T \leq 2T^*$ .

El ultimo trabajo es asignado a máquina  $M_i$  con mínima carga

Antes de la asignación tendrá  $T_i - t_j$  de carga.

Sabemos que:  $\sum_k T_k \geq m(T_i - t_j)$

La carga en todas las maquinas

$$(T_i - t_j) \leq \frac{1}{m} \sum_k T_k \leq T^*$$

$$t_j \leq T^*$$

$$T_i = (T_i - t_j) + t_j$$

$$T_i \leq 2T^*$$

Acotamos:

$$T^* \geq \frac{1}{m} \sum_j t_j$$

$$T^* \geq \max_j t_j$$



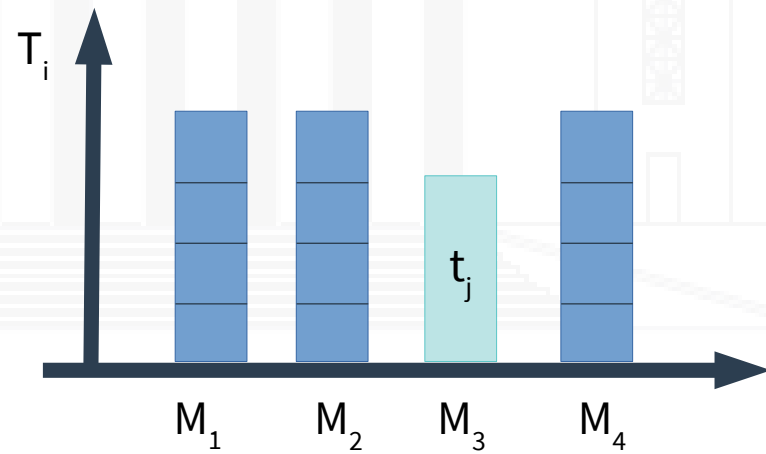
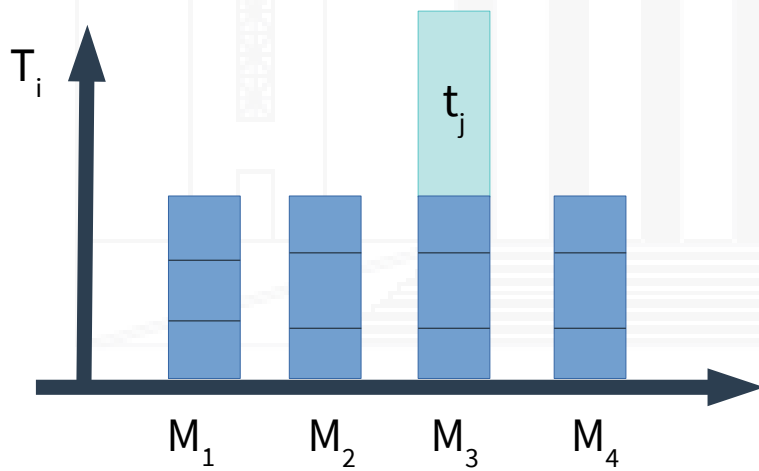
# Podemos mejorar nuestro algoritmo?

## Cuando ocurre el peor caso?

El algoritmo intenta mantener siempre el mayor balance posible.

Si la ultima tarea coincide con aquella de longitud mas grande quedará peor balanceado

Ej: Las  $j-1$  tareas de  $t_x=a$  y  $t_j \gg a$



# Algoritmo de aproximación mejorado

## Procesar primero las tareas mas extensas

Ordenar las tareas.

Comenzar sin trabajos asignados

**Ordenar las tareas de mayor a menor duración**

Definir  $T_i = 0$  y  $A(i) = \emptyset$  para todas las máquinas  $M_i$

Desde  $j = 1$  a  $n$

Sea  $M_i$  la máquina con menor  $T_k$  ( $k=1$  a  $m$ )

Asignar Tarea  $j$  a máquina  $M_i$

Establecer  $A(i) \leftarrow A(i) \cup \{j\}$

Establecer  $T_i \leftarrow T_i + t_j$

# Análisis del algoritmo mejorado

## Si hay $m$ o menos tareas

La solución es optima.

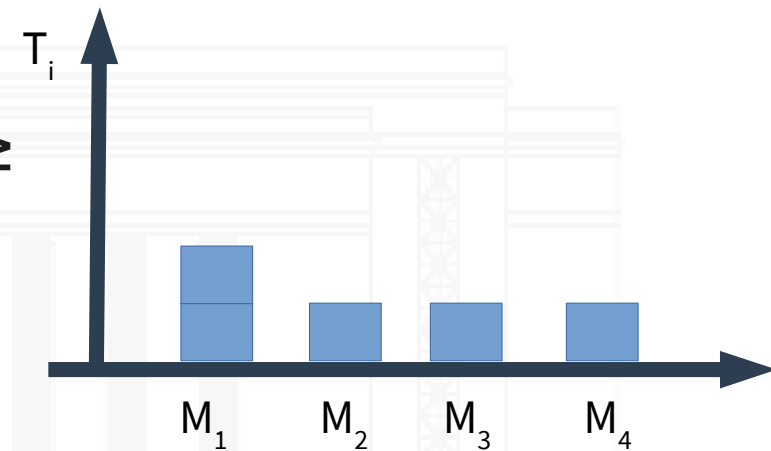
**(A.2) Si hay más de  $m$  tareas, entonces  $T^* \geq 2t_{m+1}$**

Tomemos las primeros  $m+1$  tareas ordenadas por tiempo descendiente.

Hay  $m$  máquinas, por lo tanto solo 1 recibe 2 tareas

En el peor de los casos las primeras  $m+1$  tareas tienen la misma duración.

Por lo tanto en el makespan el menos  $2t_{m+1}$



# Análisis del algoritmo mejorado (cont.)

(A.3) El algoritmo asigna los trabajos a las máquinas con un makespan  $T \leq 3/2 T^*$ .

Sea la maquina  $M_i$  que tiene al menos 2 trabajos.

Sea  $T_j$  el ultimo trabajo asignado a  $M_i$  ( $j \geq m + 1$ )

$$t_j \leq t_{m+1} \leq \frac{1}{2} T^* \quad (\text{Utilizando A.2})$$

$$T_i - t_j \leq T^*$$

$$(T_i - t_j) \leq \frac{1}{m} \sum_k T_k \leq T^*$$

$$(T_i - t_j) + T_j \leq \frac{1}{2} T^* + T^*$$

Por lo tanto:  $T_i \leq 3/2 T^*$







Presentación realizada en Julio de 2020

## Selección de centros

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# El problema de la selección de centros

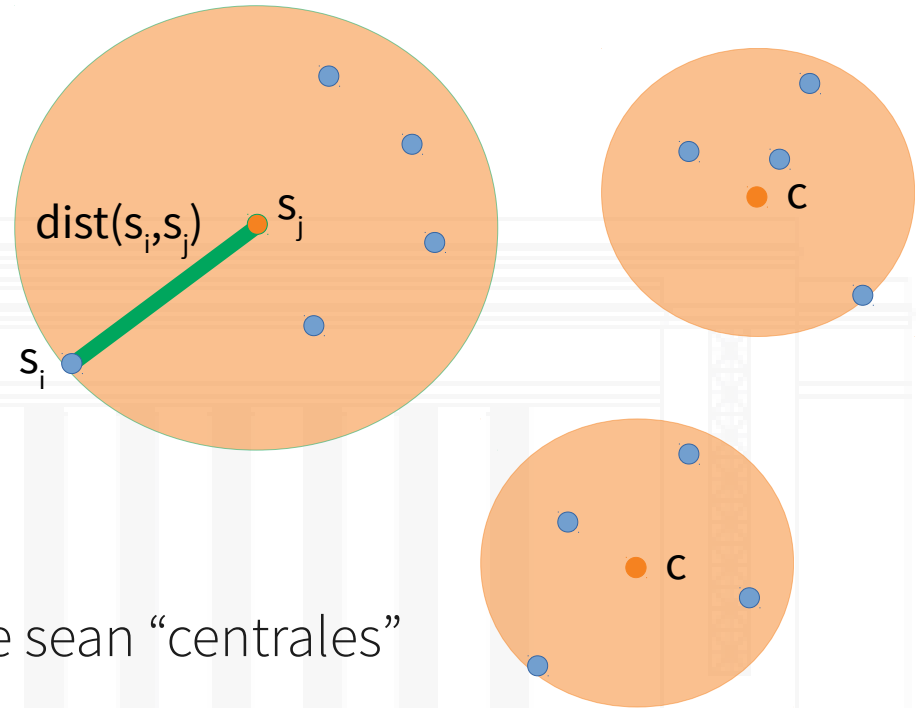
## Sea

Un set  $S$  de  $n$  sitios

Una función “distancia”

## Queremos

seleccionar  $k$  centros de tal forma que sean “centrales”



# Función distancia

## Función numérica

que mide el cuan cerca se encuentra un punto  $x$  de otro punto  $y$

### Debe satisfacer:

- $\text{dist}(x,x) = 0$
- $\text{dist}(x,y) = \text{dist}(y,x)$  (simetría)
- $\text{dist}(x,y) + \text{dist}(y,z) \geq \text{dist}(x,z)$  (desigualdad triangular)

### Ejemplo: Distancia euclidiana

# “Centralidad”

## Sea $C$ el set de centros.

Cada sitio “s” tiene una distancia a cada centro  $c$  de  $C$ .

Asignaremos a “s” a la órbita del centro  $c$  mas cercano.

$$\text{dist}(s, C) = \min_{c \in C} \text{dist}(s, c).$$

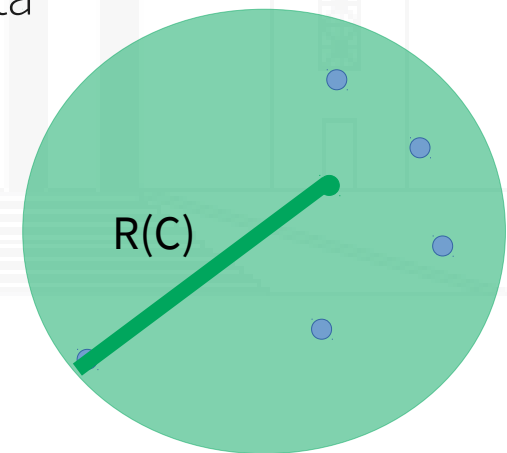
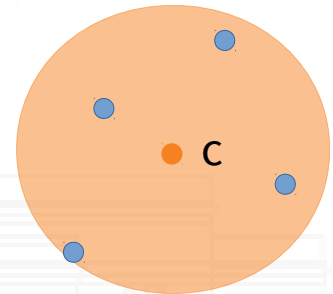
## Definimos a $r$ como el radio de cobertura.

Distancia máxima entre cada sitio y el centro al que órbita

$r(C)$  = radio de cobertura

## Intentaremos seleccionar el set $C$ de $k$ centros

para minimizar el radio de cobertura



# Tratando de construir un buen algoritmo

## Supongamos que

conocemos que existe un set  $C^*$  de  $k$  centros con radio de cobertura  $r(C^*) \leq r$

Cada sitio  $s$  debe tener un centro  $c^*$  en  $C^*$  que lo cubra

## Podemos construir

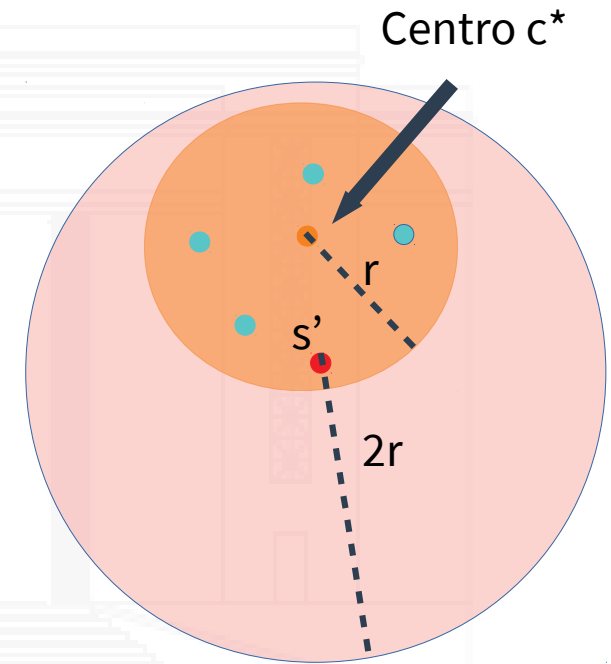
un set de  $k$  centros con radio de cobertura de como mucho  $2r$

## Seleccionamos un sitio $s'$

El sitio  $s'$  se encuentra a distancia máxima  $r$  de su centro  $c^*$

Definimos  $s'$  como centro con cobertura  $2r$

El nuevo centro  $s'$  contendrá a todos los sitios del centro  $c^*$



# Algoritmo propuesto

Sea  $S'$  la lista de sitios no asignados a un centro

$S' = S$

Sea  $C = \emptyset$

Mientras  $S \neq \emptyset$

    Seleccionar cualquier sitio  $s \in S$  y agregarlo a  $C$

    Borrar todos los sitios de  $S$  que tengan menor distancia a  $s$  de  $2r$

Si  $|C| \leq k$

    Retornar  $C$  como los centros seleccionados

Sino

    No hay  $k$  centros con radio de cobertura de al menos  $r$

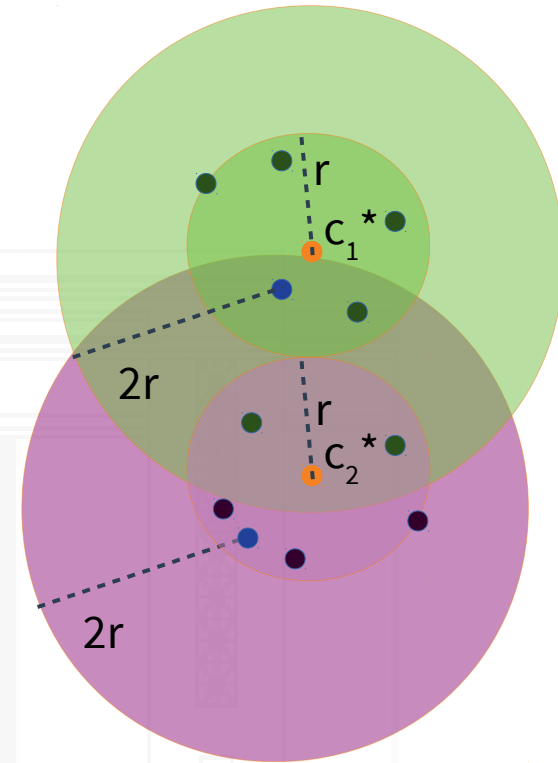
# Análisis del algoritmo

Si el algoritmo retorna un set de  $k$  centros

$$r(c) \leq 2r$$

Si selecciona más de  $k$  centros

$r(C^*) > r$  contradiciendo la afirmación





# ¿Qué valor tendrá $r$ en nuestro algoritmo?

**Podemos realizar un algoritmo iterativo probando diferentes valores de  $r$**

Iniciamos con un  $r = \text{distancia máxima entre 2 sitios} / 2$

**Ejecutamos el algoritmo y comprobamos si existe resultado**

Si existe puedo probar con un  $r$  más chico

Si no existe puedo probar con un  $r$  más grande

**Puedo iterar modificando los radios (similar a búsqueda binaria) y aproximar al resultado tanto como considere oportuno.**

El resultado será lo centros aproximados

# Un algoritmo greedy sin presuponer el radio

## Queremos resolver el problema

sin necesidad de suponer un radio de cobertura en la solución óptima

## Implica un cambio pequeño en el algoritmo anterior

Seleccionamos siempre como próximo centro al punto disponible más lejano a los centros existentes

## Si existe un centro a más de $2r$ de distancia de aquellos,

entonces el punto mas lejano debe formar parte de él

# Algoritmo propuesto

Asumimos  $k \leq |S|$  (sino definimos  $C = S$ )

Seleccionar cualquier sitio  $s$  y convertirlo en un centro  $C = \{s\}$

Mientras  $|C| < k$

    Seleccionar sitio  $s \in S$  que maximice la distancia  $\text{dist}(s, C)$

$C = C \cup \{s\}$

Returnar  $C$  como los sitios seleccionados

# Análisis del algoritmo

Si

$C^*$  es un set de centros optimos

$C$  es el set que encuentra el algoritmo

Entonces

$$r(C) \leq 2r(C^*)$$

Por contradiccion

asumiremos  $r(C^*) < \frac{1}{2} r(C)$

Por cada sitio  $c$  perteneciente a  $C$ , consideremos un circulo de radio  $\frac{1}{2} r(C)$  a su alrededor

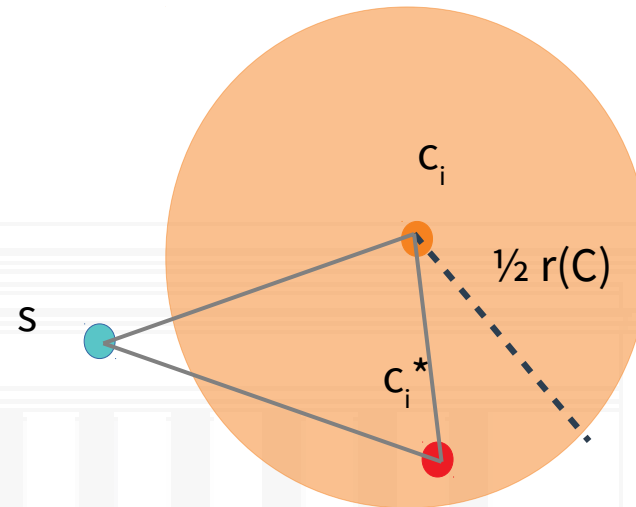
Llamaremos  $c_i$  a uno de ellos

Hay exactamente un único  $c^*$  dentro del circulo

Llamaremos  $c_i^*$  al que se encuentra dentro del circulo de  $c_i$

Consideremos cualquier sitio  $s$  y su centro mas cercano de la solución óptima  $C^*$

$$\begin{aligned} \text{dist}(s, C) &\leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*) \\ &\leq r(C^*) \text{ por que } c_i^* \text{ es su centro mas cercano} \end{aligned} \quad \longrightarrow \quad r(C) \leq 2r(C^*)$$



# Conclusión

## La solución presentada

Es un algoritmo de tipo greedy

## Corresponde a una 2-aproximación

Del problema de selección de centros

## Se ejecuta en tiempo polinomial



Presentación realizada en Julio de 2020

## Set cover

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Problema de set cover

## Sea

Set  $X$  de  $n$  elementos

Una lista  $F = \{S_1, \dots, S_m\}$  de subsets de  $X$

## Un set cover

es un subconjunto de  $F$  cuya unión es  $X$

## Objetivo

Encontrar el set cover  $S$  con menor cantidad de subsets



# Tratando de construir un buen algoritmo

## Construiremos un algoritmo greedy

Iremos seleccionando un subset de  $F$  paso a paso para el cover set

## Intentaremos

Cubrir la mayor cantidad de elementos aun no seleccionados

# Algoritmo propuesto

$R = X$  y  $S = \emptyset$  (sin sets seleccionados)

Mientras  $R \neq \emptyset$

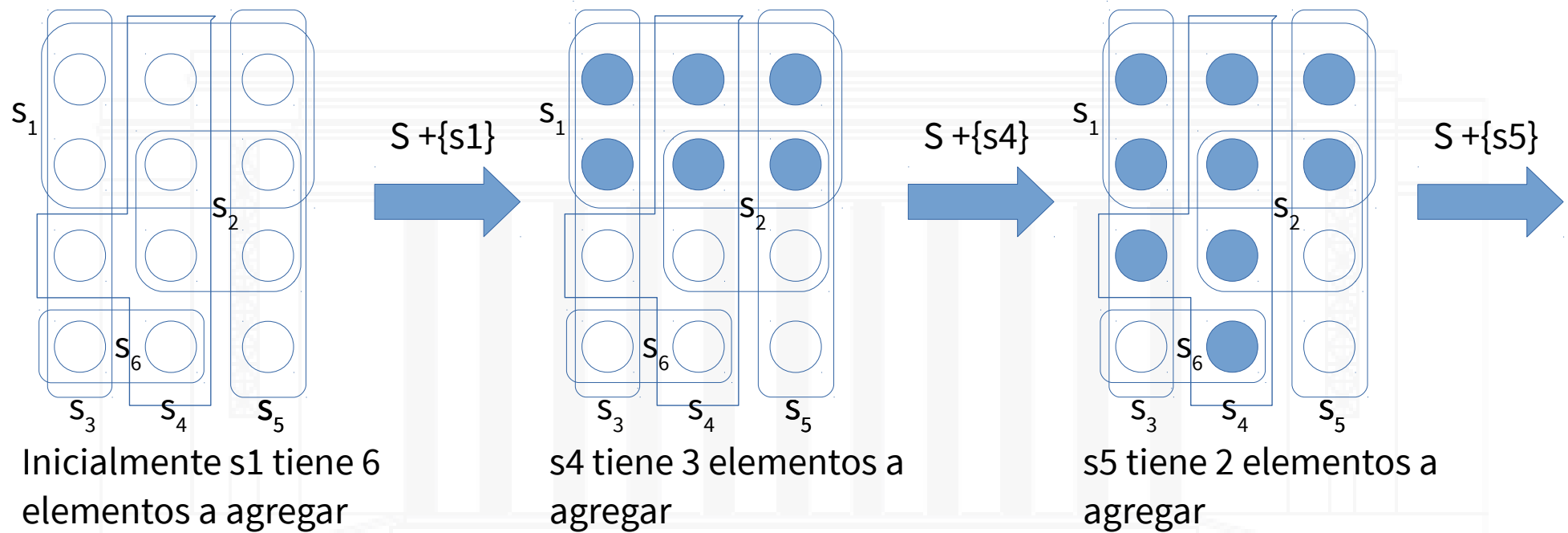
    Seleccionar set  $S_i$  con mayor  $S_i \cap R$

    Agregar set  $S_i$  a  $S$

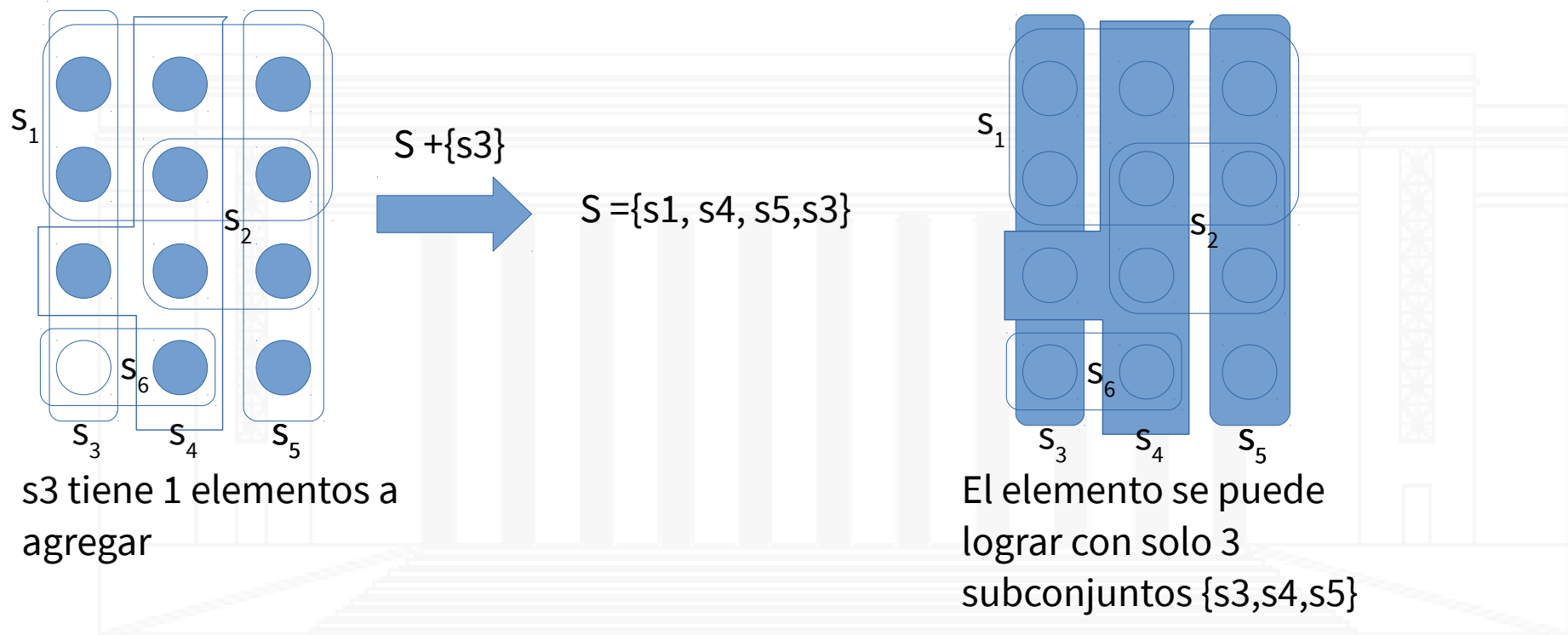
    Quitar elementos de  $S_i$  de  $R$

Retornar  $S$

# Ejemplo



# Ejemplo (cont.)



# Análisis del algoritmo

## El tiempo de ejecución del algoritmos

Esta acotado por la cantidad de subsets y elementos

Y se puede implementar en tiempo polinomial

**¿Qué tan diferente es el costo óptimo del  
Generado en el caso general?**

# Análisis del algoritmo

## Asignamos un costo de 1

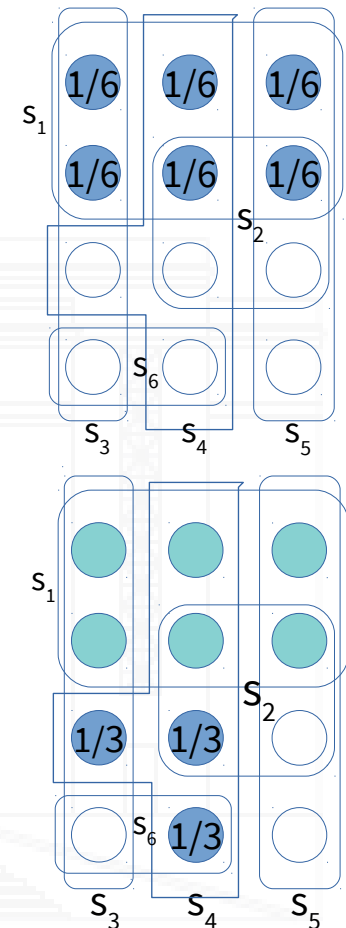
A cada set seleccionado por el algoritmo

## Distribuimos el costo

Entre los elementos cubiertos por primera vez

## Utilizaremos este costo para derivar las relación

Entre el tamaño del resultado optimo  $C^*$  y el tamaño del resultado retornado por el algoritmo greedy  $C$



# Análisis del algoritmo (cont.)

## Llamemos

Si al  $i$ -ésimo set seleccionado por el algoritmo greedy  
 $C_x$  el costo asignado al elemento  $x$ ,

## Sabemos que

A cada elemento  $x$  se le asigna el costo solo 1 vez

Si  $x$  es cubierto por  $S_i$

$$C_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}$$

# Análisis del algoritmo (cont.)

Podemos ver que

$$|C| = \sum_{x \in X} c_x$$

Ademas, cada elemento  $x$

se encuentra en al menos un set del resultado optimo. entonces

$$\sum_{s \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$$

En definitiva

$$|C| \leq \sum_{s \in C^*} \sum_{x \in S} c_x$$



# Análisis del algoritmo (cont.)

Sea  $S$  cualquier set de  $F$

Llamamos  $u_i = |S - (S_1 \cup S_2 \cup \dots \cup S_i)|$

a la cantidad de elementos aun no cubiertos luego del paso  $i$  en  $S$

Con

$$u_0 = |S|$$

Sea  $k$  el menor indice

Donde  $u_k = 0 \leftarrow$  no quedan elementos sin cubrir en  $S$

Y donde  $u_{k-1} > 0 \leftarrow$  aun quedaban elementos sin cubrir en  $S - (S_1 \cup S_2 \cup \dots \cup S_{k-1})$

Se puede ver que

$$u_{i-1} \geq u_i$$

$u_{i-1} - u_i$  es la cantidad de elementos que se cubren por  $S_i$

# Análisis del algoritmo (cont.)

Por lo que podemos expresar

$$\sum_{x \in S} c_x = \sum_{i=1}^k (u_{i-1} - u_i) \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}$$

Observar que

$$|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| = u_{i-1}$$

Por lo tanto

$$\sum_{x \in S} c_x \leq \sum_{i=1}^k (u_{i-1} - u_i) \frac{1}{u_{i-1}}$$

Que podemos reescribir como

$$\sum_{i=1}^k (u_{i-1} - u_i) \frac{1}{u_{i-1}} = \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}$$

Por la elección greedy,  
sino en ese paso  
debería seleccionar a  
S en vez de Si

# Análisis del algoritmo (cont.)

Entonces

$$\sum_{x \in S} c_x \leq \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \stackrel{j \leq u_{i-1}}{\leq} \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} = \sum_{i=1}^k \left( \sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)$$

Y

$$\sum_{i=1}^k \left( \sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) = \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) \stackrel{\text{Se cancelan las sumas intermedias}}{=} H(u_0) - H(u_k) = H(u_0) - H(0) \stackrel{H(0)=0}{=} H(u_0)$$

Por lo tanto

$$\sum_{x \in S} c_x \leq H(u_0) = H(|S|)$$

# Análisis del algoritmo (cont.)

Como

$$|C| \leq \sum_{s \in C^*} \sum_{x \in S} c_x \quad y \quad \sum_{x \in S} c_x \leq H(|S|)$$

Entonces

$$|C| \leq \sum_{s \in C^*} H(|S|) \leq |C^*| H(\max \{|S| : S \in F\})$$

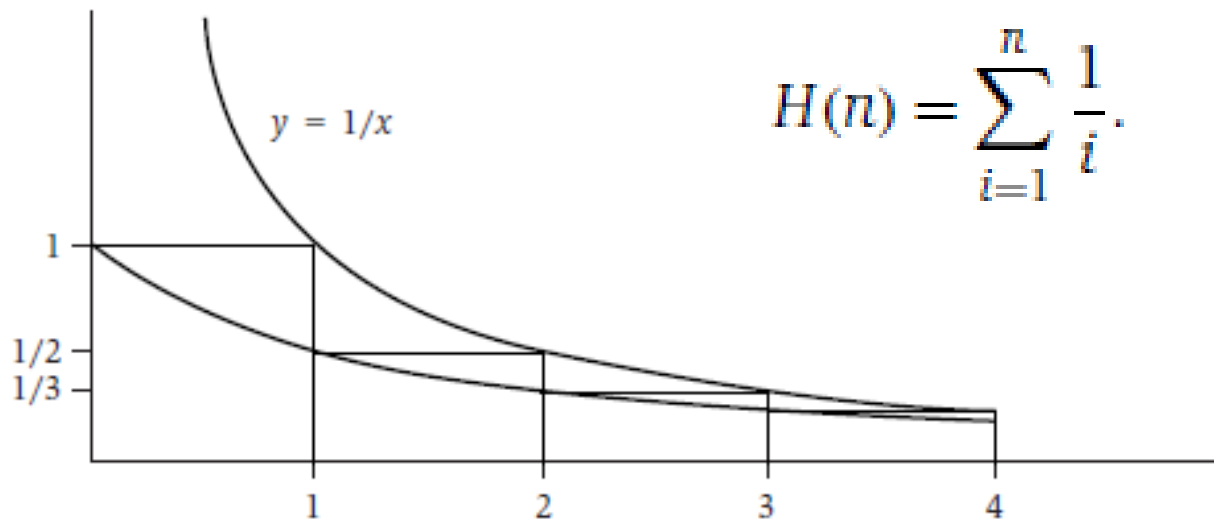
Podemos expresar  $|S|$

Por la cantidad  $n$  de elementos del set original por lo tanto  $|C| \leq |C^*| * \log(n)$   
(como mucho un set tiene los  $n$  elementos del conjunto)

# Función armónica

## Dada la función armónica

Vemos que la misma puede ser acotada por 2 funciones logarítmicas



$$H(n) = \sum_{i=1}^n \frac{1}{i}$$



$$H(n) = \Theta(\ln n).$$

# Conclusión

## Podemos expresar $|S|$

Por la cantidad  $n$  de elementos del set original por lo tanto  $|C| \leq |C^*| * \log(n)$

## Por todo lo anterior

Nuestro algoritmo es un  $(1+\log n)$ -algoritmo de aproximación



Presentación realizada en Julio de 2020

## Vertex cover

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)



# Problema Vertex cover

## Sea

Grafo  $G=(V,E)$

Cada vértice  $i$  tiene un peso  $w_i \geq 0$

## Queremos

encontrar el Set  $S \subseteq V$  donde cada arista  $E$  del grafo pertenezca a algún vértice de  $S$ .

Minimizando el costo de los vértices seleccionados.

# Costo pagado

## Existen

Diferentes subset  $S$  de  $V$  que conforman un vertex cover

## Llamaremos

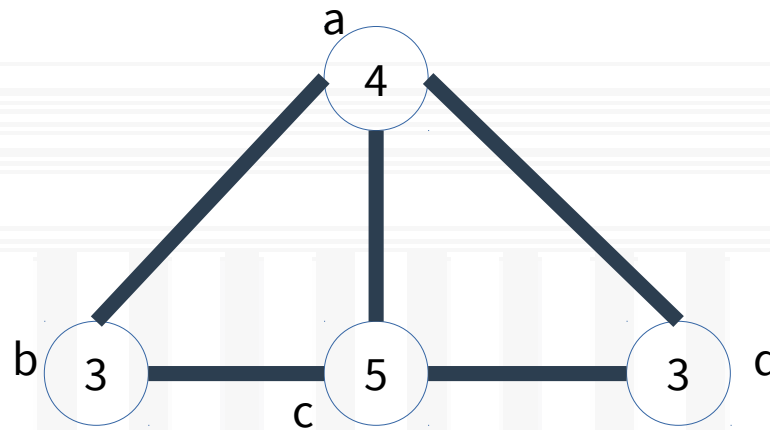
$w(S)$  como el costo del vertex cover formado por  $S \subseteq V$

## La solución optima $S^*$

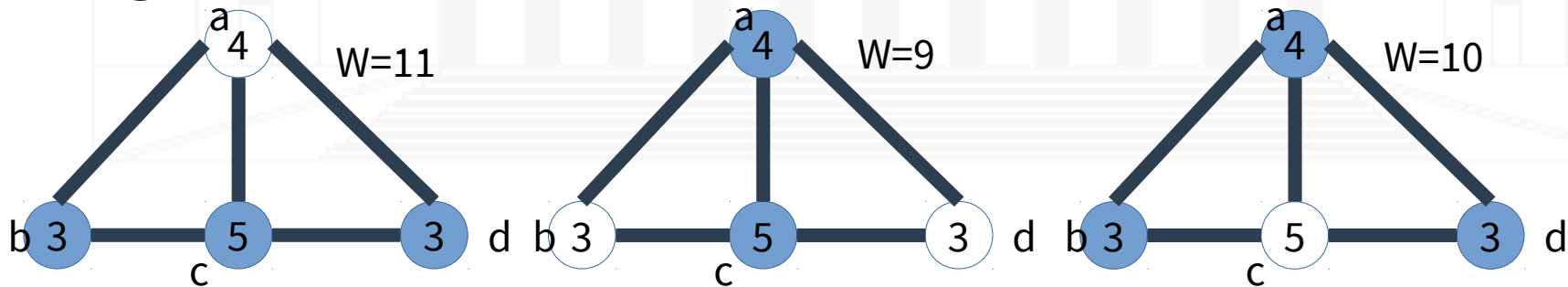
Es aquella para la que  $w(S^*) \leq w(S)$  para todo  $S$

# Ejemplo

Sea el siguiente gráfico



Las siguientes son coberturas de vértices



# Pricing method (A.K.A “primal-dual method”)

## Basados en una perspectiva económica

Podemos pensar cada peso de los vértices como un “costo”

## A cada vértice se le debe pagar por pertenecer a la solución

Cada eje es un “agente” dispuesto a pagar algo al vértice que lo cubre.

## Diremos que un vértices esta pagado

si la suma de lo pagado por sus ejes es igual al costo del vértice.

## Diremos que un precio a pagar $P_e$ por el vértice $e=(i,j)$ es “justo”

si  $P_e$  mas la suma de los otros pagos de los ejes incidentes a  $i$  no superan  $w_i$   
(idem para  $j$ )

# Algoritmo propuesto

Definir  $p_e = 0$  para todo  $e \in E$

Mientras exista un eje  $e=(i,j)$  tal que  $i$  o  $j$  no este “pagado”

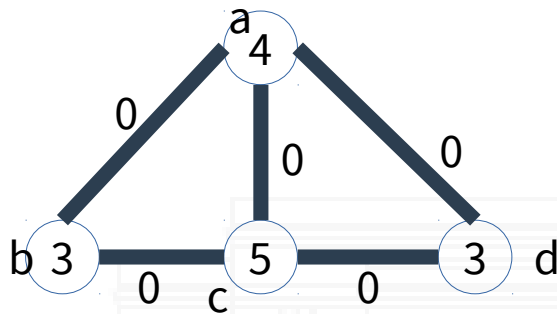
    Seleccionar el eje  $e$

    Incrementar  $p_e$  sin violar la integridad

Sea  $S$  el set de todos los nodos pagados

Retornar  $S$

# Ejemplo

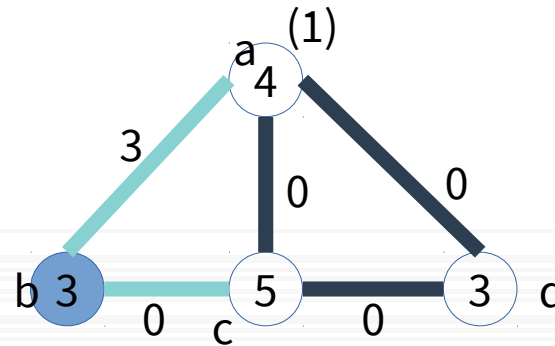


Inicialmente:  
 $p(e)=0$  para todo  $e \in E$   
 Nodos pagados:  $\emptyset$

Selecciono  
 Eje  $e=(a,b)$

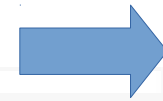


Puedo  
 pagar  
 hasta 3  
 $\min(4, 3)$

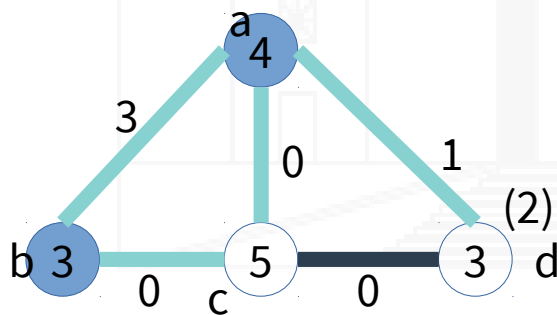


Nodos pagados: b

Selecciono  
 Eje  $e=(a,d)$



Puedo  
 pagar  
 hasta 1  
 $\min(1, 4)$

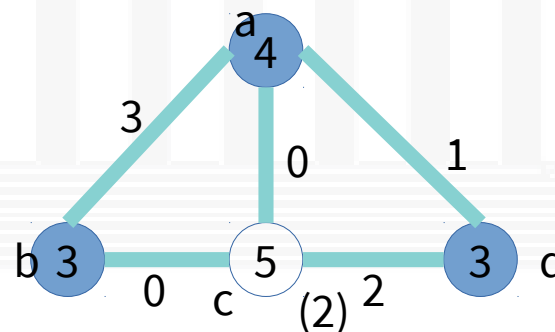


Nodos pagados: b,a

Selecciono  
 Eje  $e=(c,d)$



Puedo pagar  
 hasta 2  
 $\min(5, 2)$



Nodos pagados: b,a,d

Finalizacion:  
 No quedan ejes sin  
 cubrir  
 Vertex cover: a,b,d  
 $W(s): 10$

# Análisis del algoritmo

Para

cualquier vertex cover  $S^*$ ,  
cualquier precio justo  $p_e$ , tenemos que

**Podemos calcular**

$$w(S) = \sum_{i \in S} w_i$$

**Por definición de integridad**

tenemos que para todos los nodos  $i \in S^*$

$$\sum_{e=(i,j)} p_e \leq w_i$$

(!)El algoritmo –  
dependiendo la  
manera de elegir,  
puede construir  
cualquier vertex  
cover del Grafo (!)

# Análisis del algoritmo (cont.)

Sumando las desigualdades:

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

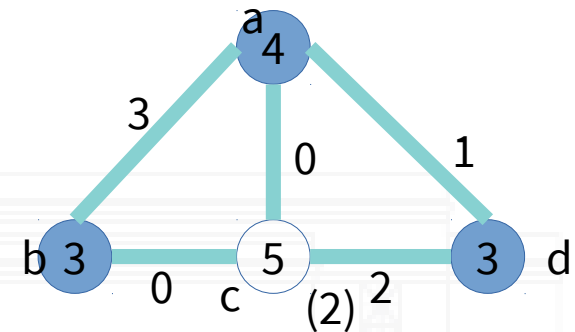
Por otro lado sabemos que

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e.$$

Finalmente combinamos y obtenemos:

$$\sum_{e \in E} p_e \leq w(S^*),$$

La suma de lo pagado por los ejes es menor a igual al costo de la cobertura de vértices



$S^*: b, a, d$   
 $w(S^*)=10$   
pagado=6



# Análisis del algoritmo (cont.)

Sea  $S$  el set retornado por el algoritmo

Todos los nodos en  $S$  están “pagados”

por lo que para todo  $i$  en  $S$ :  $\sum_{e=(i,j)} p_e = w_i$

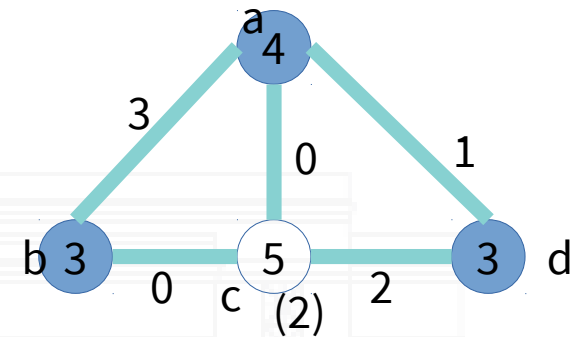
Podemos por lo tanto expresar el costo de  $S$

como 
$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e.$$

Un eje  $e=(i,j)$  puede sumar peso a dos vértices (aun sin estar en  $S$ )

por lo que: 
$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e,$$

El costo de la cobertura esta acotado por 2 veces la suma de los precios pagados



$S: b, a, d$   
 $w(S)=10$   
 $2 * \sum(p_e)=12$

# Análisis del algoritmo (cont.)

## Por funcionamiento del algoritmo

El set  $S$  obtenido es un vertex cover (sino no termina el “mientras”).

## Por los definiciones que probamos anteriormente:

$$\sum_{e \in E} p_e \leq w(S^*),$$

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e,$$

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*).$$

El costo del set  $S$  retornado por el algoritmo es como mucho el doble de algún vertex cover posible.

El algoritmo es un 2-algoritmo de aproximación



Presentación realizada en Julio de 2020

# Knapsack Problem aproximado

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Problema de la mochila

## Sea

Conjunto de  $n$  items  $X = \{x_1, \dots, x_n\}$

Cada item  $x_i$  contiene un valor  $v_i$  y un peso  $w_i$

Mochila de tamaño  $W$

## Deseamos

Encontrar un subset  $S \subseteq X$

## tal que

la suma de los pesos de los elementos en  $S$  no supere  $W$

Y la suma del valor de los elementos en  $S$  sea el máximo

# Soluciones

## Mediante programación dinámica

Hemos logrado un algoritmo pseudopolinomial  $O(nW)$

## Se ha demostrado

Que corresponde a un problema NP-Completo

## Por lo que (A menos que $P=NP$ )

No podemos encontrar un algoritmo de resolución totalmente polinomial

## Si $W$ y $N$ es muy grande

El problema se torna intractable

# Una solución aproximada...

## Propondremos

Un algoritmo de aproximación

## De tipo

Esquema de aproximación en tiempo polinómico

## Utilizaremos un parámetro

$\varepsilon$  que nos permitirá determinar la precisión deseada

## Se ejecutara

En tiempo polinómico

## Como parte de su ejecución

Utilizará programación dinámica

# Una solución parametrizada y acotable

## La solución de aproximación

Nos retornará un subconjunto de elementos  $S$

## Que no supere

Entre ellos el peso  $W$

## Con un valor total $V$

Que es igual o menor al valor máximo óptimo

## Fijaremos el parámetro $\epsilon$

para acotar la diferencia máxima entre el valor encontrado y el óptimo



# Programación dinámica (recargada)

## Necesitamos que el algoritmo

De programación dinámica utilice para hallar el óptimo el Valor (y no el peso)

## De esa forma podremos ajustar el parámetro $V$

Según nuestra conveniencia para aproximar el resultado

## El algoritmo dividirá el problema

En subproblemas que se superponen para memorizar y evitar repetir cálculo

# Subproblemas

## Llamaremos

$OPT(i,V)$


## Al subproblema de determinar

El menor peso que se puede obtener con los primeros  $i$  items cuyo valor iguale o supere el valor de al menos  $V$  en la mochila

## Se calculará el subproblema para

$i=0,\dots,n$

$V=0,\dots,V_{\max}$

Con  $V_{\max} = \sum_{j=1}^n v_j$   Valor equivalente a incluir todos los elementos en la mochila

# Casos base

## Para obtener un valor $v=0$

No hace falta poner ningún elemento

$$\text{OPT}(i,v)=0, v \leq 0$$

## Si tengo cero elementos

No puedo lograr ningún valor

(excepto si el valor es 0: Corresponde al caso anterior)

Para expresar la imposibilidad utilizaremos el  $\infty$

(o un peso mayor la suma de los pesos de todos los elementos )

$$\text{OPT}(v,i) = \infty, v > 0 \ i=0$$

# Solapamiento de subproblemas

## En un subproblema genérico $\text{OPT}(i,v)$

Pueden ocurrir 2 casos

### Que el $i$ -esimo problema no se encuentre en la solución

En ese caso buscamos el menor peso en lograr el valor  $v$  con los  $i-1$  elementos anteriores  $\rightarrow \text{OPT}(i-1,v)$

### Que el $i$ -esimo problema se encuentre en la solución

En ese caso sumamos a la mochila  $W_i$  de pesos y el menor peso para valor  $v-v_i$  con los  $i-1$  elementos  $\rightarrow \text{OPT}(i-1,v-v_i)$

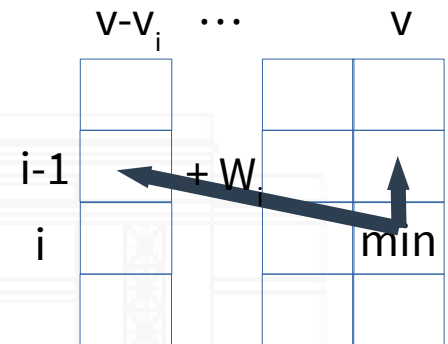
## Como se desea minimizar el peso de la mochila

el optimo contendrá el menor de los 2 casos

# Recurrencia

Podemos expresar la relación de recurrencia como

$$\begin{cases} 0 & , \text{ si } v=0 \\ \infty & , \text{ si } i=0 \text{ y } v>0 \\ \min \{ OPT(i-1, v), w_i + OPT(i-1, \max \{ 0, v - v_i \}) \} \end{cases}$$



Una vez que

tengo resueltos todos los subproblemas

**El valor que maximiza el problema sera**

El mayor  $u$  con  $u=0, \dots, V_{\max}$

que cumpla que  $OPT(n, u) \leq W$

# Pseudocódigo

## Complejidad

Temporal:  $O(nV_{\max})$

Espacial:  $O(nV_{\max})$

## Para recuperar los elementos seleccionados

Debo almacenar para cada caso si se selecciono o no que el elemento esté en el optimo

Iterar desde el optimo para atrás reconstruyendo

```
Desde i=0 a n
    OPT[i][0] = 0

Desde v=1 a Vmax
    OPT[0][v] =  $+\infty$ 

Desde i=1 a n // elementos
    Desde v=1 a Vmax // valores

        enOptimo = w[i] + OPT[i-1,v-v[i]]
        noEnOptimo = OPT[i-1,v]

        si enOptimo < noEnOptimo
            OPT[l][p] = enOptimo
        sino
            OPT[l][p] = noEnOptimo

Desde v=Vmax a 0
    si OPT[n,v] <= W
        retornar OPT[n,v]
```

# Acotando de forma mas conveniente

Si llamamos

$$v^* = \max\{V_i\} \text{ con } 0 < i \leq n$$

Podemos acotar

$$V_{max} = \sum_{j=1}^n v_j \leq n v^*$$

Por lo tanto

La complejidad de la programación dinámica sera  $O(n^2 v^*)$

(esta forma de expresarlo será ventajosa más adelante)

# Lo que tenemos hasta ahora...

## La solución es pseudo polinomial

depende de los valores  $v_i$

### Si $v^*$ es pequeño

Entonces  $O(n^2 v^*)$  “funcionará” en tiempo polinomial

### Sino

Tendremos que aproximar.



# Parámetro de redondeo

## Utilizaremos

El parámetro  $b$  de redondeo

## Para cada item $i$

Calcularemos  $\underline{v}_i = \lceil v_i / b \rceil * b$

## Todos los valores de items resultantes

Son múltiplos de  $b \rightarrow v_i \leq \underline{v}_i \leq v_i + b$

## Podemos resolver mediante programación dinámica utilizando

$\bar{v}_i = \lceil v_i / b \rceil$  ← nos asegura que sean valores enteros

No quedará  $v^*$  mas pequeño

Ejemplo:

$b=20$	$x_1$	$x_2$	$x_3$
$v_i$	126	37	413
$\underline{v}_i$	140	40	420
$\bar{v}_i$	7	2	21

↑  
 $v^*$

# Resolución del parámetro

## Resolveremos el problema

Utilizando los nuevos valores  $\bar{v}_i$

## El resultado obtenido

tiene el mismo set de elementos óptimos que utilizando  $v_i$   
(mismo peso y un difieren en un factor de  $b$ )

Obtenemos los elementos de la solución aproximada

Su valor real será menor o igual al obtenido

# Elección del parámetro $b$

## Utilizaremos

$\varepsilon$  para generar el parámetro  $b$ ,

Con  $0 < \varepsilon \leq 1$

Y por comodidad  $\varepsilon^{-1}$  un número entero

## Un valor conveniente de $b$

$$b = \varepsilon v^* / 2n$$

(esta elección nos servirá para las próximas demostraciones)

# Pseudocódigo

Obtener  $v_{\max}$

Definir  $b = \varepsilon v_{\max} / 2n$

Para cada elemento  $i$   
    Calcular  $\bar{v}_i$  con  $b$

Resolver con programación dinámica con valores  $\bar{v}_i$

Retornar el set de elementos encontrados

# Complejidad temporal global

**La programación dinámica se ejecuta en  $O(n^2 v^*)$**

Con  $v^* = \max\{V_i\}$  con  $0 < i \leq n$

**Si el item  $j$  es el de máximo valor**

Entonces  $v^* = \bar{v}_j = \lceil v_j / b \rceil$

**Siendo que  $b = \varepsilon v^* / 2n$**

Entonces  $v^* = 2n\varepsilon^{-1}$

**Todo el proceso**

será  $O(n^3 \varepsilon^{-1})$

Para un valor fijo de  $\varepsilon$  el algoritmo se ejecuta en tiempo polinomial (!)

# Margen de la aproximación

## Llamaremos

$S^*$  cualquier una solución que satisfice  $\sum_{i \in S^*} w_i \leq W$

## El algoritmo aproximado encuentra la solución optima $S$

Para los valores  $\underline{v}_i$  aproximados (fueron redondeados para arriba)

$$OPT(S) = \sum_{i \in S} v_i \geq \sum_{i \in S^*} \underline{v}_i$$

Se puede ver que

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \underline{v}_i \leq \sum_{i \in S} \underline{v}_i \leq \sum_{i \in S} b + v_i \leq nb + \sum_{i \in S} v_i$$

Por redondeo

Por ser optima por la aproximación

$$v_i \leq \underline{v}_i \leq v_i + b$$

Si Todos los elementos están en la solución

Si este fuese el máximo valor posible

$$\sum_{i \in S^*} v_i \leq nb + \sum_{i \in S} v_i$$

**(!) La solución encontrada es como mucho  $nb$  menor al máximo valor posible**

# Expresando en función de $\varepsilon$

Como  $b = \varepsilon v^* / 2n$

$$nb + \sum_{i \in S} v_i = \frac{\varepsilon}{2} v^* + \sum_{i \in S} v_i$$

Entonces

$$\sum_{i \in S^*} v_i \leq \frac{\varepsilon}{2} v^* + \sum_{i \in S} v_i$$

Como cualquier ítem entra en la mochila

Una posible solución  $S^* = \{x_j\}$  con  $v_j = v^*$

$$v^* \leq \frac{\varepsilon}{2} v^* + \sum_{i \in S} v_i \leq \frac{v^*}{2} + \sum_{i \in S} v_i \quad \Rightarrow \quad \frac{v^*}{2} \leq \sum_{i \in S} v_i \quad \Rightarrow \quad v^* \leq 2 \sum_{i \in S} v_i$$

Unificando

$$\sum_{i \in S^*} v_i \leq \frac{\varepsilon}{2} v^* + \sum_{i \in S} v_i \leq \frac{\varepsilon}{2} (2 \sum_{i \in S} v_i) + \sum_{i \in S} v_i \quad \Rightarrow \quad \sum_{i \in S^*} v_i \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

# Conclusión

Si

$S$  es la solución encontrada por el algoritmo de aproximación

$S^*$  es cualquier otra solución factible

Entonces

$$\sum_{i \in S^*} v_i \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

Por lo tanto,

Para cualquier  $\varepsilon > 0$ , la solución aproximada encuentra una solución factible cuyo valor está dentro de un factor  $(1 + \varepsilon)$  de la solución óptima

Y lo realiza en tiempo polinomial  $O(n^3 \varepsilon^{-1})$





Presentación realizada en Julio de 2020

# Problema del viajante de comercio aproximado

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Problema del viajante de comercio (TSP)

## Sea

Un conjunto de  $n$  ciudades “C”

Un conjunto de rutas con costo de tránsito

Existe una ruta que une cada par de ciudades

El costo de cada ruta puede ser simétrico o asimétrico (diferente valor ida y vuelta)

## Queremos

Obtener el circuito de menor costo

que inicia y finalice en una ciudad inicial

y pase por el resto de las ciudades 1 y solo 1 vez

# Soluciones

## Por fuerza bruta

Se resuelve en  $O(n!)$

## Mediante programación dinámica

Hemos logrado un algoritmo exponencial  $O(n^2 2^n)$

## Se ha demostrado

Que corresponde a un problema NP-Completo

## Por lo que (A menos que $P=NP$ )

No podemos encontrar un algoritmo de resolución totalmente polinomial

## Podremos aproximarlos de alguna manera?

Simplificaremos nuestro problema para lograrlo

# Problema “simplificado” del viajante de comercio

## Sea

Un conjunto de  $n$  ciudades “ $C$ ”

Un conjunto de rutas con costo no negativo de tránsito

Existe una ruta que une cada par de ciudades

El costo de cada ruta es simétrico y cumple con la desigualdad triangular

## Queremos

Obtener el circuito de menor costo

que inicia y finalice en una ciudad inicial

y pase por el resto de las ciudades 1 y solo 1 vez

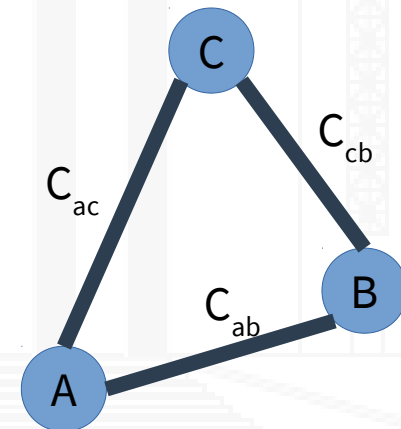
# Desigualdad triangular

## Viajar de la ciudad A a la ciudad B

De forma directa tiene un costo siempre menor o igual

Al de hacer el mismo viaje utilizando ciudades intermedias

$$C_{ab} \leq C_{ac} + C_{cb}$$



# Nomenclatura

## Podemos

Representar el problemas mediante un Grafo  $G=(V,E)$  con  $V$ : conjunto de ciudades y  $E$ : conjunto de rutas

Cada ruta  $e \in E$ , une dos ciudades  $u,v \in V$  tendrá un asociado el costo  $c(u,v) \geq 0$

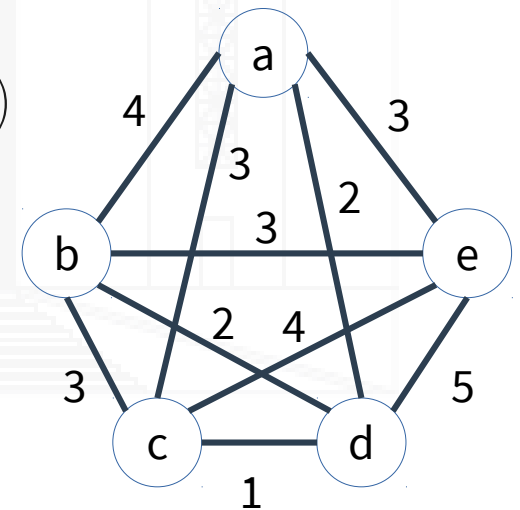
## La solución

Estará computa por una secuencia de ciudades  $(c_1, c_2, \dots, c_n, c_1)$

Constará de un subset  $A \in E$  de rutas a utilizar

## El costo total incurrido será

$$c(A) = \sum_{(u,v) \in A} c(u,v)$$

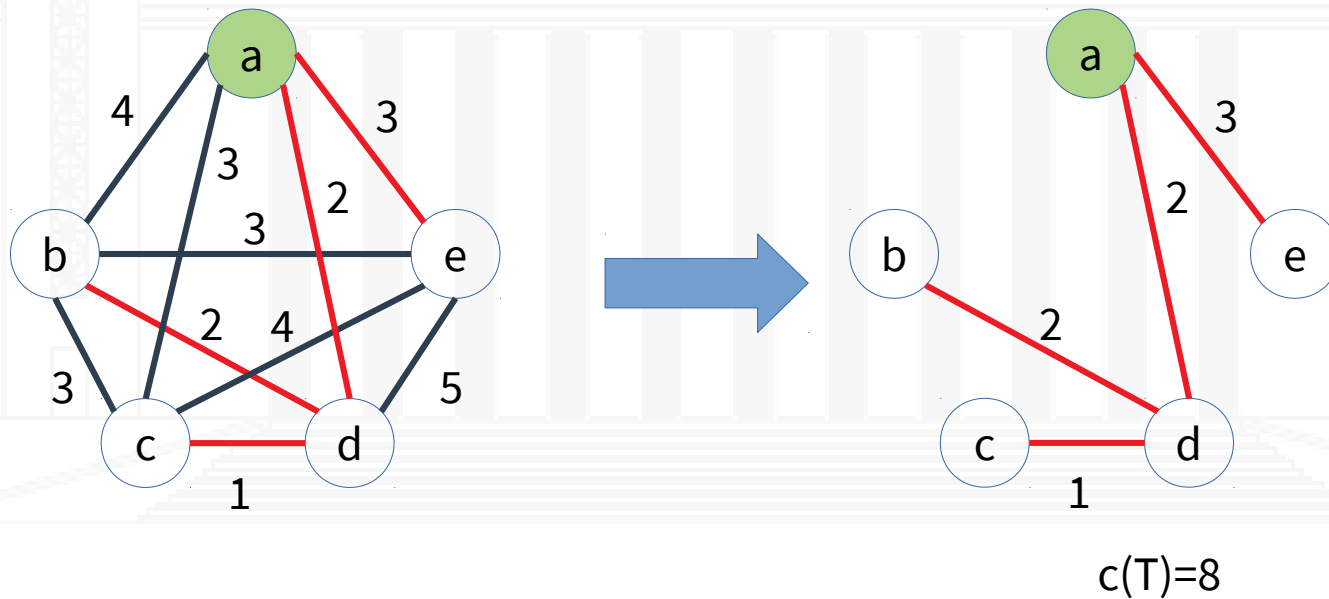


# Iniciando con un árbol recubridor

## Seleccionamos un vértice $r \in V$

Calculamos  $T$  el árbol recubridor mínimo de  $G$  usando  $r$  como raíz

Llamamos  $c(T)$  a la sumatoria de los ejes del árbol  $T$





# Full Walk

## Podemos construir $W$ el full walk de $T$

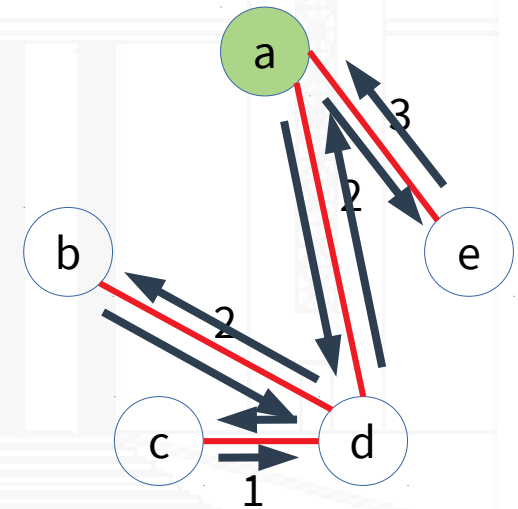
Iniciando en la raíz y recorriendo todo el árbol hasta volver al inicio

Con este recorrido visitamos todas las ciudades al menos una vez

(pero solo tendría que ser una vez)

## Llamamos $c(W)$ al costo de este recorrido

$$c(W) = 2 c(T)$$



$W: a, d, b, d, c, d, a, e, a$

$$c(W) = 2 C_{ad} + 2 C_{db} + 2 C_{cd} + \dots$$

# Aplicando desigualdad triangular

**Debemos asegurarnos que todas las ciudades (excepto la inicial)**

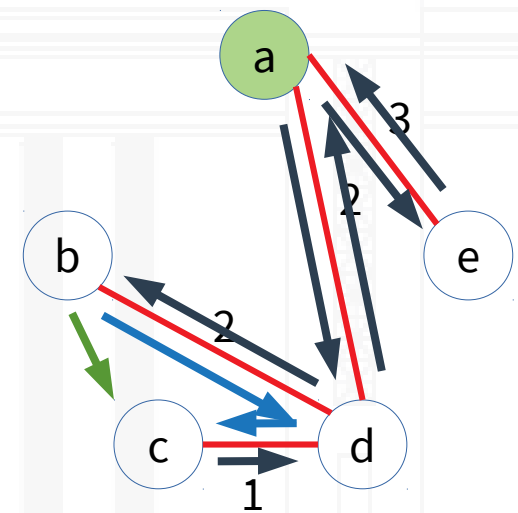
Solo sean visitadas 1 vez

## Conceptualmente

Implica no volver para atrás a una ciudad visitada e ir a la siguiente sin visitar.

**Lo podemos hacer gradualmente ciudad por ciudad**

La desigualdad triangular nos asegura que el costo final obtenido sera menor o igual al de  $c(W)$



$$C_{bc} \leq C_{bd} + C_{dc}$$

# Circuito hamiltoniano resultante

## Al finalizar nos quedara un circuito hamiltoniano H

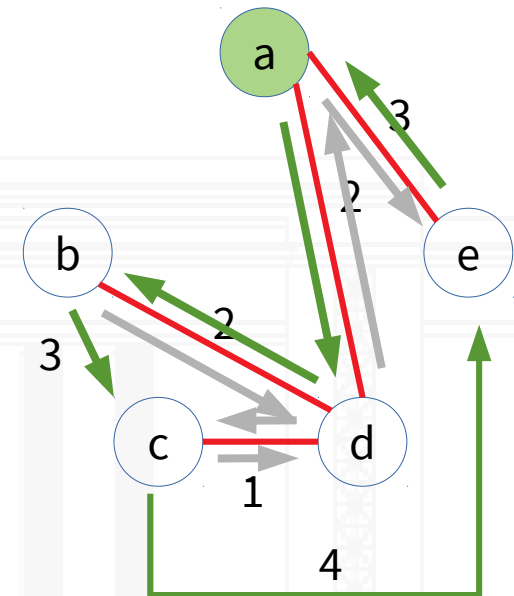
De menor costo al fullwalk y que cumple con los requerimientos del problema del viajante

## No es necesario construir el full walk

Pero el concepto del full walk es útil a la hora de la demostración de aproximación

## Podemos utilizar sobre el arbol T

Depth first search enumerando los nodos mediante preorden y obtenemos H



W: a,d,b,d,c,d,a,e,a

H: a,d,b,c,e,a

$$c(W) = 2 c(T) \geq c(H)$$

$$16 = 2 \cdot 8 \geq 14$$

# Aproximación del viajante

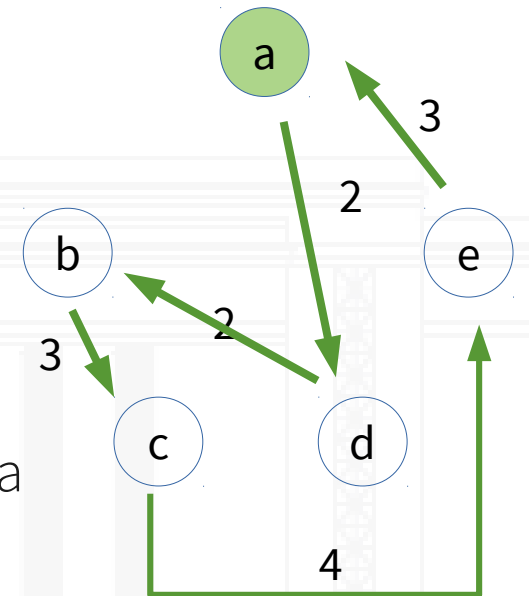
## El circuito del viajante aproximado

Corresponde al ciclo hamiltoniano H encontrado

## ¿Qué tan buena aproximación es?

Podría pasar que la solución encontrada sea la optima

Pero de no serlo, hay alguna desviación máxima que nos asegura el algoritmo?



# Análisis de la solución

Sea

$H^*$  el circuito optimo para el grafo  $G$

$T$  el árbol recubridor mínimo del grafo  $G$

**Eliminando solo un eje de  $H^*$**

Obtenemos un árbol (que podría ser  $T$ )

**Por lo tanto**

El costo  $H^*$  es mayor o igual al de  $T$

$$C(T) \leq C(H^*)$$

# Análisis de la solución (cont.)

## Como

El costo del full walk  $W$  es el doble del costo del árbol  $T$

Y

El costo del ciclo hamiltoniano es menor al costo del fullWalk

(por desigualdad triangular)

## Podemos concluir que

El algoritmo presentado es un 2-algoritmo de aproximación

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$

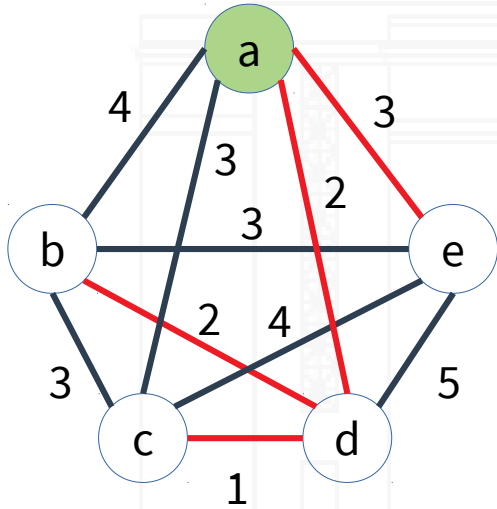
$$C(H) \leq C(W)$$



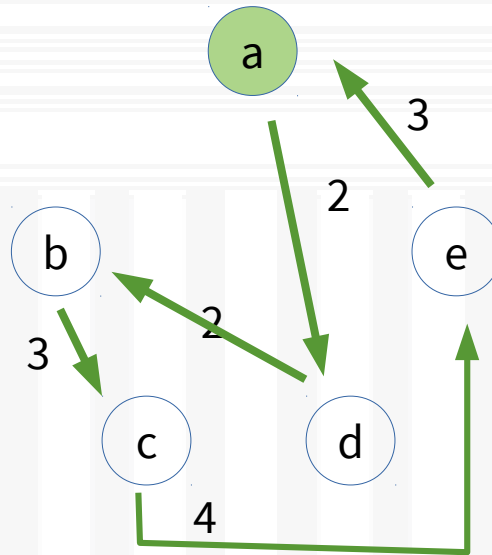
$$C(H) \leq 2C(H^*)$$

# En el ejemplo

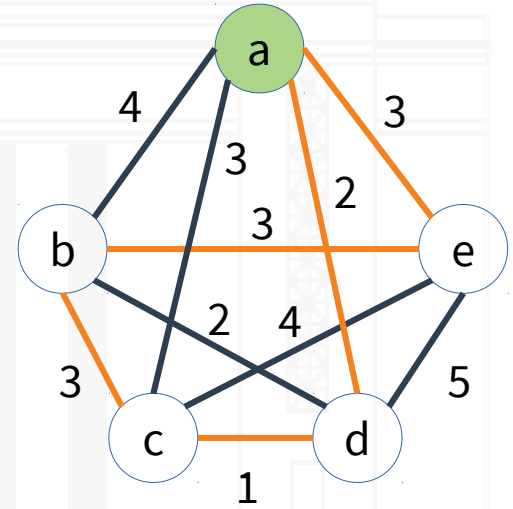
$$C(H) \leq 2C(H^*)$$



$$C(T)=8$$



$$C(H)=14$$



$$C(H^*)=12$$

# Complejidad Temporal

**El algoritmo se puede dividir en las siguientes partes:**

Cálculo del árbol recubridor mínimo de  $G$

Recorrido de  $T$  mediante DFS para generar lista  $H$  (utilizando preorden)

**Ambos algoritmos se pueden ejecutar en tiempo polinomial**

Árbol recubridor usando Kruskal  $\rightarrow O(E \log V)$

DFS en un árbol  $\rightarrow O(V)$

**Por lo que nuestro algoritmo se ejecuta en tiempo polinomial**





Presentación realizada en Enero de 2021

## Max 3-SAT

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ [vpodberezski@fi.uba.ar](mailto:vpodberezski@fi.uba.ar)

# Enunciado

## Dado

Un conjunto de  $n$  variables  $\{x_1, x_2, \dots, x_n\}$

Un conjunto de  $k$  cláusulas  $C = (x_i \vee x_j \vee x_k)$  conjugadas

## Con

Cada cláusula tiene 3 variables distintas y sin contener la misma variable y su negada

## Queremos saber

La cantidad máximas de cláusulas que se pueden satisfacer

# Planteo

## Es una variante de 3-SAT: MAX-3SAT

Intentamos maximizar y no determinar si se puede satisfacer toda la expresión

## Se ha demostrado

Que corresponde a un problema NP-HARD

## Propondremos

Utilizar un algoritmo de aproximación randomizado

# Propuesta

## Nuestro algoritmo

Es muy simple!

### Para cada variable $x_i$

Determinaremos su valor 0 o 1 en forma aleatoria e independiente

### La probabilidad de que una variable $x_i$ esté “activada”

$$\Pr(X_i=1) = 1/2$$

# Número de esperado de clausulas satisfechas

**Sea**

$Z$  la variable aleatoria igual al número de clausulas satisfechas

$Z_i$  la variable aleatoria con valor 0 o 1 de acuerdo a si la clausula  $i$  esta satisfecha

**Entonces**

$$Z = Z_1 + Z_2 + \dots + Z_k$$

**Queremos**

Determinar el numero de clausulas satisfechas esperados  $E[Z]$

# Nro de esperado de clausulas satisfechas (cont.)

**Como**

$$E[Z] = E[Z_1 + Z_2 + \dots + Z_k] = E[Z_1] + E[Z_2] + \dots + E[Z_k]$$

**Con**

$$E[Z_i] = \Pr[C_i=1]$$

**Como las variables son independientes**

$$\Pr[C_i=1] = 1 - \Pr[C_i=0] = 1 - 1/2^3 = 7/8$$

**Entonces**

$$E[Z] = k \cdot 7/8$$

Esperamos que una asignación aleatoria satisfaga a un  $7/8$  de las clausulas

# Nro de esperado de clausulas satisfechas (cont.)

## Como

No se pueden satisfacer mas de  $k$  clausulas

Y

Esperamos satisfacer  $\frac{7}{8}$  de ellas

## Entonces

El numero de esperado de clausulas satisfechas mediante una asignación aleatoria esta dentro de un factor de aproximación de  $\frac{7}{8}$  del optimo



# Una afirmación fuerte

## Para

cualquier expresión de 3SAT

## Donde

Cada clausula tiene 3 variables distintas y sin contener la misma variable y su negada

## Existe

Una asignación de verdad que satisface al menos  $\frac{7}{8}$  de las clausulas

# Esperando una buena asignación

## Este método

No garantiza  $\frac{7}{8}$  de cláusulas satisfechas!

(podría ser menos, podría ser más)

## Solo

Indica que es probable y esperable

## ¿Como puedo garantizar este resultado

En tiempo polinómico?

# Repetición del problema

## Podemos

Repetir la asignación de variables aleatorias

## Hasta

Conseguir el piso de  $\frac{7}{8} k$  cláusulas satisfechas

## Pero...

No sabemos cual sera el número esperado de repeticiones

# Una demostración previa...

**Si**

Repetimos la ejecución de pruebas independientes de un experimento

**Cada una de ellas**

Con probabilidad  $p > 0$

**Entonces**

El numero de pruebas esperado antes del primer éxito es  $1/p$

# Demostración

## Sea

Variable  $v$  tal que  $\Pr(v=\text{éxito})=p$  y  $\Pr(v=\text{fallo})=1-p$

$X$  la repetición de  $v$  un número de veces hasta el éxito

## Entonces

La probabilidad de lograr el éxito en  $j$  iteraciones

$$\Pr[X=j] = (1-p)^{j-1} * p$$

$$E[X] = \sum_{j=0}^{\infty} j * \Pr[X=j] = \sum_{j=0}^{\infty} j(1-p)^{j-1} * p = \frac{p}{1-p} \sum_{j=0}^{\infty} j(1-p)^j = \frac{p}{1-p} * \frac{1-p}{p^2}$$

$$E[X] = \frac{1}{p}$$

# Determinación de cantidad de repeticiones

## Si logramos demostrar

Que la probabilidad de asignación de  $\frac{7}{8}k$  de las clausulas es al menos  $p$

## Entonces

La cantidad de pruebas a realizar esperadas será  $1/p$

## Que valor tomará $p$ ?

Queremos mostrar que es inversamente polinomial en función de  $n$  y  $k$

$\rightarrow 1/f(n,k)$

# Determinación de cantidad de repeticiones (cont.)

## Llamaremos

$P_j$  a la probabilidad que una asignación aleatoria satisfaga exactamente  $j$  clausulas (con  $j=0,1,\dots,k$ )

## Queremos saber

$$p = \sum_{j \geq 7k/8} Pr[P_j]$$

Sabemos que el valor esperado de clausulas satisfechas

$$E[P] = \sum_{j=0}^k j * Pr[P_j] = \frac{7}{8}k \quad \leftarrow \text{Lo calculamos antes!}$$

$$\sum_{j < 7k/8} j * Pr[P_j] + \sum_{j \geq 7k/8} j * Pr[P_j] = \frac{7}{8}k$$

# Determinación de cantidad de repeticiones (cont.)

## Llamaremos $k'$

Al mayor de los números enteros estrictamente menor a  $\frac{7}{8}k$

## Entonces

$$\begin{aligned}\frac{7}{8}k &= \sum_{j < 7k/8} j * Pr[P_j] + \sum_{j \geq 7k/8} j * Pr[P_j] \leq \sum_{j < 7k/8} k' * Pr[P_j] + \sum_{j \geq 7k/8} k * Pr[P_j] \\ &\leq k' * \underbrace{\sum_{j < 7k/8} Pr[P_j]}_{1-p} + k * \underbrace{\sum_{j \geq 7k/8} Pr[P_j]}_p = k' * (1-p) + k * p \leq k' + k * p\end{aligned}$$

$$\frac{7}{8}k \leq k' + k * p$$



# Determinación de cantidad de repeticiones (cont.)

Continuando

$$\frac{7}{8}k \leq k' + kp \quad \longrightarrow \quad kp \geq \frac{7}{8}k - k'$$

Por como elegimos  $k'$

$$\frac{7}{8}k - k' \geq \frac{1}{8}k$$

Entonces

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8}$$

# Conclusión

## Como

la probabilidad de asignación de  $\frac{7}{8}k$  de las clausulas es al menos  $p=1/8k$

## Entonces (dada la propiedad antes vista)

El numero de pruebas esperado antes del primer éxito es  $8k$

## De esta forma conformando

Un  $\frac{7}{8}$ -algoritmo de aproximación randomizado



Presentación realizada en Enero de 2021