

Algoritmos randomizados: Presentación

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Algoritmos randomizados

Un algoritmo randomizado

Es aquel que resuelve un problema P

Utilizando

Como parámetro extra una cadena aleatoria “ r ”

Decisiones de ejecución

Se realizan teniendo en cuenta la lectura de la cadena aleatoria.

Son “elecciones aleatorias”

Optimalidad y complejidad temporal

Diferentes ejecuciones

de la misma instancia del problema

Puede ejecutarse

en diferente cantidad de pasos, o

Puede retornar

una salida diferente

Ventajas

Permiten

Construir soluciones “simples” de implementar (y entender)

Hallar soluciones

más rápido que los mejores algoritmos conocidos
(con la posibilidad de fallar en el intento o en el tiempo)

(Algunos) casos de uso

Verificación de identidad

Ej: determinar si el resultado de la multiplicación de 2 matrices es correcto

Ordenar o mezclar elementos

Ej: barajar cartas para un juego online

Quiebre de simetría

Ej: gestión de recurso único ante peticiones simultáneos

Balanceo de carga

Ej: asignación de tareas a varias unidades de procesamiento

Detección de propiedad mediante “testigo”

Ej: Determinación si un número es compuesto o primo

Origen de la aleatoriedad

Realizando un algoritmo

No se puede construir una función aleatoria

Existen procesos

De origen natural de tipo aleatorios
(radiación, fluctuación térmica, etc)

Computacionalmente

Lo más que se puede lograr son funciones pseudoaleatorias

Tipos de Algoritmos randomizados: Monte Carlo

Dan resultados

probablemente correctos

Se espera que

La probabilidad de obtener un valor correcto sea grande

Se ejecutan

En tiempo polinomial

Tipos de Algoritmos randomizados: Las Vegas

Dan resultados

correctos

Se ejecuta

probablemente rápidos.

Se espera que

su tiempo de ejecución sea rápido

No tiene una cota al tiempo de ejecución (No terminan hasta hallar el resultado correcto)

Clases de complejidad: RP

Se conoce

Como “RP” (o “R”)

A aquellos problemas de decisión

Para los que existe un programa “M” randomizado

Que se ejecuta en tiempo polinomial

Tal que para toda instancia I del problema

Si I es “si”, entonces $\text{pr}(M(I,r) = \text{“si”}) \geq \frac{1}{2}$

Si I es “no”, entonces $\text{pr}(M(I,r) = \text{“si”}) = 0 \quad \leftarrow$ No hay falsos positivos

	Respuesta Producida	
	SI	NO
Respuesta Correcta	SI	$\geq 1/2$
	NO	0

Si la respuesta producida es “si”, es la respuesta correcta.
Sino, no se

Clases de complejidad: co-RP

Se conoce

Como “co-RP” (o “co-R”)

A aquellos problemas de decisión

Para los que existe un programa “M” randomizado
Que se ejecuta en tiempo polinomial

Tal que para toda instancia I del problema

Si I es “si”, entonces $\text{pr}(M(I,r)=\text{“no”}) = 0 \quad \leftarrow$ No hay falsos negativos

Si I es “no”, entonces $\text{pr}(M(I,r)=\text{“no”}) \geq 1/2$

		Respuesta Producida	
		SI	NO
Respuesta Correcta	SI	1	0
	NO	$\leq 1/2$	$\geq 1/2$

Si la respuesta producida es “no”, es la respuesta correcta.
Sino, no se

Clases de complejidad: ZPP

Se conoce

Como zero-error probabilistic P (ZPP)

A aquellos problemas de decisión

Que pertenecen a RP y co-RP

Para toda instancia I del problema

Podemos ejecutar el algoritmo en RP y co-RP

En tiempo polinomial tendremos 3 respuestas posibles

Si, No y No Se sabe

La repetición de un numero no determinado de ejecuciones

Nos asegura obtener el resultado correcto

RP	co-RP	ZPP
NO	NO	NO
NO	SI	NO SE
SI	NO	(imposible)
SI	SI	SI

Corresponden a los algoritmos conocidos como Las Vegas

$RP \cap co-RP$

Clases de complejidad: BPP

Se conoce como

bounded-error probabilistic P (BPP)

A aquellos problemas de decisión

Para los que existe un programa “M” randomizado

Que se ejecuta en tiempo polinomial

Tal que para toda instancia I del problema

Si I es “si”, entonces $\text{pr}(M(I,r)=\text{“si”}) \geq 2/3$

Si I es “no”, entonces $\text{pr}(M(I,r)=\text{“si”}) \leq 1/3$

No podemos estar seguros

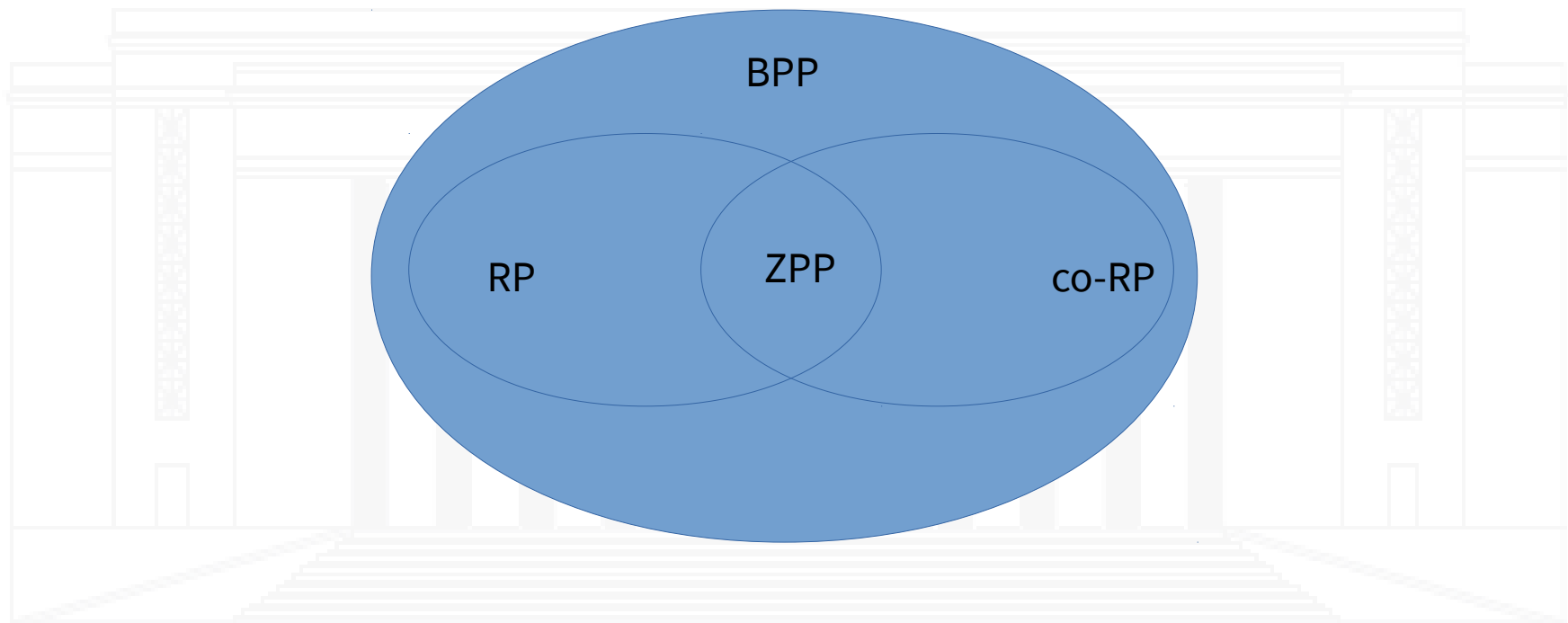
Si el resultado es correcto,

podemos afirmarlo con “alta probabilidad”

		Respuesta Producida	
		SI	NO
Respuesta Correcta	SI	$\geq 2/3$	$\leq 1/3$
	NO	$\leq 1/3$	$\geq 2/3$

Corresponden a los
algoritmos
conocidos como
Monte Carlo

Relación entre clases





Presentación realizada en Junio de 2020

Mezcla aleatoria

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Enunciado

Sea

Set A de n elementos

Queremos

Generar un listado de A ordenado aleatoriamente

Mezclar un conjunto de elementos se utiliza en

Juegos de azar

Reproducción de música aleatoria

Modelos estadísticos

Simulaciones

Pruebas de complejidad algorítmica

Método 1: Permutación por ordenamiento

Para cada i elemento en A

Generaremos un número p_i aleatorio como su clave

Utilizando p_i para cada elemento a_i como “clave”

Ordenaremos A

Podemos elegir

Cualquier algoritmo de ordenamiento como caja negra para resolverlo

Pseudocódigo

```
Sea A[1...n] conjunto a ordenar  
Sea P[1...n] vector numérico // vector de prioridades
```

```
Desde j=1 a n  
    P[j] = random_value(1...x)
```

```
Ordenar A utilizando P como clave
```

```
Retornar A
```

Problemas del método

La generación de la clave de ordenamiento (valor entre 1 y x)

De cada elemento en A

Puede perder uniformidad aleatoria

Si existen claves duplicadas

El ordenamiento

De claves repetidas depende el método de ordenamiento (puede ser estable o no)

Y es un proceso determinístico

Problemas del método (cont.)

Para disminuir la posibilidad de claves repetidas

Podemos establecer el valor $X \gg n$ (por ejemplo n^5)

Aun así la posibilidad persistirá

Se puede agregar un registro de claves utilizadas

y volver a seleccionar si surge una ya utilizada

Si evitamos claves repetidas

Podemos utilizar cualquier algoritmo de ordenamiento como caja negra

Análisis de complejidad

El proceso de generacion de claves (con $x \gg n$)

Es $O(n)$ en tiempo

En $O(n)$ en espacio

El proceso de ordenamiento

Es $O(n \log n)$

Es $O(n)$ en espacio (depende del método elegido, podría ser $O(1)$)

La complejidad total

Temporal: $O(n \log n)$

Espacial: $O(n)$

Análisis de uniformidad

Llamamos

E_i al evento que el elemento $A[i]$ obtiene la i -ésima menor clave

Suponemos

Que todo evento E_i ocurre (corresponde a una permutación posible)

Y son independientes entre ellos

Entonces

$$\Pr(E_1 \cap E_2 \cap \dots \cap E_n) = \Pr(E_1) * \overset{1/n}{\Pr(E_2 / E_1)} * \dots * \overset{1/(n-1)}{\Pr(E_n / E_{n-1} \cap E_{n-2} \cap \dots \cap E_1)}$$

$$\Pr(E_1 \cap E_2 \cap \dots \cap E_n) = 1 / n!$$

Si probamos cualquier otra permutación

Veremos que su probabilidad también es $1 / n!$

Por lo tanto este método genera una permutación aleatoria uniforme

Método 2: Algoritmo de Fisher-Yates

Algoritmo de mezcla

Propuesto por Ronald A. Fisher y Frank Yates

En libro “Statistical tables for biological, agricultural and medical research” (1948)

También conocido como

“barajado del sombrero”

Tiene gran cantidad de variantes

(analizaremos 1 variante)

Fisher Yates: Descripción “popular”

Se introducen todos los números

en un sombrero,
se agita el contenido(se mezclan)

Se van sacando de a uno

Y se listan en el mismo orden en que se sacan
hasta que no quede ninguno.



El resultado es el conjunto mezclado.

Descripción algorítmica

Para cada elemento $A[i]$

Generamos un valor x al azar entre i y n

Intercambiamos $A[i]$ con $A[x]$

Sea $A[1..n]$ conjunto a ordenar

Desde $i=1$ a n

intercambiar $A[i]$ con $A[\text{random_value}(i..n)]$

Retornar A

Análisis de uniformidad

En la primera posición $A[1]$

Puede quedar cualquier elemento de A con posibilidad $1/n$

En la segunda posición $A[2]$

Puede quedar cualquier elemento de A menos el que quedo en el primer lugar con posibilidad $1/(n-1)$

En la posición $A[x]$

Puede quedar cualquier elemento menos los $x-1$ que salieron antes: $1/(n-x+1)$

Cualquier permutación

Tiene probabilidad también es $1/n!$

Por lo tanto este método genera una permutación aleatoria uniforme

Ejemplo

Si

$$A = \{a_1, a_2, a_3, a_4\}$$

i	Rand	Intercambio
1	$[1 \text{ a } 4] \rightarrow 2$	$\{a_1, a_2, a_3, a_4\} \rightarrow \{a_2, a_1, a_3, a_4\}$
2	$[2 \text{ a } 4] \rightarrow 3$	$\{a_2, a_1, a_3, a_4\} \rightarrow \{a_2, a_3, a_1, a_4\}$
3	$[3 \text{ a } 4] \rightarrow 3$	$\{a_2, a_3, a_1, a_4\} \rightarrow \{a_2, a_3, a_1, a_4\}$
4	-	$\{a_2, a_3, a_1, a_4\}$

Un pequeño cambio... un significativo cambio

Es un error común

Cambiar el rango de intercambio en cada iteración

Pasar de

```
Desde i=1 a n  
intercambiar A[i] con A[random_value(i...n)]
```

A:

```
Desde i=1 a n  
intercambiar A[i] con A[random_value(1...n)]
```

Pero tiene resultados que invalidan la uniformidad aleatoria

Un pequeño cambio... (cont.)

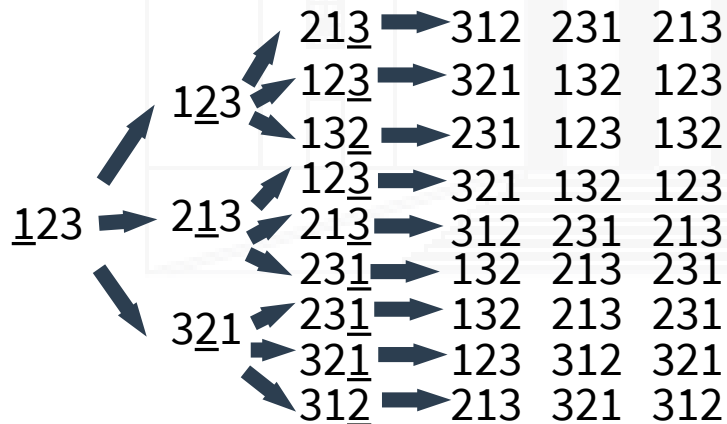
Si tenemos

$$A = \{a_1, a_2, a_3\}$$

Existen 6 ordenamientos posibles

$\{a_1, a_2, a_3\}, \{a_1, a_3, a_2\}, \{a_3, a_2, a_1\}, \{a_3, a_1, a_2\}, \{a_2, a_1, a_3\}, \{a_2, a_3, a_1\}$

Pero utilizando el algoritmo modificado



**No queda
uniformemente
aleatorio!**

Perm	#
$\{a_1, a_2, a_3\}$	4
$\{a_1, a_3, a_2\}$	5
$\{a_3, a_2, a_1\}$	4
$\{a_3, a_1, a_2\}$	4
$\{a_2, a_1, a_3\}$	5
$\{a_2, a_3, a_1\}$	5



Presentación realizada en Junio de 2020

k-conectividad de ejes de un grafo

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Enunciado

Sea

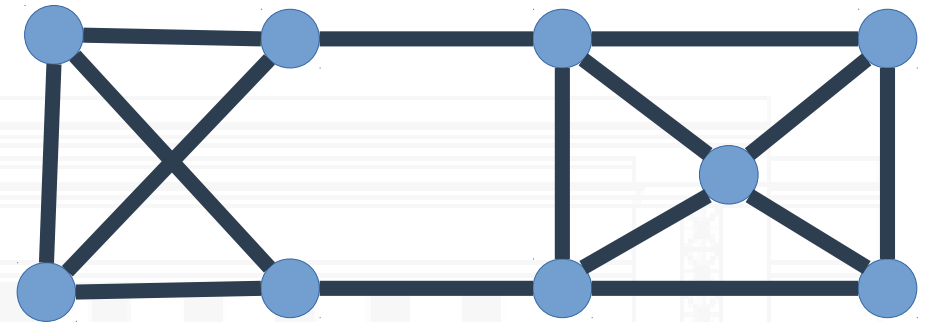
$G=(V,E)$ grafo conexo y no dirigido

Deseamos saber

¿Cuántos ejes se pueden remover antes que G deje de ser conexo?

Este problema se conoce como

K-conectividad de ejes en un grafo



Análisis del problema

Podemos pensar el problema

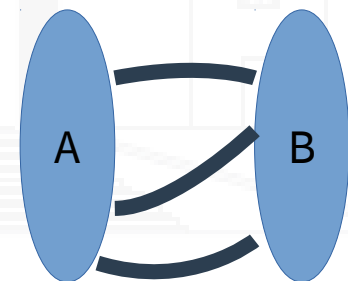
Como encontrar el corte global mínimo del grafo

Analizamos

toda posible subdivisión A-B del grafo en 2 conjuntos disjuntos

Contamos para cada corte

la cantidad de ejes entre conjuntos



Reducción a problema de flujos

Por cada eje $e=(u,v)$

Creamos 2 ejes dirigidos (u,v) y (v,u)

Les asignamos una capacidad de 1

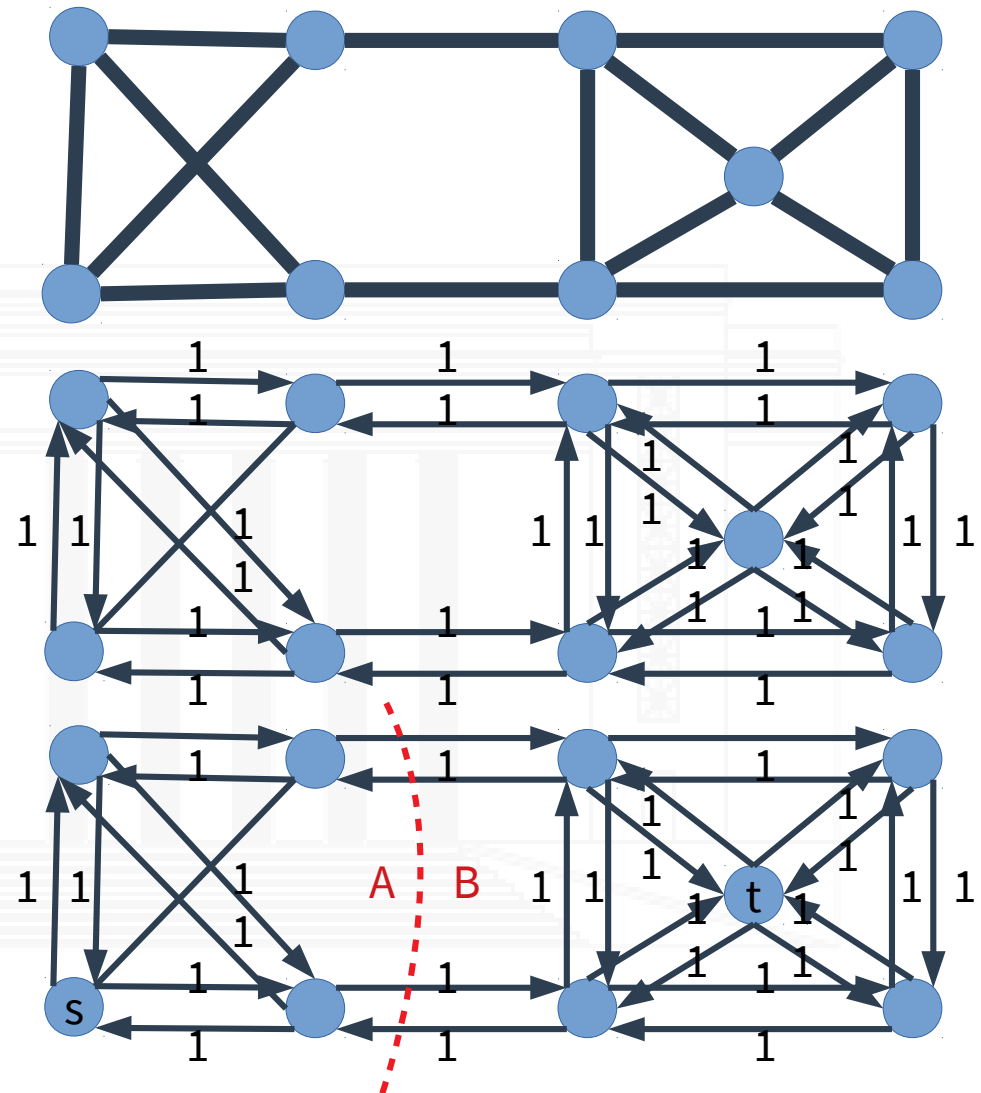
Por cada combinación posible de 2 nodos

Etiquetarlos como s y t respectivamente

Resolver “MAX-FLOW MIN-CUT”

El menor de los cortes mínimos

Corresponde al valor de la K -conectividad de ejes del grafo



Complejidad de la solución

Debemos repetir

El problema de flujo máximo

$$\binom{2}{|V|} = \frac{|V|!}{2! \cdot (|V|-2)!} = O(|V|^2)$$

Cada problema de flujo

Si usamos Ford-Fulkerson $\rightarrow O(C|E|)$,

Siendo C, la sumatoria de las capacidades que salen de la fuente.

Como todas las capacidades son 1, y como mucho puede estar conectado con $|V|-1$ vértices.

Si expresamos la cantidad de ejes en función de los vertices, en el peor de los casos tendremos $\rightarrow |E| = |V| \cdot (|V|-1)/2$

La complejidad será $O(|V|^3)$

Finalmente nos queda como complejidad

$$O(|V|^5)$$

Una mejora de una magnitud

No hace falta realizar

Las $|V|^2$ combinaciones de nodos para fuente y sumideros

Si analizamos el problema, veremos que muchos de los cortes A-B se repiten entre diferente elección del nodo fuente y sumidero

Alcanza

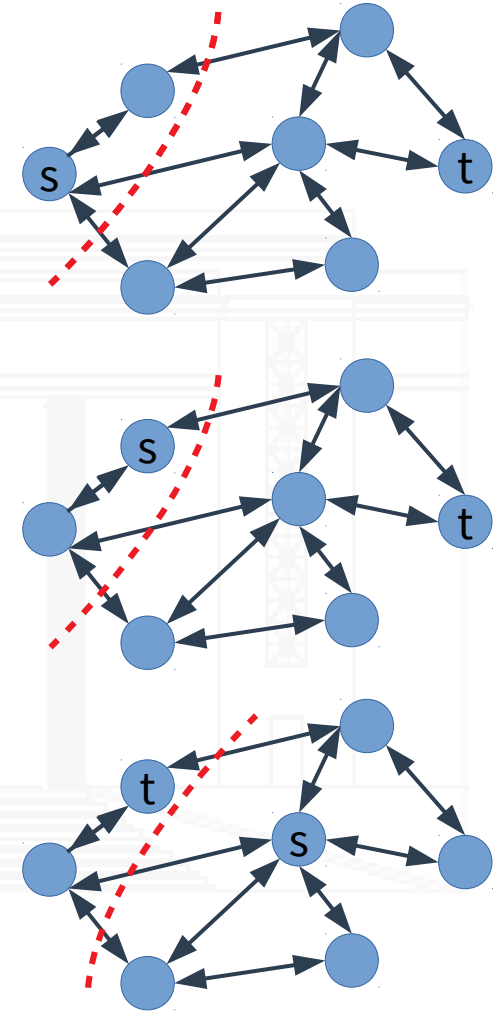
Seleccionando 1 nodo como fuente

Y utilizando en cada subproceso de MAX-FLOW MIN-CUT un sumidero diferente entre los $|V|-1$ nodos restantes

Son $O(|V|)$ en total

La complejidad

Baja a $O(|V|^4)$



Una propuesta superadora

Se puede resolver más rápido?

Propondremos utilizar un algoritmo randomizado

Presentaremos

Karger's algorithm

Propuesto en 1993 por David Karger

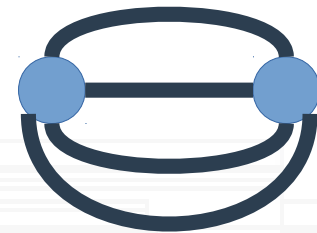
En "Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm"

Algoritmo de Karger

Es un algoritmo

Randomizado que funciona en tiempo polinomial

Y puede retornar un resultado erroneo



Multigrafo de 2
nodos

Funciona

Para multigrafos (pueden existir más de un eje que une a dos nodos)

Utiliza el proceso

De contracción del grafo (por lo que se lo conoce también como “contraction Algorithm”)

Va reduciendo el tamaño del grafo iterativamente

Proceso de contracción

Seleccionar

Un eje $e=(u,v)$ de forma aleatoria y uniforme

Reemplazar

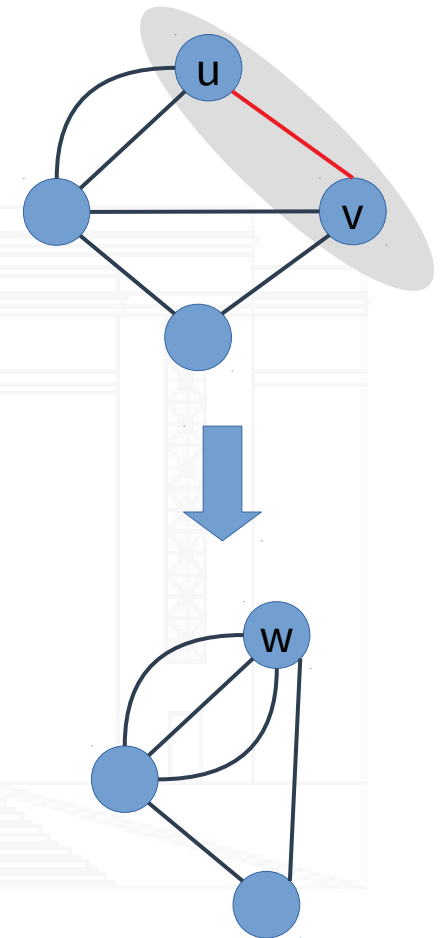
los nodos u y v por un nuevo nodo w

Todos los ejes (u,v) se eliminan

Los ejes (u,a) y (v,a) con $a \in E - \{u,v\}$ se reemplaza por (w,a)

Repetir

Hasta que solo queden 2 nodos en el grafo G resultante



Funcionamiento

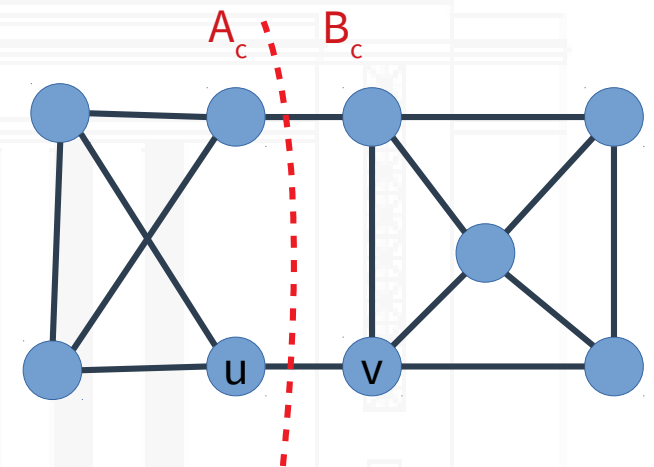
Si consideramos el corte mínimo “C”

Divide el grafo en 2 conjuntos que llamaremos A_c y B_c

El algoritmo no encontrará el valor k

Si en alguna iteración contrae nodos u, v que se encuentran en A_c y B_c respectivamente

Para eso debe seleccionar un eje que pase el corte mínimo global



Solo hay 2 entre 15 de posibilidad de seleccionar un eje en el corte mínimo

Fin de la ejecución

Al finalizar la ejecución

Quedarán 2 “super nodos”

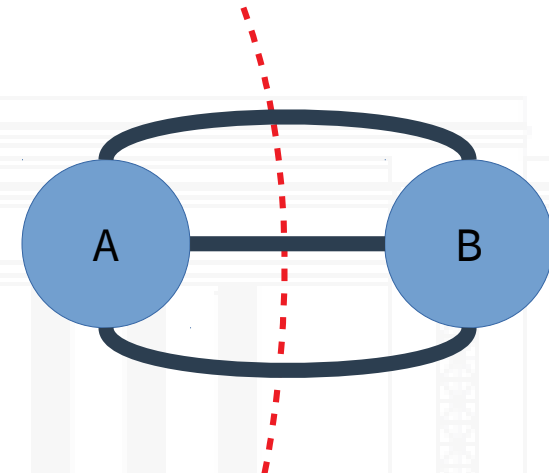
Con ejes entre ellos

Representan

Un posible corte A-B

La cantidad de ejes entre A y B

Son “probablemente” la K-conectividad de ejes del grafo



Pseudocódigo y Complejidad

La iteración principal

Se ejecuta $|V|-2$ veces

La construcción del nuevo grafo

Tiene como cota $O(|V|)$ si se utiliza una matriz de adyacencia para representar el grafo

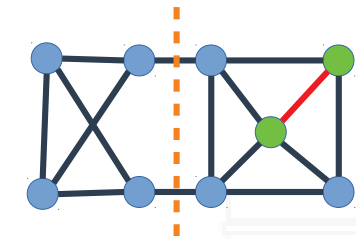
Una ejecución del algoritmo

Tiene una complejidad temporal $O(|V|^2)$

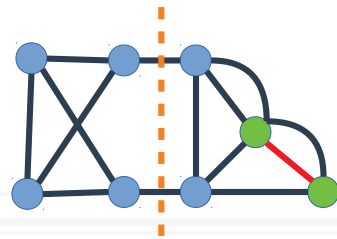
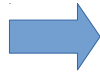
Y una complejidad espacial $O(|V|^2)$

```
Mientras  $|G.V| > 2$   
    Seleccionar aleatoriamente eje  
         $e = (u, v)$  de  $G.E$   
  
    Contraer  $G$  mediante  $e$   
  
Retornar  $|G.E|$ 
```

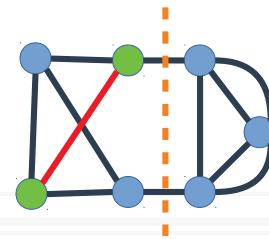
Ejemplo



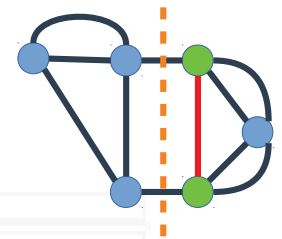
Probabilidad de
elegir eje en corte:
 $\frac{2}{15}$



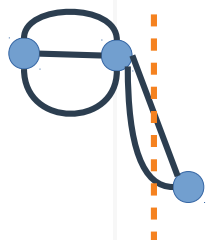
Probabilidad de
elegir eje en corte:
 $\frac{2}{14}$



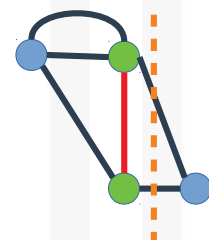
Probabilidad de
elegir eje en corte:
 $\frac{2}{12}$



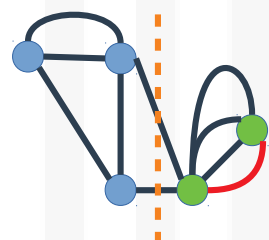
Probabilidad de
elegir eje en corte:
 $\frac{2}{11}$



Probabilidad de
elegir eje en corte:
 $\frac{2}{5}$

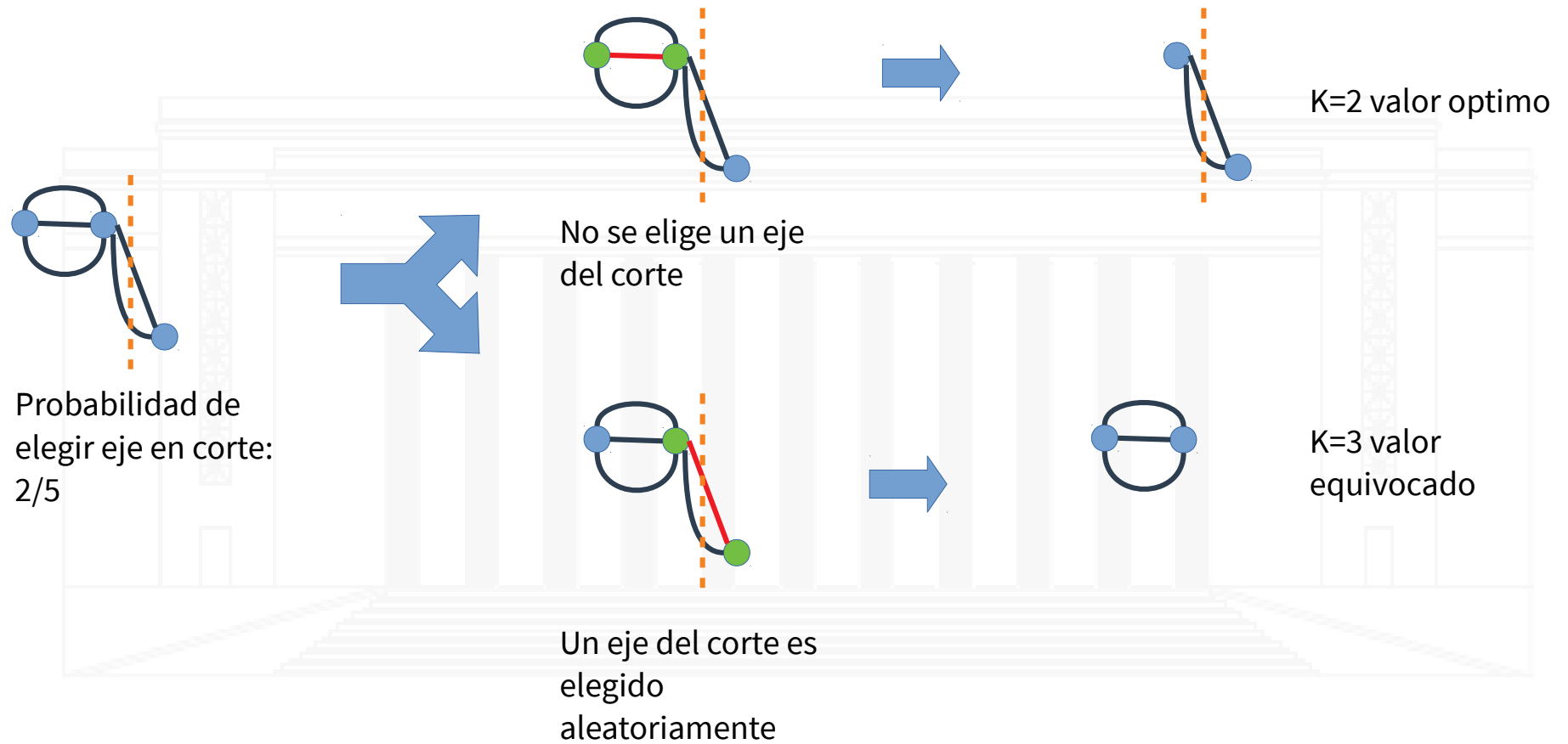


Probabilidad de
elegir eje en corte:
 $\frac{2}{6}$



Probabilidad de
elegir eje en corte:
 $\frac{2}{10}$

Ejemplo (cont.)



Probabilidad de éxito

Llamaremos

ε al evento de encontrar el corte mínimo global

Para lograr ε

Se tiene que cumplir que en cada iteración no se selecciona un eje de "O"

Llamaremos ε_k

Al evento de en la iteración k no contraer un eje de "O"

Probabilidad de éxito (cont.)

Por lo tanto

$$\varepsilon = \varepsilon_1 \cap \varepsilon_2 \cap \dots \cap \varepsilon_{n-2}$$

La probabilidad que ocurra el evento ε

$$\Pr(\varepsilon) = \Pr(\varepsilon_1 \cap \varepsilon_2 \cap \dots \cap \varepsilon_{n-2})$$

$$\Pr(\varepsilon) = \Pr(\varepsilon_1) * \Pr(\varepsilon_2 / \varepsilon_1) * \dots * \Pr(\varepsilon_{n-2} / \varepsilon_{n-3}, \varepsilon_{n-4}, \dots, \varepsilon_1)$$

Probabilidad de éxito (cont.)

En la primera iteración

$$\Pr(\varepsilon_1) = 1 - \Pr(\bar{\varepsilon}_1)$$

Donde

$$\Pr(\bar{\varepsilon}_1) = k / |E|$$

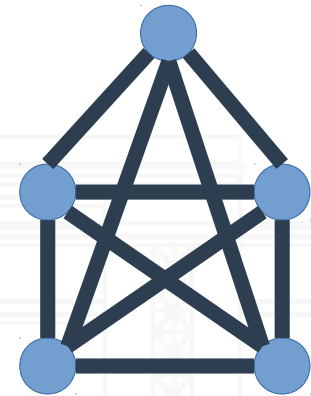
Podemos acotar

$|E| \geq k|V| / 2$ (pues todos los nodos tienen al menos k ejes)

Por lo tanto

$$\Pr(\bar{\varepsilon}_1) \leq k / (k|V| / 2) = 2 / |V|$$

$$\Pr(\varepsilon_1) \geq 1 - (2 / |V|) = (|V| - 2) / |V|$$



En un grafo completo:

$$k = |V| - 1$$

$$|E| = |V| * (|V| - 1) / 2$$

Probabilidad de éxito (cont.)

Hasta el momento

acotamos $\Pr(\varepsilon_1)$

De

$$\Pr(\varepsilon) = \Pr(\varepsilon_1) * \Pr(\varepsilon_2 / \varepsilon_1) * \dots * \Pr(\varepsilon_{n-2} / \varepsilon_{n-3}, \varepsilon_{n-4}, \dots, \varepsilon_1)$$

¿Cómo calculamos

$$\Pr(\varepsilon_i / \varepsilon_{i-1}, \varepsilon_{i-2}, \dots, \varepsilon_1) ?$$

Probabilidad de éxito (cont.)

Al momento del evento ε_i

Quedan $|V|-i+1$ nodos

Cada uno de esos nodos

Deben tener al menos k ejes incidentes

(es absurdo que algo menor, o el algoritmo podría retornar un valor menor a k !
... y el algoritmo de contracción no funciona de esa manera)

Por lo tanto al momento de ε_i

$$|E| \geq k * (|V|-i+1) / 2$$

Probabilidad de éxito (cont.)

Con

$$|E| \geq k * (|V| - i + 1) / 2$$

Podemos acotar

$$\Pr(\bar{\epsilon}_i / \epsilon_{i-1}, \epsilon_{i-2}, \dots, \epsilon_1) \leq k / (k * (|V| - i + 1) / 2) = 2 / (|V| - i + 1)$$

$$\Pr(\epsilon_i / \epsilon_{i-1}, \epsilon_{i-2}, \dots, \epsilon_1) \geq 1 - 2 / (|V| - i + 1) = (|V| - i - 1) / (|V| - i + 1)$$

... y con esto podemos calcular la probabilidad total

Probabilidad de éxito (cont.)

La probabilidad de encontrar con éxito k

$$\Pr(\varepsilon) = \Pr(\varepsilon_1) * \Pr(\varepsilon_2 / \varepsilon_1) * \dots * \Pr(\varepsilon_{n-2} / \varepsilon_{n-3}, \varepsilon_{n-4}, \dots, \varepsilon_1)$$

$$\Pr(\varepsilon) \geq \frac{|V| - (|V| - 2) - 1}{|V| - (|V| - 2) + 1} * \frac{|V| - (|V| - 3) - 1}{|V| - (|V| - 3) + 1} * \dots * \frac{|V| - 2}{|V|} = \frac{1}{3} * \frac{2}{4} * \dots * \frac{|V| - 2}{|V|}$$

$$\Pr(\varepsilon) \geq \frac{2! * (|V| - 2)!}{|V|!} = \binom{|V|}{2}^{-1} = 1 / \binom{|V|}{2}$$

$$\Pr(\varepsilon) \geq \frac{2}{|V| * (|V| - 1)}$$

Si el grafo es muy grande

... parece ser una probabilidad muy pequeña

Mejorar las chances de exito

Podemos

Ejecutar varias veces el programa y quedarnos con el mejor resultado encontrado.

Cada ejecución

Aumenta la probabilidad de hallar k

Podemos medir

La probabilidad que en n iteraciones no hallemos k como $\left(1 - 1/\binom{|V|}{2}\right)^n$

Mejorar las chances de éxito (cont.)

¿Cuántas iteraciones

Debemos realizar hasta tener una alta probabilidad?

Partiendo de

$$\left(1 - \frac{1}{\binom{|V|}{2}}\right)^n \rightarrow \left(1 - \frac{1}{\binom{|V|}{2}}\right)^{\binom{|V|}{2}} \leq \frac{1}{e}$$

Por lo tanto

si ejecuto $n = \binom{|V|}{2} * \ln |V|$, entonces

$$\left(1 - \frac{1}{\binom{|V|}{2}}\right)^{\binom{|V|}{2} * \ln |V|} \leq \left(\frac{1}{e}\right)^{\ln |V|} = 1/|V|$$

Hint:

$$\left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}, x \geq 1$$

Complejidad total

Ejecutar

el algoritmo Karger $O(|V|^2 * \log |V|)$ veces

Nos da una complejidad temporal

$$O(|V|^4 * \log |V|)$$

Todo esto y funciona peor comparado al método anterior ????

Y asegura con alta probabilidad

¿¿¿Y encima puede fallar???

Encontrar el valor k

ALTO! Esto no termina aca....

Mejoras: steiner-karger

En 1996

David Karger y Clifford Stein

Publicaron

“A new approach to the Minimum cut problem”

<http://www.columbia.edu/~cs2035/courses/ieor6614.S09/Contraction.pdf>

Es una variante del anterior

Si $|V|$ es pequeño resuelve por fuerza bruta

Sino contrae $|V| / \sqrt{2}$ nodos y aplica técnica de división y conquista con el resto

Encuentra con alta probabilidad k

con en $O(|V|^2 \log^3 |V|)$ tiempo



Presentación realizada en Junio de 2020

Resolución de conflictos en sistemas distribuidos

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Resolución de conflictos en sistemas distribuidos

Tenemos n procesos $\{P_1, \dots, P_n\}$

Cada proceso no tiene comunicación con el resto.

Pueden solicitar acceso a un mismo recurso (ej: base de datos)

Las solicitudes se realizan en rondas discretas.

Si solo 1 proceso pide el recurso se concede el acceso

Si mas de un proceso solicita acceso, ninguno lo obtiene.

Cómo podemos resolver los accesos y maximizar la probabilidad de obtener el recurso?

Quiebre de simetría

Si nadie solicita el recurso

El turno se “pierde”

Si solo 1 proceso solicita el recurso

Lo obtendrá con éxito

Si mas de 1 proceso solicita el recurso

NINGUNO lo obtiene y deberá volver a solicitarlo

Si cada vez que un proceso falla,

Solicita el proceso inmediatamente en el turno siguiente

Se provocará un atasco. ← queremos evitar que esto ocurra

Una solución mediante algoritmo randomizado

Sea $p > 0$

la probabilidad de que un proceso solicita acceso en un determinado turno

La probabilidad es independiente entre cada proceso.

En cada ronda los procesos deciden acceder o no en base a p

Si en un turno un proceso pide el recurso y falla,

No lo pedirá determinísticamente en el siguiente.

Se aplicará la misma probabilidad “ p ” para solicitarlo.

Cómo se comporta el sistema? Cual deber ser el valor de p ?

Evento de intento de acceso

Llamaremos al evento $A[i,t]$

Intento del proceso P_i de acceder al recurso en la ronda t

Sabemos que estará signado por la probabilidad p

Podemos determinar tanto el intento como el no-intento:

$$\text{Prob}(A[i,t]) = p$$

$$\overline{\text{Prob}(A[i,t])} = 1 - p$$

Evento de acceso con éxito

Llamaremos $S[i,t]$ al evento de éxito en el acceso

Implica que el proceso P_i intento acceder al recurso en la ronda t

Ningún otro proceso lo intento en esa misma ronda

Podemos expresarlo como:

$$S[i,t] = A[i,t] \cap (\cap \overline{A[j,t]}), j \neq i$$

Su probabilidad:

$$Prob(S[i,t]) = Prob(A[i,t]) \cdot \prod_{j \neq i} Prob(\overline{A[j,t]}) = p * (1 - p)^{(n-1)}$$

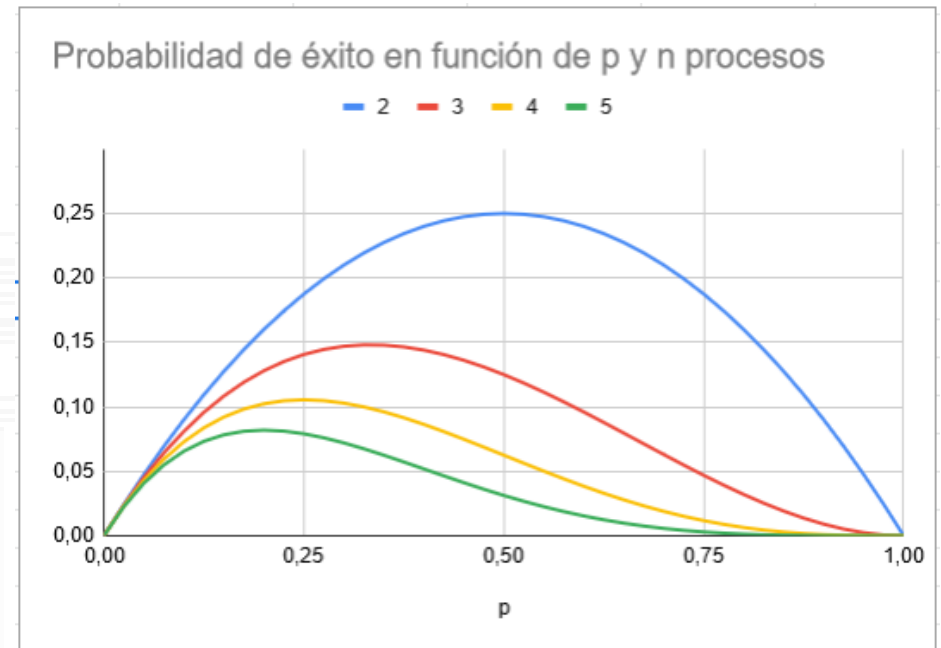
Maximizar la probabilidad de éxito

Sea $f(p) = p * (1 - p)^{(n-1)}$

P es un valor entre 0 y 1

$f(p=0) = 0 \leftarrow$ Nunca intento

$f(p=1) = 0 \leftarrow$ Siempre intento



Maximización de $f(p)$,

cálculo de la derivada $f'(p) = (1 - p)^{n-1} - (n - 1)p(1 - p)^{n-2}$

La derivada

tiene un cero en $p = 1/n \rightarrow$ con ese valor maximizamos

Maximizar la probabilidad de éxito (cont.)

Una vez que tenemos p

Podemos reemplazarlo en nuestra probabilidad

$$Prob(S[i, t]) = \frac{1}{n} * \left(1 - \frac{1}{n}\right)^{(n-1)}$$

Como se comporta esta función

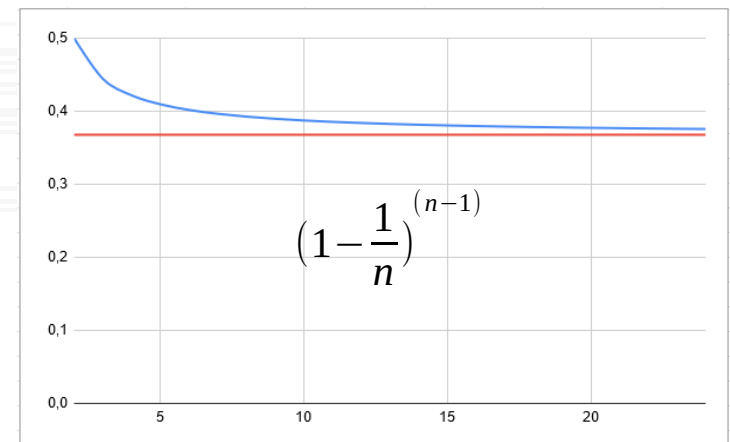
En función de n?

Si analizamos $\left(1 - \frac{1}{n}\right)^{(n-1)}$

Veremos que (para $n > 1$) inicia en $\frac{1}{2}$ y converge monotonamente a $1/e$

Por lo que podemos acotar

$$\frac{1}{en} \leq Prob(S[i, t]) \leq \frac{1}{2n} \quad \longrightarrow \quad Prob(S[i, t]) = \Theta\left(\frac{1}{n}\right)$$



¿Cuánto tardará un proceso en tener éxito?

Si tomamos el proceso P_i

¿Cuanto tardara en acceder al recurso?

Si n es grande

Difícilmente lo consiga en su primer intento

Llamaremos $F[i,t]$ Fallo (del protocolo) para el proceso i

Si luego de t rondas el proceso i aun no pudo acceder al recurso

¿Cuánto tardará un proceso en tener éxito? (cont.)

Podemos calcular

$$\Pr(F[i,t]) = \overline{\Pr(S[i,1])} * \overline{\Pr(S[i,2])} * \dots * \overline{\Pr(S[i,t])}$$

Con

$$\overline{\text{Prob}(S[i,t])} = 1 - \text{Prob}(S[i,t])$$

$$\frac{1}{en} \leq \text{Prob}(S[i,t]) \leq \frac{1}{2n}$$

Por lo tanto

$$\Pr(F[i,t]) \leq \left(1 - \frac{1}{en}\right)^t$$

¿Cuánto tardará un proceso en tener éxito? (cont.)

Si elegimos $t = \lceil en \rceil$

$$Pr(F[i, t]) \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e}$$

La probabilidad de que el proceso P_i no tenga éxito entre $t=1$ y $\lceil en \rceil$

Está acotada por e^{-1} (de forma independiente a n)

¿Cuánto tardará un proceso en tener éxito? (cont.)

Si elegimos t ligeramente superior a $\lceil en \rceil$

Por ejemplo $\lceil en \rceil * (c * \ln n)$

$$Pr(F[i, t]) \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil^{c * \ln n}} \leq e^{-c \ln n} = n^{-c}$$

La probabilidad de que el proceso P_i no tenga éxito entre $t = \lceil en \rceil$ y $\Theta(n \ln n)$

Desciende precipitadamente

¿Cuánto tardará un proceso en tener éxito? (cont.)

En conclusión

Antes de $\Theta(n)$ rondas la probabilidad de fracaso está acotada por una constante

Luego, entre $\Theta(n)$ y $\Theta(n \ln n)$ rondas la probabilidad cae, delimitado por un polinomio inverso en n .

¿Cuánto tardarán todos los procesos en tener éxito?

Llamaremos fallo general del protocolo $F[t]$

Si luego de t rondas al menos un proceso aun no pudo acceder al recurso

¿Qué valor debe tener t

Para que la probabilidad de fallo general del protocolo sea razonablemente pequeña?

$$Pr(F[t]) = Pr(F[1,t] \cup F[2,t] \cup \dots \cup F[n,t])$$

Es la unión de eventos no independientes. Es complejo de calcular. Pero podemos determinar una cota

¿Cuánto tardarán todos los procesos en tener éxito?

Se puede ver que

$$Pr(F[t]) = Pr(F[1,t] \cup F[2,t] \cup \dots \cup F[n,t]) \leq \sum_{i=1}^n Pr(F[i,t])$$

Las probabilidades de fallo

de cada proceso son iguales

Tenemos

n procesos \rightarrow sumamos n veces $F[i,t]$

Para que la probabilidad de fracaso sea pequeña

$F[i,t]$ tiene que ser significativamente menor a $1/n$

¿Cuánto tardarán todos los procesos en tener éxito?

Antes de $\Theta(n)$ rondas la $F[i,t]$ está acotada por una constante

$$Pr(F[t]) \leq \sum_{i=1}^n Pr(F[i,t]) \leq nc$$

No logramos la cota requerida

Si tomamos $t = \lceil en \rceil * (2 * \ln n) \leftarrow c=2$

$$Pr(F[t]) \leq \sum_{i=1}^n Pr(F[i,t]) \leq n * n^{-2} = \frac{1}{n}$$

logramos la cota que nos solicitamos

¿Cuánto tardarán todos los procesos en tener éxito?

En conclusión

Con probabilidad de al menos $1 - 1/n$

Todos los procesos

Tienen éxito en acceder al recurso al menos 1 vez

En no mas de

$t = \lceil \ln l \rceil * (2 * \ln n)$ rondas



Presentación realizada en Junio de 2020

K-esimo elemento y Quicksort randomizado

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Calculo de la mediana

Sea

un set de números $n = \{a_1, a_2, \dots, a_n\}$

La mediana

es el numero que queda en la posición del medio si se presentan ordenados

Ej:

$10, 6, 2, 15, 1 \rightarrow 1, 2, 6, 10, 15$

Formalmente

El k-esimo numero más grande de S tal que

$k = (n+1)/2$, si n es par

$K = n/2$, si n es impar

Solución determinística

Podemos calcularlo mediante:

Sort $\rightarrow O(n \log n)$

Podemos hacerlo mejor?

Usaremos un algoritmo randomizado + división y conquista

Encontrar el k-esimo elemento

Sea

El set S de n números,

Un número k entre 1 y n

La función $\text{SELECT}(S,k)$

Retorna el k -esimo mayor elemento.

Casos particulares:

Mediana: $k = n/2$ o $(n+1)/2$

Minimo $k=1$

Máximo $k=n$

División del problema: El pivot

Seleccionar un elemento $a_i \in S$ como pivot Y formar 2 sets

$$S^- = \{a_j : a_j < a_i\}$$

$$S^+ = \{a_j : a_j > a_i\}$$

Pueden ocurrir 3 cosas:

Si $|S^-| = k-1$ entonces a_i es el k -esimo elemento.

Si $|S^-| > k-1$ nos quedamos con S^- y repetimos el Proceso

Si $|S^-| < k-1$ nos quedamos con S^+ y repetimos, buscando el $k-1-|s^-|$ elemento

Algoritmo

```
select(S,k)
  S1={} Sr={}
  p = calcularpivot(k)
  Desde j=1 a k
    Si sj < p
      S1 += {sj}
    Si sj > p
      Sr+={sj}

  si size(s1) = k-1
    return p;
  Sino si size(s1) > k-1
    select (s1 , k)
  sino
    select (sr, k - 1 - size(s1))
```

Análisis del pivot

Si pudiésemos seleccionar el pivot justo como el valor medio

Siempre nos quedaríamos con la mitad de los elementos.

Nos quedaría una recurrencia como: $T(n) = T(n/2) + cn$

Aplicando el teorema maestro: $O(n)$

En el peor de los casos si selecciono el menor (o mayor de los elementos

La recurrencia me queda $T(n) = T(n-1) + cn$

Nos quedaría una complejidad de $O(n^2)$

Tenemos que intentar seleccionar un pivot “central”!

Pivot “centrado”

Podemos intentar seleccionar un pivot que al menos $\varepsilon * n$ elementos menores y mayores que él ($\varepsilon > 0$)

La recurrencia me quedaría como: $T(n) = T((1-\varepsilon)n) + cn$

La complejidad nos quedaría lineal.

Cuanto mas central, mas rápido achicamos el conjunto analizado.

Una elección al azar

Proponemos seleccionar un $a_i \in S$ como pivot uniformemente al azar

Consideramos centrales a los elementos que al menos dejan $\frac{1}{4}$ de los elementos del lado izquierdo o a la derecha

La mitad de los elementos son centrales ($\epsilon=1/4$)

La probabilidad de seleccionar un pivot central es de $\frac{1}{2}$

Al dividir en S^+ y S^- verificamos que la división cumpla el requisito

Si no cumple, volvemos a seleccionar al azar otro pivot

Probabilísticamente tendría que repetir a lo sumo 2 veces la elección

Este proceso es $O(n)$

Análisis de cada fase

En cada fase j del algoritmo

Reduzco al menos en $\frac{1}{4}$ el tamaño del problema

El tamaño del conjunto que estoy analizando esta acotado por

$$n\left(\frac{3}{4}\right)^{j+1} \leq |S'| \leq n\left(\frac{3}{4}\right)^j$$

La cantidad de pruebas de pivot esperado es a lo sumo 2.

Análisis global

En cada fase X_j

la cantidad de elementos es a lo sumo $n(3/4)^j$

La cantidad de operaciones en cada fase son lineales $c \cdot n(3/4)^j$

Se espera que la cantidad de repeticiones de cada fase sea 2, entonces $E[X_j] \leq 2c \cdot n(3/4)^j$ (esperanza de la fase X_j)

El algoritmos esta conformado por una sucesión de fases

$$X = X_0 + X_1 + X_2 + \dots$$

La esperanza total es
$$E[X] \leq \sum_j E[X_j] \leq \sum_j 2cn \left(\frac{3}{4}\right)^j = 2cn \sum_j \left(\frac{3}{4}\right)^j \leq 8cn$$

QuickSort

Es un algoritmo de ordenamiento

creado por C. A. R. Hoare.

Utiliza

Division y conquista

Divide en cada paso

En 2 subproblemas utilizando un valor pivot

Por un lado se procesan los valores menores al pivot y por el otro los mayores

Pseudocódigo

```
QuickSort(S)
  Si  $|S| \leq 3$ 
    Ordenar S
    Retornar S
  Sino
     $p = \text{seleccionarPivot}(S)$ 
    Por cada elemento de S
      Ponerlo en S- si es menor a p
      Ponerlo en S+ si es mayor a p

    S- = QuickSort(S-)
    S+ = QuickSort(S+)

  Retornar S-, p, S+
```


Análisis de la solución

La eficiencia de la solución

Depende de la selección del pivot

Si el pivot es el valor medio

Divide los problemas en partes iguales

Queda una recurrencia $T(n) = 2T(n/2) + O(n)$

Que es $O(n \log n)$

Si el pivot es “malo”,

Deja separado en tamaños de subproblemas muy dispares

En el peor caso nos queda una recurrencia $T(n) = T(n-1) + O(n)$

Que es $O(n^2)$

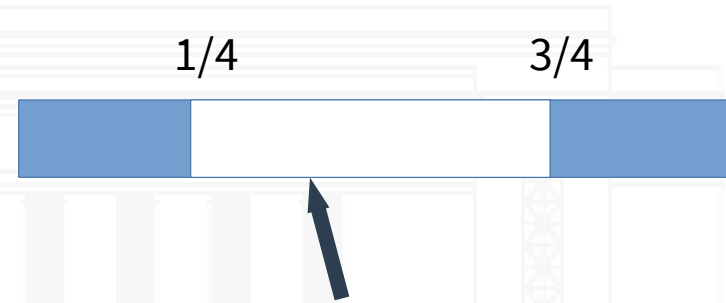
Quicksort randomizado

Modificaremos Quicksort

Intentaremos elegir el pivot aleatoriamente

Queremos que sea “central”

Ni entre $\frac{1}{4} * |S|$ inicial ni final



Al dividir en S^+ y S^- verificamos que la división cumpla el requisito

Si no cumple, volvemos a seleccionar al azar otro pivot

Probabilisticamente tendría que repetir a lo sumo 2 veces la elección

Pseudocódigo randomizado

```
QuickSort(S)
  Si  $|S| \leq 3$ 
    Ordenar S
    Retornar S
  Sino
    Repetir
       $p = \text{seleccionarAleatoriamentePivot}(S)$ 
      Por cada elemento de S
        Ponerlo en S- si es menor a p
        Ponerlo en S+ si es mayor a p
      Hasta que  $|S-| \geq 1/4 * |S|$  y  $|S+| \geq 1/4 * |S|$ 
    S- = QuickSort(S-)
    S+ = QuickSort(S+)
  Retornar S-, p, S+
```

Complejidad

En cada fase iteración j

la cantidad de elementos es a lo sumo $|S|(3/4)^j$

La cantidad de operaciones en cada fase son lineales $c \cdot |S|(3/4)^j$

Se espera que la cantidad de repeticiones de cada fase sea 2.

Hay a lo sumo $O(\log|S|)$

Iteraciones internas

Por lo tanto

El proceso total es $O(|S|\log|S|)$



Presentación realizada en Junio de 2020

Algoritmo Freivalds

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Verificador de multiplicador de matrices

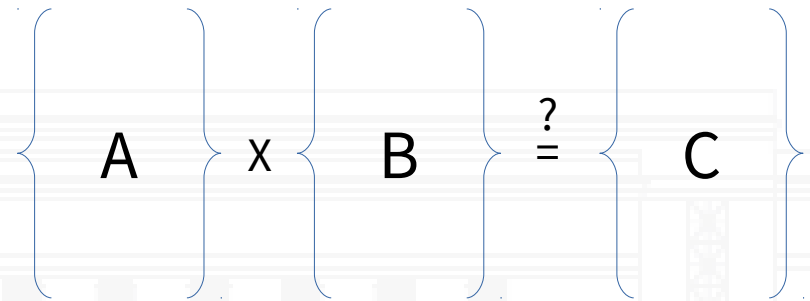
Sea:

las Matrices A y B de $n \times n$

la matriz C,

Queremos verificar

$$C = A \times B$$


$$\left\{ A \right\} \times \left\{ B \right\} \stackrel{?}{=} \left\{ C \right\}$$

Algoritmos de multiplicación conocidos

Algoritmos de multiplicación:

“naive”: $O(n^3)$

Strassen (1969): $O(n^{\log_2 7})$

Le Gall (2014): $O(n^{2,37286})$

...

?? (??): $O(n^w)$, $w \geq 2$

(!) Cuanto menor w , mayor la constante k del algoritmo (recién para n muy grandes hay ganancia real en su aplicación)

Algoritmo Freivalds

Sea un vector r $[1..n]$

Tal que $r_i = \{0,1\}$ con equiprobabilidad para todo i

Calcular:

$$D = A \times (B \times r) - (C \times r)$$

Si $D = \text{vector cero} \Rightarrow A \times B$ es probablemente C (retorna “si”)

Si $D \neq \text{vector cero} \Rightarrow A \times B$ NO es c (retorna “no”)

$$\left\{ \begin{matrix} D \end{matrix} \right\} = \left\{ \begin{matrix} A \end{matrix} \right\} \times \left\{ \begin{matrix} B \\ r \end{matrix} \right\} - \left\{ \begin{matrix} C \\ r \end{matrix} \right\}$$

Algoritmo Freivalds (cont.)

Es un algoritmo de tipo Montecarlo

La complejidad en tiempo es $O(N^2)$

Si $AxB = C \Rightarrow \Pr[\text{resp}=\text{si}] = 1$

Esta afirmación es trivial. Para cualquier r seleccionado $(AxB)_{xr} = (C)_{xr}$

Si $AxB \neq C \Rightarrow \Pr[\text{resp}=\text{si}] \leq 1/2$

Requiere una demostración

Falsos positivos

Afirmación:

Si $AB \neq C \Rightarrow \text{Prob}[ABr \neq Cr] \geq \frac{1}{2}$

Hipotesis:

Sea $D = AB - C$ tal que $D \neq 0$

Queremos mostrar que hay muchos r tal que $Dr \neq 0$,
específicamente $\text{Prob}[Dr \neq 0] \geq \frac{1}{2}$ para un r elegido aleatoriamente

Probaremos que para cada $Dr = 0$ donde $D \neq 0$, existe un r' tal que $Dr' \neq 0$ y $D \neq 0$

Falso positivo

Negativo

Falsos positivos (cont.)

$$D = AB - C \neq 0$$

Existe i, j tal que $d_{ij} \neq 0$

Seleccionamos un vector v con $v_j=1$ y $v_{x \neq j}=0$

Vemos que $Dxv = Dv \neq 0$

Sea cualquier r que pueda ser elegido aleatoriamente por el algoritmo tal que $Dr=0$,

Sea $r' = r + v \Rightarrow Dr' = D(r+v) = 0 + Dv \neq 0$

r con r' tienen una relacion de 1 a 1 (al tener solo 1 elemento “switchheado”)

Por lo tanto el numero de $r' / Dr' \neq 0 \geq$ numero de $r / Dr=0$

Finalmente $\text{Prob}[Dr \neq 0] \geq 1/2$

$$\begin{matrix} & j \\ \begin{matrix} i \\ \vdots \\ \end{matrix} & \begin{pmatrix} & | & \\ \hline & \text{blue square} & \\ \hline & | & \end{pmatrix} & \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ d_{ij} \\ \vdots \\ 0 \end{pmatrix} \\ & \mathbf{D} & \mathbf{v} & \mathbf{Dv} \end{matrix}$$

Ejemplo

$$A = \begin{Bmatrix} 2 & 3 \\ 3 & 4 \end{Bmatrix} \quad B = \begin{Bmatrix} 1 & 0 \\ 1 & 2 \end{Bmatrix} \quad C = \begin{Bmatrix} 6 & 5 \\ 8 & 7 \end{Bmatrix}$$

$$r = \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} \rightarrow \begin{Bmatrix} 2 & 3 \\ 3 & 4 \end{Bmatrix} \begin{Bmatrix} 1 & 0 \\ 1 & 2 \end{Bmatrix} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} - \begin{Bmatrix} 6 & 5 \\ 8 & 7 \end{Bmatrix} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

Falso positivo!

Resp: "Si"

$$r = \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} \rightarrow \begin{Bmatrix} 2 & 3 \\ 3 & 4 \end{Bmatrix} \begin{Bmatrix} 1 & 0 \\ 1 & 2 \end{Bmatrix} \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} - \begin{Bmatrix} 6 & 5 \\ 8 & 7 \end{Bmatrix} \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} = \begin{Bmatrix} -1 \\ -1 \end{Bmatrix}$$

Resp: "No"

Basta encontrar una respuesta en "no" para determinar que $AxB \neq C$.

Margen de error

Para disminuir la posibilidad de los falsos positivos podemos ejecutar k veces el mismo

El orden de complejidad será $O(kn^2)$

La probabilidad de falso positivo sera $\leq 1/2^k$

Cuanto mayor k , la probabilidad tiende a cero

Si $k=n$, la complejidad pasa a ser n^3



Presentación realizada en Junio de 2020

Diccionarios randomizados

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Diccionarios

Existe

Un universo U de posibles elementos extremadamente grande

Queremos una estructura para

manipular un subconjunto S de U de tamaño apreciablemente menor.

Definiremos

en n elementos de S como la cantidad máxima a manipular.

Deseamos

almacenarlos, recuperarlos, y/o eliminarlos en tiempo constante por operación

Funciones de Hashing

Sea

H un Hash Table, vector de n posiciones.

la función $h:U \rightarrow \{0,1,\dots,n-1\}$ mapea un elemento de U a una posición

Cada elemento $u \in U$ a almacenar

se almacena en la posición de H que indica $h(u)$

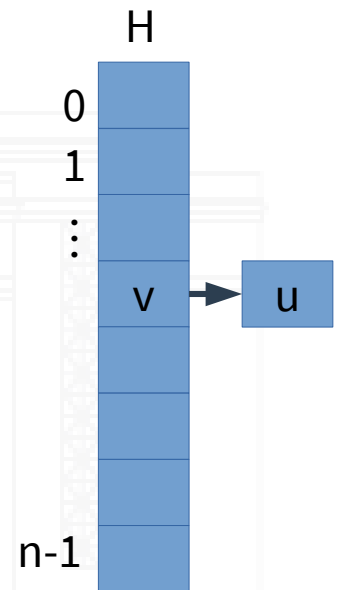
Sinónimos y colisiones

Se conoce como sinónimos

$$a\ u, v \in U / h(v) = h(u)$$

Si 2 o más sinónimos se intentan almacenar en H

Se produce una colisión: se deben almacenar en el mismo lugar.



La estructura, debe lidiar con las colisiones

Existen diferentes maneras de hacerlo

Por ejemplo: almacenar cada sinónimo como una lista enlazada

Tiempos de acceso

Si H no tiene almacenado sinónimos

El tiempo por operación es calcular $h(u) \leftarrow O(1)$

Si H tiene colisiones

El tiempo por operación es $h(u) +$ recorrer la lista de sinónimos en la posición $H(h(u))$

Queremos que

la función de hashing distribuya lo mejor posible los elementos en el intervalo de H

Evitando que la tabla H contenga muchos sinónimos

Una buena función de hashing

Queremos

minimizar la probabilidad de colisiones.

Se suelen utilizar

funciones del estilo $h(u): u \bmod p$, con p primos.

Funcionan bien empíricamente y para la mayoría de los subconjuntos de elementos

Deseamos que

que podamos probar su eficiencia con alta probabilidad

Una buena función de hashing (cont.)

Nos atrae

la idea de distribuir uniformemente y para eso utilizar una función aleatoria

No podemos incluir lo probabilístico en las posiciones,

sino no podemos asegurar volver a encontrar un elemento almacenado.

Queremos que la posibilidad de una colisión

Sea probabilísticamente pequeña

Clase universal de funciones de Hashing

Una familia de funciones de hashing \mathcal{H}

debe cumplir 2 condiciones

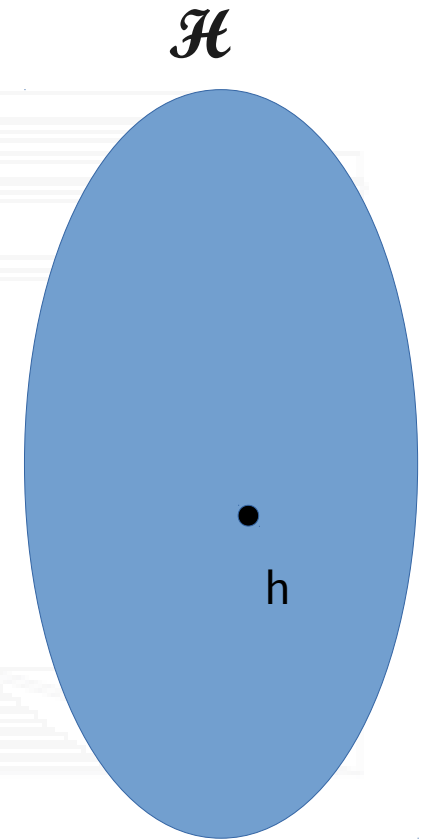
Cada h de \mathcal{H}

se debe representar de manera compacta y
se debe calcular de forma eficiente.

Para cualquier $u, v \in U$

la probabilidad, seleccionando h de la familia \mathcal{H} al azar,

De que $h(u) = h(v)$ es a lo sumo $1/n$



Cantidad de colisiones esperadas

Sea

\mathcal{H} una clase de funciones de hashing universales que mapea el universo U al set $\{0, 1, \dots, n-1\}$,

S un subset arbitrario de U de tamaño máximo n

$u \in U$ (cualquier elemento)

Definimos

la variable aleatoria X como la cantidad de elementos $s \in S$ para los que $h(s) = h(u)$ para una selección aleatoria de $h \in \mathcal{H}$

La variable aleatoria X_s para un elemento aleatorio $s \in S$, igual a 1 si $h(s)=h(u)$, sino igual a 0

Cantidad de colisiones esperadas (cont.)

Como $h \in \mathcal{H}$

$$E[X_s] = \Pr[X_s = 1] \leq 1/n$$

Entonces

$$E[X] = \sum_{s \in S} E[X_s] \leq |S| \frac{1}{n} \leq 1$$

La cantidad de elementos esperados que colisionan con u de S

es constante (no importa el tamaño de n) y es como mucho 1.

Diseño de una C. de F. U. de H.

Usaremos

un número p primo $\approx n$ como el tamaño de la tabla de Hash H .

Representaremos a cada elemento $u \in U$

como un vector $X = (x_1, x_2, \dots, x_r)$ con un r fijo, $0 \leq x_i < p$ para cada i

Sea A el set de todos los vectores de la forma:

$a = (a_1, a_2, \dots, a_r)$ con $0 \leq a_i < p$ para cada i

Definimos la función lineal
$$h_a(x) = \left(\sum_{i=1}^r a_i \cdot x_i \right) \bmod p$$

Diseño de una C. de F. U. de H. (cont.)

Definiremos la familia de funciones de hashing

$$H = \{ h_a / a \in A \}$$

Para construir el diccionario

Se selecciona un número primo $p \geq n$

Se genera aleatoriamente uniforme un vector $a \in A$

Con esta se define h_a

Para almacenar la función de hashing

Solo hace falta almacenar el vector a seleccionado \leftarrow es compacto

... solo falta verificar su probabilidad de generar colisiones

Una propiedad previa necesaria...

Sea

p primo

$z \not\equiv 0 \pmod{p}$ (z no es divisible por p).

Entonces

$a \cdot z \equiv m \pmod{p}$

Tiene como mucho una solución con $0 \leq a < p$

Es nuestra propuesta una C.F.H.U?

Sean $x=(x_1, x_2, \dots, x_r)$, $y=(y_1, y_2, \dots, y_r) \in U$

Como $x \neq y$,

entonces tiene que existir al menos un j tal que $x_j \neq y_j$

Vamos a elegir el vector random a

Elegimos al azar todos los $a_i / i \neq j$

El elegir a_j tal que $h_a(x) = h_a(y) \leftarrow$ forzamos a que sean sinónimos

Es nuestra propuesta una C.F.H.U? (cont.)

Como queremos

$$h_a(x) = h_a(y) \rightarrow h_a(x) - h_a(y) = 0$$

Entonces:

$$a_j \cdot \underbrace{(x_j - y_j)}_z = \sum_{i \neq j} \underbrace{a_i (x_i - y_i)}_m \mod p$$

No divisible por p

$$a_j \cdot z = m \mod p$$

Valor fijo ya que todos estos a_i están fijados

Por la propiedad

Solo existe un valor a_j que satisface esta igualdad ($0 \leq a_j < p$).

Entonces solo existe una probabilidad de $1/p$ de elegirlo.

El resto de los valores no influyen en esta selección. Por lo tanto la probabilidad global es $1/p$ (por lo tanto es una clase universal de funciones de hashing)



Presentación realizada en Julio de 2020

Randomizado: Puntos más cercanos en el plano

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski

✉ vpodberezski@fi.uba.ar

Problema

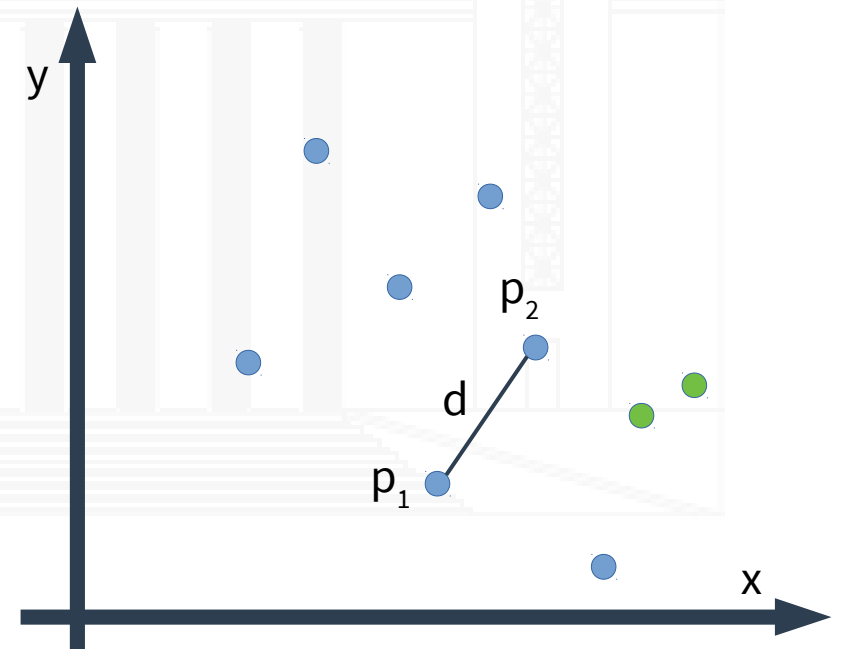
Sea

P un conjunto de “ n ” puntos en el plano

$d(p_1, p_2)$ la función distancia entre $p_1=(x_1, y_1)$, $p_2=(x_2, y_2) \in P$

Queremos

Encontrar los puntos más cercanos en P



Un problema conocido

Ya planeamos una solución al problema

Utilizando división y conquista en $O(n \log n)$

¿Podemos hacerlo mejor?

Plantearemos una manera utilizando randomización

Esperaremos que funcione en $O(n)$

Utilizaremos el método propuesto en 1995 por

M. Golin, R. Raman, C. Schwarz y M. Smid

“Simple randomized algorithms for closest pair problems”

<https://people.scs.carleton.ca/~michielsimplerando.pdf>

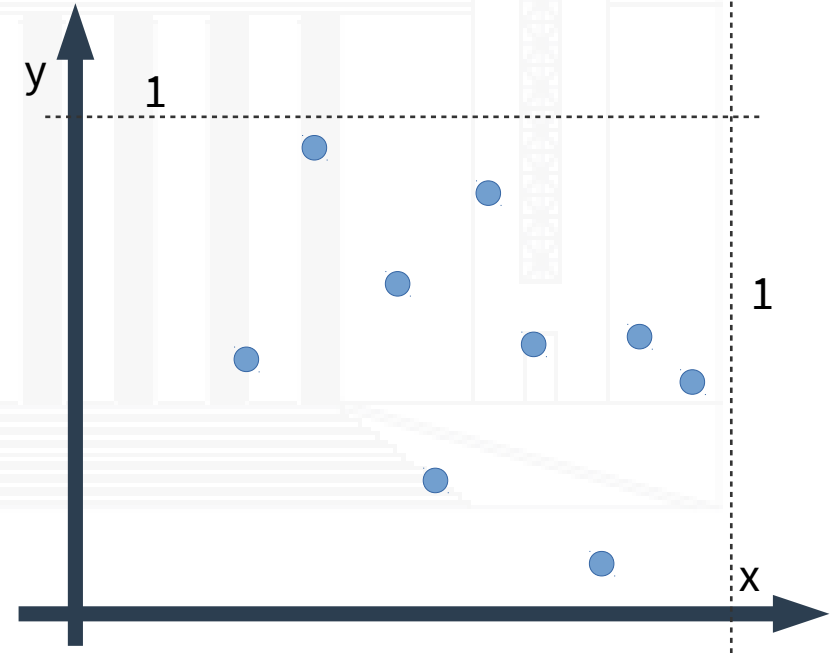
Consideraciones

Utilizaremos como supuesto

Que cada punto $i=(x_i, y_i)$ se encuentra entre las coordenadas $0 < x_i, y_i \leq 1$

Si no

Se pueden escalar en tiempo lineal



Un proceso gradual

Comenzaremos unicamente con 2 puntos: p_1 y p_2

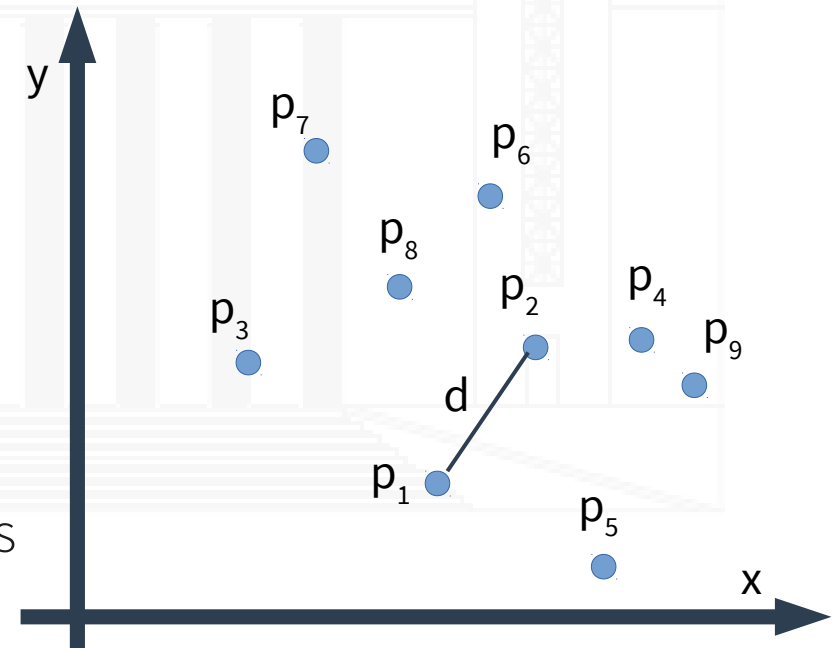
Consideraremos que esos son los más cercanos $\rightarrow \delta = d(p_1, p_2)$

Gradualmente iremos agregando de a 1 punto (p_3, p_4, \dots)

Y verificaremos si el nuevo punto es más cercano a alguno de los anteriores analizados que el par actual

¿Cómo evitamos

tener que comparar un nuevo punto con todos los anteriores?



Regionalización del área

Podemos construir una regionalización

del área de los puntos

Utilizaremos como medida

La mitad de la distancia menor actual

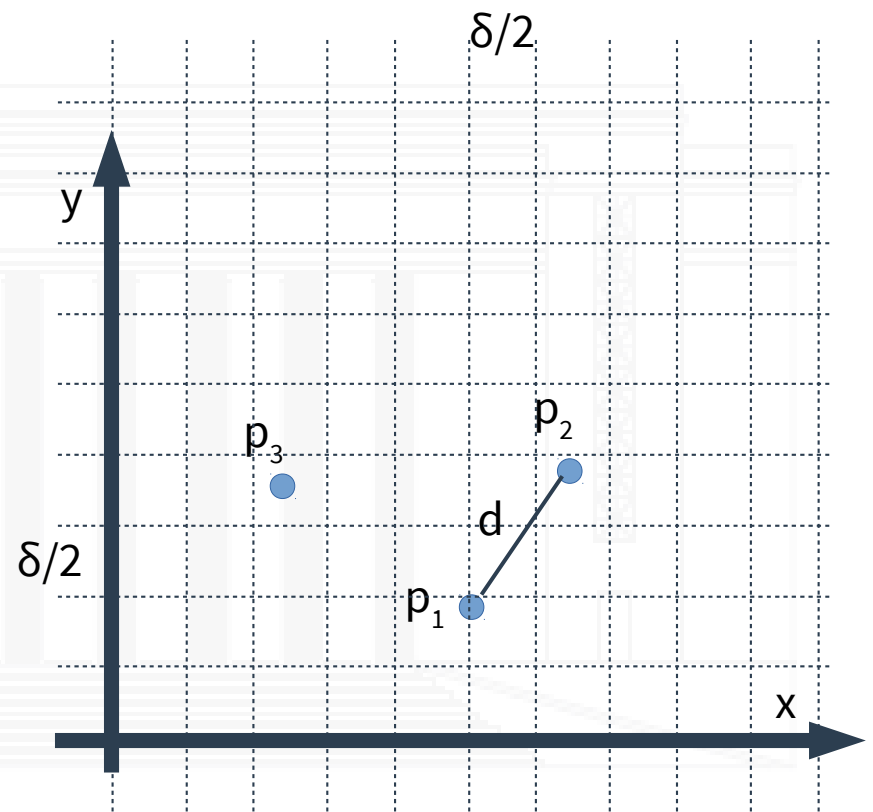
Para un nuevo punto ver

Si se encuentra en la misma celda

O en alguna de sus circundantes

En ese caso

Puede ser un nuevo punto mas cercano con alguno de los puntos preexistentes



Regionalización del área (cont.)

Verificar si están en la misma o en alguna de las 24 celdas circundantes

Y su distancia es menos al actual y al resto de los candidatos

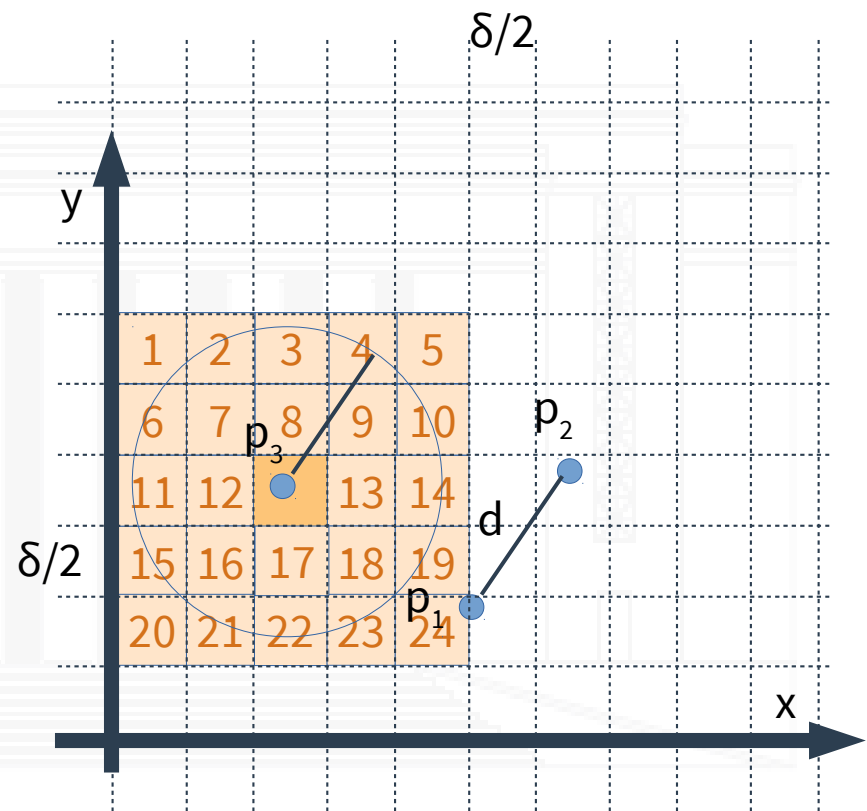
(pueden existir varios puntos en esa situación)

En ese caso también reemplazan a los más cercanos

Si no

Sigue siendo el par anterior el más cercano

Lo agregamos a la grilla y probamos con el siguiente



Pseudocódigo (parcial)

Definir la menor distancia $\delta = d(p_1, p_2)$

Crear grilla G con tamaño $\delta/2$

Insertar p_1 y p_2 celda correspondiente de G

Desde $i=3$ a n

Sea c celda correspondiente a p_i

Sean R los puntos en las 25 celdas cercanas a c en G

```
// pueden ser de 0 o 25
```

Calcular δ_r para $r_x \in R / \min\{d(p_i, r_x)\}$

$$\text{Si } \delta_r < \delta$$
$$\delta = \delta_r$$

...

Sino

Agregar p_i a grilla en celda correspondiente en G

Retornar δ

Análisis (parcial)

El proceso exterior se realiza $O(n)$

1 por cada punto

Analizar si el mínimo actual sigue siéndolo

Requiere realizar una inspección de 25 celdas $\rightarrow O(1)$? (usamos una matriz??)

Y calcular un máximo de $k=25$ distancias $\rightarrow O(1)$

Si el punto analizado no conforma el mínimo

se agrega a la grilla $\rightarrow O(1)$?

Si el punto es el nuevo mínimo

...?

Un nuevo mínimo

Si un punto p_i recién insertado conforma con uno previo

El nuevo mínimo $\delta = d(p_i, p_r)$ se debe tomar a consideración ($r < i$)

Crear una nueva grilla con celdas de $(\delta / 2) \times (\delta / 2)$

Reinsertar los $i-1$ puntos previos y el recién evaluados

¿Cómo podemos caracterizar este proceso?

reinsertar cada punto es $O(1)$

Insertar el punto i requiere i operación $O(1)$

¿Cuántas veces se ejecuta el reinsertar?

Pseudocodigo (actualizado)

```
Definir la menor distancia  $\delta = d(p_1, p_2)$ 
Crear grilla G con tamaño  $\delta/2$ 
Insertar p1 y p2 celda correspondiente de G

Desde i=3 a n
  Sea c celda correspondiente a  $p_i$ 
  Sean R los puntos en las 25 celdas cercanas a c en G
  // pueden ser de 0 o 25
  Calcular  $\delta_r$  para  $r_x \in R / \min\{d(p_i, r_x)\}$ 
  Si  $\delta_r < \delta$ 
     $\delta = \delta_r$ 
    Crear grilla G con tamaño  $\delta/2$ 
    Desde j=1 a i
      Insertar  $p_j$  en la grilla
  Sino
    Agregar  $p_i$  a grilla en celda correspondiente en G
Retornar  $\delta$ 
```

Análisis: El mejor caso

Supongamos

Que la distancia mínima corresponde a los puntos p_1 y p_2

En ese caso

El reinsertar no se llama ninguna vez!

El proceso

Se ejecuta 1 vez para cada punto

Para cada punto se verifican a lo sumo 25 distancias

Globalmente

Tiene una complejidad de $O(n)$

Análisis: El peor caso

Supongamos

Que cada punto ingresado corresponde a un cambio del mínimo

En ese caso

El reinsertar se llama n veces!

El proceso

Se ejecuta 1 vez para cada punto

Para cada punto se verifican a lo sumo 25 distancias

Se regenera la grilla

Se reinsertan los i puntos previos por cada punto

Globalmente

Tiene una complejidad de $O(n^2)$

Otro caso...

En los casos anteriores

Asumimos un determinado orden en el procesamiento de los puntos

Qué pasa si suponemos

que el orden de los puntos es aleatorio?

Será similar a alguno de los casos anteriores? O algo intermedio?

Analizaremos probabilísticamente esta situación

Estimación

Llamaremos

X a la variable aleatoria que especifica la cantidad total de operaciones de inserción en la grilla realizadas

X_i a la variable aleatoria igual a 1 si el i -ésimo punto en el orden aleatorio causa que la distancia mínima cambie. Sino toma el valor de 0

Podemos representar

$$X = n + \sum_{i=1}^n i X_i$$

Si el i -ésimo punto es 1 \rightarrow tengo que reinsertar los i puntos en la nueva grilla

Todos los puntos los inserto al menos 1 vez

Probabilidad de cambio de mínimo

Sean

$P^* = p_1, p_2, \dots, p_i$ los primeros i puntos

$r, s \in P^*$ los puntos de menor distancia

Solo hay cambio de mínimo

si $p_i = r$ o $p_i = s$

Dado que los puntos están en orden aleatorio

La probabilidad de que r (s similarmente) sea el ultimo punto es $1/i$

Por lo tanto

La probabilidad de cambio es $P[X_i=1] = 1/i + 1/i = 2/i$

Esto corresponde a una cota superior

Dado que pueden existir otros pares de puntos con igual distancia a r y s en cuyo caso no se haría el cambio de mínimo

$$P[X_i=1] \leq 2/i$$

Valor esperado de inserciones

Con

la probabilidad calculada $P[X_i=1] \leq 2/i$

La variable aleatoria $X = n + \sum_{i=1}^n i X_i$

Podemos calcular

El valor esperado $E[X] = n + \sum_{i=1}^n i E[X_i] \leq n + \sum_{i=1}^n i * 2/i \rightarrow E[X] \leq n + 2n = 3n$

En conclusión:

Si el orden de los puntos es aleatorio ESPERAMOS $O(n)$ operaciones de inserciones.

Por lo tanto

El proceso global es $O(n)$

... pero debemos asegurarnos QUE LOS PUNTOS ESTÉN EN ORDEN ALEATORIOS (con alta probabilidad!)

Pseudocodigo (actualizado)

Mezclar aleatoriamente los puntos

Definir la menor distancia $\delta = d(p_1, p_2)$

Crear grilla G con tamaño $\delta/2$

Insertar p_1 y p_2 celda correspondiente de G

Desde $i=3$ a n

Sea c celda correspondiente a p_i

Sean R los puntos en las 25 celdas cercanas a c en G

// pueden ser de 0 o 25

Calcular δ_r para $r_x \in R$ / $\min\{d(p_i, r_x)\}$

Si $\delta_r < \delta$

$\delta = \delta_r$

Crear grilla G con tamaño $\delta/2$

Desde $j=1$ a i

Insertar p_j en la grilla

Sino

Agregar p_i a grilla en celda correspondiente en G

Retornar δ

Un problema de tamaño...

Al realizar el análisis de complejidad espacial

Se revela un problema

La cantidad celdas de la grilla

Puede crecer velozmente

La cantidad

no depende de la cantidad de puntos

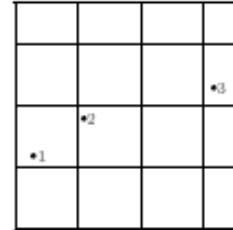
Depende de la distancia mínima encontrada

La utilización de una matriz

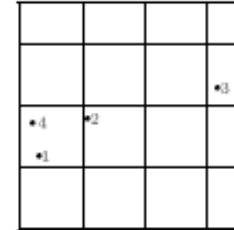
puede volverse inmanejable

Se debe encontrar una alternativa

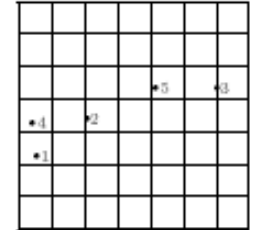
$$\delta(S_2) = d(p_1, p_2)$$



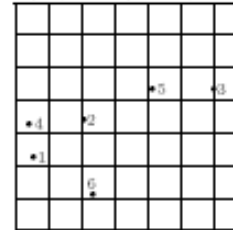
$$\delta(S_3) = d(p_1, p_2)$$



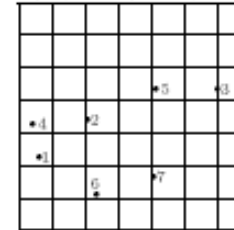
$$\delta(S_4) = d(p_4, p_1)$$



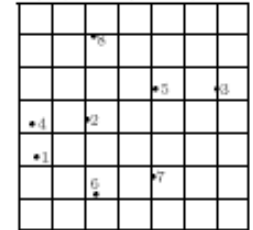
$$\delta(S_5) = d(p_4, p_1)$$



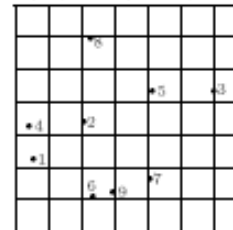
$$\delta(S_6) = d(p_4, p_1)$$



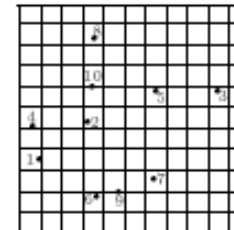
$$\delta(S_7) = d(p_4, p_1)$$



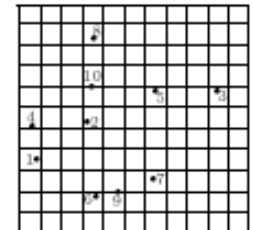
$$\delta(S_8) = d(p_4, p_1)$$



$$\delta(S_9) = d(p_9, p_6)$$



$$\delta(S_{10}) = d(p_9, p_6)$$



Alternativas para la grilla

Existen varias alternativas.

Los autores de la solución proponen:

Arboles de búsquedas balanceados

Hashing perfecto dinámico

La utilización de arboles de búsqueda balanceados

Permiten buscar el punto mas cercano en la etapa i en $O(\log(i))$

Llevando por lo tanto la complejidad temporal esperada del proceso a $O(n \log n)$

Diccionarios

El uso de un diccionario

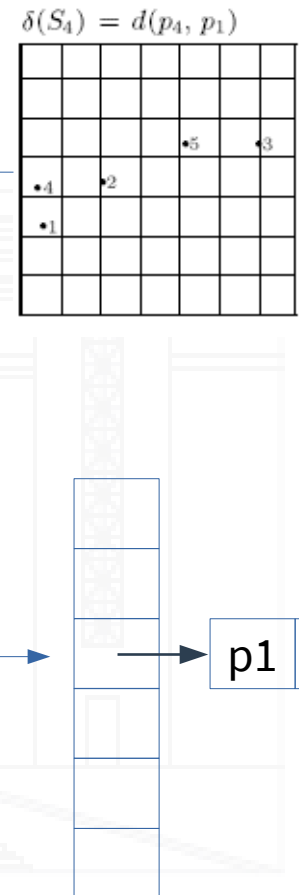
Parece ideal para este problema

Tenemos un universo grande de elementos (celdas)

(podemos nomenciar cada uno de las celdas de la grilla)

Y un subconjunto acotado de elementos a tratar

(a lo sumo 1 celda por punto)



Hashing perfecto dinámico

Corresponde a una estructura de datos

Que “espera” restringir las colisiones

Utiliza Clases universal de funciones de Hashing

Requiere conocer el posible universo de items a insertar

(lo sabemos! escalamos los puntos y conocemos por δ la cantidad de celdas)

Permite

Crear el diccionario con n puntos (ocupando celdas) en tiempo esperado $O(n)$

Buscar un punto en $O(1)$

insertar un punto en una celda en esperado $O(1)$

Pseudocodigo (final)

```
Mezclar aleatoriamente los puntos
Definir la menor distancia  $\delta = d(p_1, p_2)$ 
Crear diccionario G con celdas de tamaño  $\delta/2$ 
Insertar  $p_1$  y  $p_2$  en el diccionario

Desde  $i=3$  a  $n$ 
  Sea  $c$  celda correspondiente a  $p_i$ 
  Sean  $R$  los puntos en las 25 celdas cercanas a  $c$  en G // pueden ser de 0 o 25
  Calcular  $\delta_r$  para  $r_x \in R / \min\{d(p_i, r_x)\}$ 
  Si  $\delta_r < \delta$ 
     $\delta = \delta_r$ 
    Crear diccionario G con celdas de tamaño  $\delta/2$ 
    Desde  $j=1$  a  $i$ 
      Insertar  $p_j$  en el diccionario
  Sino
    Agregar  $p_i$  a grilla en celda correspondiente en G
Retornar  $\delta$ 
```



Presentación realizada en Enero de 2021