CC4102/CC40A/CC53A - Diseno y Analisis de Algoritmos

Jeremy Barbay

23 May 2011

Índice

1. Algoritmos no convencionales (5 semanas = 10 charlas = 900mns)

1.1. Referencias

- Paralelismo:
    - Section 12.3.2 of “Introduction to Algorithms, A Creative Approach”, Udi Manber, p. 382]]

1.2. Descripcion de la Unidad

1.2.1. Resultados de Aprendisajes de la Unidad

- Comprender el concepto de algoritmos
    - aleatorizados,
    - probabilisticos,
    - aproximados
    - paralelos
- y cuando son relevantes
- ser capaz de disenar y analizar algoritmos de estos tipos
- Conocer algunos casos de estudio relevantes

1.2.2. Principales casos de estudio:

- primalidad
- Karp Rabin para busqueda en strings
- numero mayoritario
- arboles binarios de busqueda aleatorizados
- quicksort
- hashing universal y perfecto
- aproximaciones para recubrimiento de vertices

- vendedor viajero
- mochila
- ordenamiento paralelo
- paralel prefix

1.3. Aleatorizacion (1 semana = 2 charlas)

* REFERENCIA:

- Capitulo 1 en “Randomized Algorithms”, de Rajeev Motwani and Prabhakar Raghavan.

1.3.1. Definiciones

- **Algoritmos deterministico**
  - algoritmo que usa solamente instrucciones **deterministicas**.
  - algoritmo de cual la ejecucion (y, entonces, el rendimiento) depende solamente del input.
- **Algoritmos aleatorizados**
  - algoritmo que usa una instruccion **aleatorizada** (potentialemente muchas veces)
  - una distribucion de probabilidades sobre una familia de algoritmos deterministicos.
- **Analisis probabilistica**
  - analysis del rendimiento de un algoritmo, en promedio sobre
    - el aleatorio de la entrada, o
    - el aleatorio del algoritmo, o
    - los dos.
  - veamos que son nociones equivalentes.
- Formalizacion

Algoritmos clasicos	Non clasicos
Siempre hacen lo mismo	aleatorizados
Nunca se equivocan	Monte Carlo
Siempre terminan	Las Vegas

- Clasificacion de los algoritmos **de decision** aleatorizados
  1. Probabilistico: Monte Carlo
     - $P(error) < \epsilon$
     - Ejemplo:
       - deteccion de cliquas
     - Se puede considerar tambien variantes mas fines:
       - two-sided error (=¿clase de complejidad BPP)
         - ⋄ $P(accept|negative) < \epsilon$
         - ⋄ $P(refuse|negative) > 1 - \epsilon$
         - ⋄ $P(accept|positive) > 1 - \epsilon$
         - ⋄ $P(refuse|positive) < \epsilon$

- One-sided error
  - $P(accept|negative) < \epsilon$
  - $P(refuse|negative) > 1 - \epsilon$
  - $P(accept|positive) = 1$
  - $P(refuse|positive) = 0$

2. Probabilistico: Las Vegas
   - Tiempo es una variable aleatoria
   - Ejemplos:
     - determinar primalidad [Miller Robin]
     - busqueda en arreglo desordenado
     - interseccion de arreglos ordenados
     - etc...

- Relacion
  - Si se puede verificar el resultado en tiempo razonable, Monte Carlos iterado hasta correcto, genera Las Vegas

1.3.2. El poder de un algoritmo aleatorizado: Ejemplos

Nota: Trae los juguetes/cajas de colores, con un tesoro a esconder a dentro. Ejemplos de algoritmos o estructuras de datos aleatorizados

1. **hidden coin**
   - Decidir si un elemento pertenece en un lista desordenada de tamano k, o si hay una moneda a dentro de una de las k caja.
   - cual son las complejidades determinisitica y aleatorizada del problema de encontrar **una** moneda, con c la cantidad de monedas,
     - si $c = 1$?
     - si $c = n - 1$?
     - si $c = n/2$?
2. Respuestas:
   - Si una sola instancia de la valor buscada
     - k en el peor caso deterministico
     - $k/2$ en (promedio y en) el peor caso aleatorio
       - con una direccion al azar
       - con $\lg(k!)$ bits aleatorios
   - Si r instancias de la valor buscada
     - $k - r$ en el peor caso deterministico
     - $O(k/r)$ en (promedio y en) el peor caso aleatorio
3. Decidir si un elemento pertenece en una lista ordenadas de tamano n
   - $\Theta(\lg n)$ comparaciones en ambos casos, deterministico y probabilistico.
4. **problema de union**
   - Decidir si un elemento pertenece en una de las k listas ordenadas de tamano n

- Si una sola lista contiene la valor buscada
  - k busquedas en el peor caso deterministico, que da $k\lg(n)$ comparaciones
  - $k/2$ busquedas en (promedio y en) el peor caso aleatorio, que da $k\lg(n)/2$ comparaciones
- Si $r<k$ listas contienen la valor buscada
  - $k-r$ busquedas en el peor caso deterministico, que dan $(k-r)\lg(n)$ comparaciones
  - k/r busquedas en (promedio y en) el peor caso aleatorio, que dan $(k/r)\lg(n)$ comparaciones
- Si $r=k$ listas contienen la valor buscada
  - k busquedas en el peor caso deterministico, en promedio y en el peor caso aleatorio, que dan $k\lg n$ comparaciones

5. **problema de interseccion**
   - dado k arreglos ordenados de tamano n cada uno, y un elemento x.
   - cual son las complejidades determinisitica y aleatorizada del problema de encontrar **un** arreglo que no contiene x (i.e. mostrar que la interseccion de $\{x\}$ con $\cap A$ es vacilla)?
     - si $c=1$ arreglo contiene x?
     - si $c=n-1$ arreglos contienen x?
     - si $c=n/2$ arreglos contienen x?
6. Eso se puede aplicar a la interseccion de posting lists (Google Queries).

1.3.3. Aleatorizacion de la entrada

* Independencia de la distribucion de la entrada

- Si el input sigue una distribucion non-conocida, el input perturbado tiene una distribucion conocida (para una perturbacion bien elegida)
- Ejemplo:
  - flip b de una bit con probabilidad p que puede ser distinta de $1/2$.
  - suma-lo modulo 1 con un otro bit aleatorizado, con probabilidad $1/2$ de ser uno.
  - la suma es igual a uno con probabilidad $1/2$.

* Estructuras de datos aleatorizadas

- **funciones de hash**: estructura de datos aleatorizada, donde (a,b) son elegidos al azar.
- **skiplists**: estructura de datos aleatorizada, que simula en promedio un arbol binario

SkipLists (Ya vista en CC3001!)

1. Estructuras de datos para diccionarios
   - Arreglo ordenado
   - "Move To Front" list (did they see it already?)
   - Arboles binarios
   - Arboles binarios aleatorizados
   - Arboles 2-3 (they saw it already?)
   - Red-Black Trees (they saw it already?)
   - AVL

- Skip List
- Splay trees

2. Skip Lists

- Motivacion
  - un arbol binario con entradas aleatorizadas tienen una altura $O(\lg n)$, pero eso supone un orden de input aleatorizados.
  - El objetivo de las “skip lists” es de poner el aleatorio a dentro de la estructura.
  - tambien, es el equivalente de una busqueda binaria en listas via un resumen de resumen de resumen. . .
- Definicion
  - una skip-list de altura h para un diccionario D de n elementos es una familia de lists $S_0, \dots, S_h$ y un puntero al primero elemento de S_h, tal que
    - S_0 contiene D;
    - cada S_i contiene un subconjunto aleatorio de S_{i-1} (en promedio la mitad)
    - se puede navegar de la izquierda a la derecha, y de la cima hasta abajo.
  - se puede ver como n torres de altura aleatorizadas, conectadas horizontalmente con punteros de la izquierda a la derecha.
  - la informacion del diccionario estan solamente en S_0 (no se duplica)
- Ejemplo

4	X	-	-	-	-	-	-	¿	X
3	X	-	¿	X	-	-	-	¿	X
2	X	-	¿	X	X	¿	X	¿	X
1	X	X	¿	X	X	¿	X	¿	X
0	X	X	X	X	X	X	X	X	X
	$-\infty$	10	20	30	40	50	60	70	∞

- Operaciones
  * Search(x):
  - Start a the first element of S_h
  - while not in S_0
    - go down one level
    - go right till finding a key larger than x
- Search(x)
- create a tower of random height ($p = 1/2$ to increase height, typically)
- insert it in all the lists it cuts.
  * Delete(x)
- Search(x)
- remove tower from all lists.
- Ejemplos
  * Insert(55) con secuencia aleatora (1,0)

4	X	-	-	-	-	-	-	-	¿	X
3	X	-	¿	X	-	-	-	-	¿	X
2	X	-	¿	X	X	-	¿	X	¿	X
1	X	X	¿	X	X	**¿**	**X**	X	¿	X
0	X	X	X	X	X	X	**X**	X	X	X
	$-\infty$	10	20	30	40	50	**55**	60	70	∞

* Insert(25) con (1,1,1,1,0)

5	**X**	-	-	-	-	-	-	-	-	**¿**	**X**
4	X	-	**¿**	**X**	-	-	-	-	-	¿	X
3	X	-	**¿**	**X**	X	-	-	-	-	¿	X
2	X	-	**¿**	**X**	X	X	-	¿	X	¿	X
1	X	X	**¿**	**X**	X	X	¿	X	X	¿	X
0	X	X	X	**X**	X	X	X	X	X	X	X
	$-\infty$	10	20	**25**	30	40	50	55	60	70	∞

* Delete(60)

5	X	-	-	-	-	-	-	-	¿	X
4	X	-	¿	X	-	-	-	-	¿	X
3	X	-	¿	X	X	-	-	-	¿	X
2	X	-	¿	X	X	X	-	-	**¿**	X
1	X	X	¿	X	X	X	¿	X	¿	X
0	X	X	X	X	X	X	X	X	X	X
	$-\infty$	10	20	25	30	40	50	55	70	∞

- Analisis
  * Espacio: Cuanto nodos en promedio?
  - cuanto nodos en lista S$_i$?
    - $n/2^i$ en promedio
  - Summa sobre todos los niveles
    - $n\sum 1/2^i < 2n$
- altura promedio es $O(\lg n)$
- tiempo promedio es $O(\lg n)$

Paginamiento al Azar :OPTIONAL:

- REFERENCIA:
  - Capitulo 3 en "Online Computation and Competitive Analysis", de Allan Borodin y Ran El-Yaniv
  - Capitulo 13 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan, p. 368

Tipos de Adversarios (cf p372 [Motwani Raghavan]) Veamos en el caso deterministico un tipo de adversario offline, como medida de dificultad para las instancias online. Para el problema de paginamiento con k paginas, el ratio optima entre un algoritmo online y offline es de k (e.g. entre LRU y LFD).

- DEFINICION:

  En el caso aleatorizado, se puede considerar mas tipos de adversarios, cada uno definiendo una medida de dificultad y un modelo de complejidad.

  1. Adversario "Oblivious" ("Oblivious Adversary")
     El adversario conoce A pero no R: el elija su instancia completamente al inicial, antes de la ejecucion online del algoritmo.
  2. Adversario Offline adaptativo
     Para este definicion, es mas facil de pensar a un adversario como un agente distinto del algoritmo offline con quien se compara el algoritmo online.

     El adversario conoce A en total, pero R online, y le utiliza para generar una instancia peor I. Este instancia I es utilizada de nuevo para ejecutar el algoritmo offline (quien conoce el futuro) y producir las complejidades (a cada instante online) con cual comparar la complejidad del algoritmo online.

3. Adversario Online adaptativo
   En este definicion, el algoritmo conoce A en total, construir la instancia I online como en el caso precedente, pero tiene de tiene de resolverla online tambien (de una manera, no se ve en el futuro).

Comparacion de los Tipos de adversarios.

- Por las definiciones, es claro que
  - el adversario offline adaptativo
    - es mas poderoso que
  - el adversario online adaptativo
    - es mas poderoso que
  - el adversario oblivious

Competitiva Ratios * Para un algoritmo online A, para cada tipo de adversario se define un ratio de competitividad:

- C_A^{obl}: competitivo ratio con adversario oblivious
- C_A^{aon}: competitivo ratio con adversario adaptativo online
- C_A^{aof}: competitivo ratio con adversario adaptativo offline

Es obvio que, para A fijada, considerando un adversario mas poderoso va aumentar el ratio competitivo:

$$C_A^{obl} \leq C_A^{aon} \leq C_A^{aof}.$$

* Para un problema el ratio de competitividad de un problema es el ratio de competitividad minima sobre todos los algoritmos correctos para este problema:

$$C^{obl} \leq C^{aon} \leq C^{aof} \leq C^{det}$$

donde C^{det} es el competitivo ratio de un algoritmo online deterministico.

Arboles Binarios de Busqueda aleatorizados

- Conrado Martinez. Randomized binary search trees. Algorithms Seminar. Universitat Politecnica de Catalunya, Spain, 1996.
- Conrado Martinez and Salvador Roura. Randomized binary search trees. J. ACM, 45(2):288–323, 1998.
- http://en.wikipedia.org/wiki/Treaphttp://en.wikipedia.org/wiki/Treap, seccion "Randomized$_{\text{binary search tree}}$".

1.3.4. Complejidad Probabilistica: cotas inferiores

* REFERENCIA:

1. Capitulo 1 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan.
2. Problema
   - Strategia de adversario no funciona
   - En algunos casos, teoria de codigos es suficiente (e.g. busqueda en arreglo ordenado). Eso es una cota inferior sobre el tamano del certificado.
   - En otros caso, teoria de codigos no es suficiente. En particular, cuando el precio para verificar un certificado es mas pequeno que de encontrarlo. En estos casos, utilizamos otras tecnicas:

- teoria de juegos (que vamos a ver) y equilibro de Nash
- cotas sobre la comunicacion en un sistema de "Interactive Proof"

3. Algunas Notaciones Algebraicas

   Sea:

   - A una familia de n_a algoritmos deterministicos
   - a un vector $(0, ..., 0, 1, 0, ..., 0)$ de dimension n_a
   - α una distribucion de probabilidad de dimension n_a
   - B una familia de n_n instancias
   - b un vector $(0, ..., 0, 1, 0, ..., 0)$ de dimension n_b
   - β una distribucion de probabilidad de dimension n_b
   - M una matriz de dimension $n_a \times n_b$ tal que $\mathrm{M_{a,b}}$ es el costo del algoritmo a sobre la instancia b. Por definicion,
     - $a^t M b = M_{a,b}$
     - $\alpha^t M b$ es la complejidad en promedio (sobre el aleatorio del algoritmo α) de α sobre b
     - $a^t M \beta$ es la complejidad en promedio (sobre la distribucion de instancias β) de a sobre β
     - α^t M β es la complejidad en promedio del algoritmo aleatorizados α sobre la distribucion de instancia β.

4. von Neuman's theorem: infsup = supinf = minmax = maxmin

   a) OPCIONAL Existencia de $\tilde{\alpha}$ et $\tilde{\beta}$

   Dado ϕ y ψ definidas sobre $\mathcal{R}^m$ y $\mathcal{R}^n$ por

   $$\phi(\alpha) = \sup_\beta \alpha^T M \beta \text{ y } \psi(\beta) = \inf_\alpha \alpha^T M \beta$$

   Entonces:

   - $\phi(\alpha) = \max_\beta \alpha^T M \beta$
   - $\psi(\beta) = \min_\alpha \alpha^T M \beta$
   - hay estrategias mixtas
     - $\tilde{\alpha}$ por A
     - $\tilde{\beta}$ por B
   - tal que
     - ϕ es a su minima en $\tilde{\alpha}$ y
     - ψ es a su maxima en $\tilde{\beta}$.

   b) Resultado de von Neuman:

   Dado un juego Γ definido por la matrica \$M\$˜:

   $$\min_\alpha \max_\beta \alpha^T M \beta = \max_\beta \min_\alpha \alpha^T M \beta$$

   c) Interpretacion:

   - Este resultado significa que si consideramos ambos distribuciones sobre algoritmos y instancias, no importa el orden del max o min:
     - podemos elegir el mejor algoritmo (i.e. minimizar sobre los algoritmos aleatorizados) y despues elegir la peor distribucion de instancias para el (i.e. maximizar sobre las distribuciones de instancias), o al reves

- podemos elegir la peor distribucion de instancias (i.e. maximizar sobre las distribuciones de instancias), y considerar el mejor algoritmo (i.e. minimizar sobre los algoritmos aleatorizados) para este distribucion.

- ATENCION!!!! Veamos que
  - El promedio (sobre las instancias) de las complejidades (de los algoritmos) en el peor caso
  - no es igual
  - al peor caso (sobre las instancias) de la complejidad en promedio (sobre el aleatorio del algoritmo)
  - donde el segundo termo es realmente la complejidad de un algoritmo aleatorizados.
- Todavia falta la relacion con la complejidad en el peor caso b de un algoritmo aleatorizados α:

$$\max_b \min_\alpha \alpha^T M b$$

5. Lema de Loomis

   * Dado una estrategia aleatoria α, emite una instancia b tal que α es tan mal en b que en el peor β.

$$\forall \alpha \exists b, \max_\beta \alpha^T M \beta = \alpha^T M b$$

   * Dado una distribucion de instancias β, existe un algoritmo deterministico a tal que a es tan bien que el mejor algoritmo aleatorizados α sobre la distribucion de instancias β:

$$\forall \beta \exists a, \min_\alpha \alpha^T M \beta = a^T M \beta$$

   * Interpretacion:

   - En frente a una distribucion de instancias especifica, siempre existe un algoritmo deterministico optima en comparacion con los algoritmos aleatorizados (que incluen los deterministicos).
   - En frente a un algoritmo aleatorizados, siempre existe una instan ca tan mal que la pero distribucion de instancias.

6. Principe de Yao

   * Del leima de Loomis podemos concluir que

$$\max_\beta \alpha^T M \beta = \max_b \alpha^T M b$$

$$\min_\alpha \alpha^T M \beta = \min_a a^T M \beta$$

   * Del resultado de von Neuman sabemos que maxmin=minmax (sobre α y β):

$$\min_\alpha \max_\beta \alpha^T M \beta = \max_\beta \min_\alpha \alpha^T M \beta$$

   * Entonces

$$\min_a lpha \max_b \alpha^T M b = (Loomis) \min_\alpha \max_\beta \alpha^T M \beta = (vonNeuman) \max_\beta \min_\alpha \alpha^T M \beta = (Loomis) \max_\beta \min_a a^T M \beta$$

   * Interpretacion

mín$_\alpha$ máx$_b$ $\alpha^T Mb$
La complejidad
del mejor algoritmo aleatorizado
en el peor caso

es igual a

máx$_\beta$ mín$_a$ $\alpha^T Mb$
La complejidad
del mejor algoritmo deterministico
sobre la peor distribucion de instancias

"El peor caso del mejor algoritmo aleatorizado corresponde a la peor distribucion para el mejor algoritmo deterministico."

* Ejemplos de cotas inferiores:

1. Decidir si un elemento pertenece en una lista ordenadas de tamano n

   - Cual es el peor caso b de un algoritmo aleatorizado α?
   - Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (del modelo de comparaciones)
   - Consideramos la distribucion uniforma.
   - Cada algoritmo deterministico se puede representar como un arbol de decision (binario) con $2n+1$ hojas.
   - Ya utilizamos para la cota inferior deterministica que la altura de un tal arbol es al menos $\lg(2n+1) \in \Omega(\lg n)$. Esta propiedad se muestra por recurrencia.
   - De manera similar, se puede mostrar por recurrencia que la altura en promedio de un tal arbol binario es al menos $\lg(2n+1) \in \Omega(\lg n)$.
   - Entonces, la complejidad promedio de cada algoritmo deterministico a sobre β_0 es al menos $\lg(2n+1) \in \Omega(\lg n)$.
   - Entonces, utilizando el principie de Yao, la complejidad en el peor caso de un algoritmo aleatorizado en el modelo de comparaciones es al menos $\lg(2n+1) \in \Omega(\lg n)$.
   - El corolario interesante, es que el algoritmo deterministico de busqueda binaria es **oprima** a dentro de la clase mas general de algoritmos aleatorizados.

2. Decidir si un elemento pertenece en un lista desordenada de tamano k * Si una sola instancia de la valor buscada ($r = 1$) * Cotas superiores

   - k en el peor caso deterministico
   - $(k+1)/2$ en el peor caso aleatorio
     - con una direccion al azar
     - con $\lg(k!)$ bits aleatorios

3. Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (en el modelo de comparaciones).

4. Consideramos la distribucion uniforma (cada algoritmo reordena la instancia a su gusto, de toda manera, entonces solamente la distribucion uniforma tiene sentido): cada posicion es elegida con probabilidad $1/k$

5. Se puede considerar solamente los algoritmos que no consideran mas que una ves cada posicion, y que consideran todas las posiciones en el peor caso: entonces cada algoritmo puede ser representado por una permutacion sobre k.

6. Dado un algoritmo deterministico a, para cada $i \in [1, k]$, hay una instancia sobre cual el performe i comparaciones. Entonces, su complejidad en promedio en este instancia es $\sum_i i/k$, que es $k(k+1)/2k = (k+1)/2$. Como eso es verdad para todos los algoritmos deterministicos, es verdad para el mejor de ellos tambien.

7. Entonces, utilizando el principio de Yao, la complejidad en el peor caso de un algoritmo aleatorizado en el modelo de comparaciones es al menos $(k+1)/2$.

   * Si r instancias de la valor buscada * Cotas superiores

8. $k - r$ en el peor caso deterministico

9. $O(k/r)$ en (promedio y en) el peor caso aleatorio * Cota inferior

10. Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (en el modelo de comparaciones).

11. Consideramos la distribucion uniforma (cada algoritmo reordena la instancia a su gusto, de toda manera, entonces solamente la distribucion uniforma tiene sentido): cada posicion es elegida con probabilidad $1/k$

12. Se puede considerar solamente los algoritmos que no consideran mas que una ves cada posicion.

13. De verdad, no algoritmo tiene de considerar mas posiciones que $k-r+1$, entonces hay menos algoritmos que de permutaciones sobre k elementos. Para simplificar la prueba, podemos exigir que los algoritmos especifican una permutacion entera, pero no vamos a contar las comparaciones despues que un de las r valores fue encontrada.

14. Decidir si un elemento pertenece en k listas ordenadas de tamano n/k * Cotas superiores * Si una sola lista contiene la valor buscada

    - k busquedas en el peor caso deterministico, que da $k \lg(n/k)$ comparaciones
    - $k/2$ busquedas en (promedio y en) el peor caso aleatorio, que da $k \lg(n/k)/2$ comparaciones

15. $k - r$ busquedas en el peor caso deterministico, que dan $(k-r)\lg(n/(k-r))$ comparaciones

16. k/r busquedas en (promedio y en) el peor caso aleatorio, que dan $(k/r)\lg(n/k)$ comparaciones

    * Si $r = k$ listas contienen la valor buscada

17. k busquedas en el peor caso deterministico, en promedio y en el peor caso aleatorio, que dan $k \lg(n/k)$ comparaciones

18. Aplicacion:

    algoritmos de interseccion de listas ordenadas.

* Conclusion:
  * Relacion fuerte entre algoritmos aleatorizados y complejidad en promedio
  * El peor caso de un algoritmo aleatorio corresponde a la peor distribucion para un algoritmo deterministico.
  * Otras aplicaciones importantes de los algoritmos aleatorizados
  * "Online Algorithms", en particular paginamiento.
  * Algoritmos de aproximacion
  * Hashing

1.3.5. Relacion con Problemas NP-Dificiles

* Ejemplos de Problemas NP-Dificiles

1. **Maxcut**
   - dado un grafe $G = (V, E)$
   - encontrar una partition (L, R) tq $L \cup R = V$ y que **maximiza** la cantidad de aristas entre L y R
   - el problema es NP dificil
   - se aproxima con un factor de dos con un algoritmo aleatorizado en tiempo polinomial.
2. **mincut**
   - dado un grafe $G = (V, E)$
   - encontrar una partition (L, R) tq $L \cup R = V$ y que minimiza la cantidad de aristas entre L y R
   - el problema es NP dificil

* Relacion con la nocion de NP:

- El arbol representando la ejecucion de un algoritmo non-deterministico en tiempo polynomial (i.e. NP) se decomposa en dos partes, de altura polynomial $p(n)$:
  - una parte de **decisiones** non-deterministica (fan-out)
  - una parte de **verificacion** deterministica (straight)
- Si una solamente de las $2^{p(n)}$ soluciones corresponde a una solucion valida del problema, la aleatorizacion no ayuda, pero si una proporcion constante (e.g. \$1/2,1/3,1/4,... \$) de las ramas corresponden a una solucion correcta, existe un algoritmo aleatorizado que resuelve el problema NP-dificil en tiempo polynomial **en promedio**.

1.3.6. Complejidad de un algoritmo aleatorizado

* Considera algoritmos con comparaciones

- algoritmos deterministicos se pueden ver como arboles de decision.
- algoritmos aleatorios se pueden ver (de manera intercambiable) como
  - una distribucion sobre los arboles de decision,
  - un arbol de decision con algunos nodos "aleatorios".
- La complejidad en una instancia de un algoritmo aleatorio es el promedio de la complejidad (en este instancia) de los algoritmos deterministicos que le compasan:
  $C((A_r)_r, I) = E_r(C(A_r, I))$
- La complejidad en el peor caso de un algoritmo aleatorio es el peor caso del promedio de la complejidad de los algoritmos deterministicos que le composan:
  $C((A_r)_r) = \text{máx}_I\, C((A_r)_r, I) = \text{máx}_I\, E_r(C(A_r, I))$

1.3.7. Primalidad

* REFERENCIAS:

- Primalidad: Capitulo 14.6 en “Randomized Algorithms”, de Rajeev Motwani and Prabhakar Raghavan.

- http://en.wikipedia.org/wiki/Primality$_{t}estComplexityhttp : //en.wikipedia.org/wiki/Primality_test\#Complexity$
* Algoritmo “Random Walks” para SAT

1. eliga cualquieras valores $x_1, \ldots, x_n$
2. si todas las clausilas son satisfechas,
   - accepta
3. sino
   - eliga una clausula non satisfecha (deterministicamente o no)
   - eliga una de las variables de esta closula.
4. Repite es r veces.

* PRIMES is in coNP

si $x \in coNP$, eliga non-deterministicamente una decomposicion de x y verificalo.

* PRIMES is in NP (hence in NP∩ coNP)

In 1975, Vaughan Pratt showed that there existed a certificate for primality that was checkable in polynomial time, and thus that PRIMES was in NP, and therefore in NP Á coNP.

* PRIMES in coRP

The subsequent discovery of the Solovay-Strassen and Miller-Rabin algorithms put PRIMES in coRP.

* PRIMES in $ZPP = RP \cap coRP$

In 1992, the Adleman-Huang algorithm reduced the complexity to $ZPP = RP \cap coRP$, which superseded Pratt’s result.

* PRIMES in QP

The cyclotomy test of Adleman, Pomerance, and Rumely from 1983 put PRIMES in QP (quasi-polynomial time), which is not known to be comparable with the classes mentioned above.

* PRIMES in P

Because of its tractability in practice, polynomial-time algorithms assuming the Riemann hypothesis, and other similar evidence, it was long suspected but not proven that primality could be solved in polynomial time. The existence of the AKS primality test finally settled this long-standing question and placed PRIMES in P.

* $PRIMES \in NC$? $PRIMES \in L$?

PRIMES is not known to be P-complete, and it is not known whether it lies in classes lying inside P such as NC or L.

1.3.8. Clases de complejidad aleatorizada :BONUS:

RP

- “A **precise** polynomial-time bounded nondeterministic Turing Machine”, aka “maquina de Turing non-deterministica acotadada polinomialemente **precisa**”, es una maquina tal que su ejecucion sobre las entradas de tamano n toman tiempo p(n) **todas**.
- “A **polynomial Monte-Carlo Turing machine**”, aka una “*maquina de Turing de Monte-Carlo* polynomial” para el idioma L, es una tal maquina tal que
  - si $x \in L$, al menos la mitad de los $2^{p(|x|)}$ caminos acceptan x
  - si $x \notin L$, **todas** los caminos rechazan x.

- La definicion corresponde exactamente a la definicion de algoritmos de Monte-Carlo. La classe de idiomas reconocidos por una **maquina de Turing de Monte-Carlo** polynomial es RP.
- $P \subset BP \subset NP$:
  - un algoritmo en P accepta con todos sus caminos cuando una palabra x es en L, que es "al menos" la mitad.
  - un algoritmo en NP accepta con al menos un camino: un algoritmo en RP accepta con al menos la mitad de sus caminos.

ZPP

- $ZPP = RP \cap coRP$
- $Primes \in ZPP$

PP

- Maquina que, si $x \in L$, accepta en la mayoridad de sus entradas.
- PP probablemente no en NP
- PP probablemente no en RP

BPP

$$BPP = \left\{ L, \forall x, \left\{ \begin{array}{l} x \in L \Rightarrow 3/4 \text{ de los caminos acceptan } x \\ x \notin L \Rightarrow 3/4 \text{ de los caminos rechazan } x \end{array} \right. \right\}$$

$RP \subset BPP \subset PP$
$BPP = coBPP$

1.4. Nociones de aproximabilidad (2 semanas = 4 charlas)

- problemas que son o no aproximables

1.4.1. (Motivacion)

Aproximacion de problemas de optimizacion NP-dificiles

- Que problemas NP dificiles conocen?
  - colorisacion de grafos
  - ciclo hamiltonian
  - Recubrimiento de Vertices (Vertex Cover)
  - Bin Packing
  - Problema de la Mochila
  - Vendedor viajero (Traveling Salesman)
- Que hacer cuando se necessita una solucion en tiempo polinomial?
  - Consideramos los problemas NP completos de decision, generalmente de optimizacion. Si se necesita una soluciono en tiempo polinomial, se puede considerar una aproximacion.

1.4.2. $p(n)$-aproximacion

Definicion Dado un problema de minimizacion, un algoritmo A es un

* $p(n)$**-aproximacion** si $\forall n \max \frac{C_A(x)}{C_{OPT}(x)} \leq p(n)$

* $p(n)$**-aproximacion** si $\forall n \max \frac{C_{OPT}(x)}{C_A(x)} \leq p(n)$

* Notas:

1. Aqui consideramos la cualidad de la solucion, NO la complejidad del algoritmo. Usualmente el problema es NP-dificil, y el algoritmo de aproximacion es de complejidad polinomial.
2. Las razones estas ≥ 1 (elijamos las definiciones para eso)
3. A veces consideramos tambien $C_A(x) - C_{OPT}(x)$ (minimizacion) y $C_{OPT}(x) - C_A(x)$: eso se llama un "esquema de aproximacion polinomial" y vamos a verlo mas tarde.

Ejemplo: Bin Packing (un problema que es 2-aproximable)

- DEFINICION

  Dado n valores $x_1, \ldots, x_n$, $0 \leq x_i \leq 1/2$, cual es la menor cantidad de cajas de tamano 1 necesarias para empaquetarlas?

- Algoritmo "Greedy"
  - Considerando los x_i en orden,
  - llenar la caja actual todo lo posible,
  - pasar ala siguiente caja.

- Analisis
  - Este algoritmo tiene complejidad lineal $O(n)$.
  - Greedy da una 2-aproximacion.
  - (se puede mostrar facilamente en las instancias donde el algoritmo optima llene completamente todas las cajas.)

- Ademas, hay mejores aproximaciones,
  1. Best-fit tiene performance ratio de 1.7 en el peor caso
  2. Extract from "Best-Fit Bin-Packing with Random Order (1997), Kenyon"

     Best-fit is the best known algorithm for on-line binpacking, in the sense that no algorithm is known to behave better both in the worst case (when Best-fit has performance ratio 1.7) and in the average uniform case, with items drawn uniformly in the interval [0; 1] (then Best-fit has expected wasted space O(n 1=2 (log n) 3=4)). In practical applications, Best-fit appears to perform within a few percent of optimal. In this paper, in the spirit of previous work in computational geometry, we study the expected performance ratio, taking the worst-case multiset of items L, and assuming that the elements of L are inserted in random order, with all permutations equally likely. We show a lower bound of 1:08 : : : and an upper bound of 1:5 on the random order performance ratio of Best-fit. The upper bound contrasts with the result that in the worst case, any (deterministic or randomized) on-line bin-packing algorithm has performance ratio at least 1:54 : : :. 1
  3. Un otro paper donde particionan los x_i en tres classes, placeando los x_i mas grande primero, buscando el placamiento optimal de los x_i promedio, y usando un algoritmo greedy para los x_i pequenos. [Karpinski?]
  4. la distribucion de los tamanos de las cajas hacen la instancia dificil o facil.

Ejemplo: Recubrimiento de Vertices (Vertex Cover)

- DEFINICION:

  Dado un grafo $G = (V, E)$, cual es el menor $V' \subseteq V$ tal que V' sea un vertex cover de G. (o sea V' mas pequeno tq $\forall e=(u,v)\in E, u \in V' ov \in V') Algoritmodeaproximacion :$

-
  - $V' \leftarrow \emptyset$
  - while $E \neq \emptyset$
    - sea $(u, v) \in E$
    - $V' \leftarrow V' \cup u, v$
    - $E \leftarrow E\ (x, y), x = uox = voy = uoy = v$
  - return V

- Discusion: es una 2-aproximacion o no?
- LEMA: El algoritmo es una 2-aproximacion.
  - PRUEBA:
    - Cada par u, v que la aproximacion elige esta conectada, entonces u o v estas en cualquier soluciono optima de Vertex Cover.
    - Como se eliminan las aristas incidentes en u y v, los siguientes pares que se eligen no tienen interseccion con el actual, entonces cada 2 nodos que el algoritmo elige, uno pertenece a la solucion optima.
    - quod erat demonstrandum, (QED).

Ejemplo: Vendedor viajero (Traveling Salesman)

- DEFINICION:

  Dado $G = (V, E)$ dirigido y $C : E \rightarrow R^+$ una funcion de costos, encontrar un recorrido que toque cada ciudad una vez, y minimice la suma de los costos de las aristas recorridas.

- LEMA: Si c satisface la desigualdad triangular $\forall x, y, zc(x, y)+x(y, z) \geq c(x, z)$, hay una 2-aproximacion.
  - PROOF:
    - Algoritmo de Aproximacion (con desigualdad triangular)
      - construir un arbol cobertor minimo (MST)
      - se puede hacer en tiempo polinomial, con programacion lineal.
      - $C_{MST} \leq C_{OPT}$
      - producimos un recorrido en profundidad DFS del MST:
      - $C_{DFS} = 2C_{MST} \leq 2C_{OPT}$
      - (factor dos porque el camino de vuelta puede ser al maximo de tamano igual al tamano del camino de ida)
      - eliminamos los nodos repetidos del camino, que no crece el costo
      - $C_A \leq 2C_{OPT}$
    - quod erat demonstradndum, QED.

- LEMA: Si c no satisface la desigualdad triangular, el problema de vendedor viajero no es aproximables en tiempo polinomial (a menos que P=NP$).
  - PROOF:
    - Supongamos que existe una p(n)-aproximacion de tiempo polinomial.

- ○ Dado un grafo $G = (V, E)$
- ○ Construimos un grafo $G' = (V, E')$ tal que
  - ◇ $E' = V^2$ (grafo completo), y
  - ◇ $c(u, v) = 1$ si $(u, v) \in E$ $np(n)$ sino
- ○ Si no hay un Ciclo Hamiltonian en E,
  - ◇ todas las soluciones usan al menos una arista que no es en E
  - ◇ entonces todas las soluciones tienen costo mas que $np(n)$
- ○ Si hay un Ciclo Hamiltonian en E
  - ◇ tenemos una aproximacion de una soluciono de costo menos que $(n-1)p(n)$ QED

Ejemplo: Vertex Cover con pesos

- DEFINICION
  Dado $G = (V, E)$ y $c : V^+$, se quiere un $V^* \subseteq V$ que cubra E y que minimice $\sum_{v \in V^*} c(v)$.
- Este problema es NP Completo.
- LEMA: Vertex Cover con pesos es 2-aproximable
- PROOF:

  1. Sea variables $x(v) \in 0, 1$, $\forall v \in V$
     - el costo de V^* sera $\sum x(v)c(v)$
     - objetivo: $\min \sum_{v \in V} x(v)c(v)$ donde $0 \leq x(v) \leq 1 \forall v \in V$
     - $x(u) + x(v) \geq 1 \forall u, v \in E$
  2. $x(v) \in Z$ sere programacion entera, que es NP Completa $x(v) \in R$ sere programacion lineal, que es polinomial el valor $\sum x(v)c(v)$ que produce el programo lineal es inferior a la mejor solucion al problema de VC.
  3. Algoritmo
     - resolver el problema de programacion lineal
       - ○ nos da $x(v_1), \ldots, x(v_n)$
     - $V^* \leftarrow \emptyset$
     - for $v \in V$
       - ○ si $x(v) \geq 1/2$
         - ◇ $V^* \leftarrow V^* \cup v$
     - return V^*

4. Propiedades

    - V^* es un vertex cover:

        - si $\forall (u,v) \in E, x(u) + x(v) \geq 1$
        - entonces, $x(u) \geq 1/2ox(v)/geq1/2$
        - entonces, $u \in V^*ov \in V^*$

    - V^* es una 2-aproximacion:

        - $c(V^*) = \sum_v y(v)c(v)$ donde $y(v) = 1six(v) \geq 1/2$, y 0 sino
        - entonces $y(v) \leq 2x(v)$
        - $\sum y(v)c(v) \leq \sum 2x(v)c(v) \leq 2OPT$
        - $C(V^*) \leq 2OPT$

5. QED

1.4.3. 4.2.2 PTAS y FPTAS

Definiciones * Esquema de aproximacion poliniomial **PTAS**

Un **esquema de aproximacion polinomial** para un problema es un algoritmo A que recibe como input

- una instancia del problema y
- un parametro $\varepsilon > 0$

y produce una $(1 + \varepsilon)$ aproximacion. Para todo ε fijo, el tiempo de A debe ser polinomial en n, el tamaño de la instancia.

* Ejemplos de complejidades: Cuales tienen sentidos?

- $O(n^{2/3})$
- $O(\frac{1}{\varepsilon^2} n^2)$
- $O(2^{\varepsilon} n)$
- $O(2^{1/\varepsilon} n)$

* Esquema de aproximacion completamente polinomial **FPTAS**

Un **esquema de aproximacion completamente polinomial** es un PTAS donde el tiempo del algoritmo es polinomial en n y en $1/\varepsilon$.

Ejemplo: Problema de la Mochila

1. Definicion

    - Dado

        - n pesos $p_1, \ldots, p_n \geq 0$,
        - n valores $v_1, \ldots, v_n \geq 0$,
        - un peso total maximo P

- Queremos encontrar $S \subset [1..n[$ tal que
  - $\sum_{i \in S} p_i \leq P$
  - $\sum_{i \in S} v_i$ sea maximal.

2. Solucion Exacta

- Dado $L = \{y_1, , \ldots y_m\}$,
  - definimos $L + x = \{y_1 + x, \ldots, y_m + x\}$.
- Algoritmo:
  - $L \leftarrow \{\emptyset\}$
  - for $i \leftarrow 1$ to n
    - $L \leftarrow$ merge $(L, L + x_i)$
    - prune(L, P) (remudando las valores $> P$)
  - return máx(L)

3. Solucion aproximada (inspirada del algoritmo exacto)

- Operacion "Recorte"
  - Definimos la operacion de **recorte** de una lista L con parametro δ:
    - Dado $y, z \in L$, z **represente** a y si
      - $y/(1 + \delta) \leq z \leq y$
  - vamos a eliminar de L todos los y que sean representados por alguno z no eliminado de L.
  - Recortar(L, δ)
    - Sea $L = \{y_1, \ldots, y_m\}$
    - $L' \leftarrow \{y_1\}$
    - $Last \leftarrow y_1$
    - for $i \leftarrow 2$ to m
      - si $y_i > Last(1 + \delta)$
      - $L \leftarrow L'.\{y_i\}$
      - $Last \leftarrow y_i$
    - return L'
- Algoritmo de Aproximacion:
  - $L \leftarrow \{\emptyset\}$

- for $i \leftarrow 1$ to n
  - $L \leftarrow merge(L, L + x_i)$
  - $prune(L, t)$ (remudando las valores $> t$)
  - $L \leftarrow recortar(L, \epsilon/2n)$
- return $\max(L)$

▪ Analysis

- El resultado es una $(1+\epsilon)$-aproximacion:

  a) retorne una solucion valida, tal que

  - $\sum(S') \leq t$ para algun $S' \subset S$

  b) en el paso i, para todo $z \in L_{OPT}$, existe un $y \in L_A$ tal que z representa a y.

  - Luego de los n pasos, el z^* optimo en L_{OPT} tiene un representante $y^* \in L_A$ tal que
  $$z^*/(1+\epsilon/2n)^n \leq y^* \leq z^*$$

  c) Para mostrar que el algoritmo es una $(1+\epsilon)$ aproximacion,

  - hay que mostrar que
    - ◇ $z^*/(1+\epsilon) \leq y^*$
  - entonces, debemos mostrar que
    - ◇ $(1+\epsilon/2n)^n \leq 1+\epsilon$
  - Eso se muestra con puro calculo:
    - ◇ $(1+\epsilon/2n)^n \leq ?1+\epsilon$
    - ◇ $e^{n \lg(1+\epsilon/2n)}$
    - ◇ $\leq e^{n\epsilon/2n}$
    - ◇ $= e^{\epsilon/2}$
    - ◇ $\leq ?e^{\ln(1+\epsilon)}$
    - ◇ eso es equivalente a elegir ϵ tal que $\epsilon/2 \leq \ln(1+\epsilon)$
    - ◇ i.e. cualquier tal que $0 < \epsilon \leq 1$

  d) El algoritmo es polinomial (en todos los parametros)

  - despues de recortar dedos $y_i, y_{i+1} \in L$ se cumple $y_{i+1} > y_i(1+\delta)$ y el ultimo elemento es $\leq t$
  - entonces, la lista contiene $0, 1$ y luego a lo mas $\lfloor \log_{(1+\delta)} t \rfloor$
  - entonces el largo de L en cada iteracion no supera $2 + \frac{\log t}{\log(1+\varepsilon/2n)}$

- Nota que $\ln(1+x)$
    - $= -\ln(1/(1+x))$
    - $= -\ln((1+x-x)/(1+x))$
    - $= -\ln(1-x/(1+x))$
    - $= -\ln(1+y) \geq -y$
    - $\geq -(-x/(1+x)) = x/(1+x)$
- Entonces
    - $2 + \frac{\ln t}{\ln 1+\epsilon/2n}$
    - $\leq 2 + ((1+\epsilon/2n)2n\ln t)/\epsilon$
    - $= (2n\ln t)/\epsilon + \ln t + 2$
    - $= O(n\lg t/\epsilon)$
- Entonces cada iteracion toma $O(n\lg t/\epsilon)$ operaciones
- Las n iteraciones en total toman
    - $O(n^2\lg t/\epsilon)$ operaciones

1.5. Algoritmos paralelos y distribuidos (2 semanas = 4 charlas)

- Medidas de complejidad
- Tecnicas de diseno

1.5.1. PREREQUISITOS

- Chap 12 of "Introduction to Algorithms, A Creative Approach", Udi Manber, p. 375
- http://www.catonmat.net/blog/mit-introduction-to-algorithms-part-thirteenhttp://www.catonmat.net/blog/mit-introduction-to-algorithms-part-thirteen

1.5.2. Modelos de paralelismo y modelo PRAM

* Instrucciones

- SIMD: Single Instruccion, Multiple Data
- MIMD: Multiple Instruccion, Multiple Data

* Memoria

- compartida
- distribuida

* 2*2 combinaciones posibles:

	Memoria compartida	Memoria distribuida
SIMD	PRAM	redes de interconexion (weak computer units) (hipercubos, meshes, etc. . .)
MIMD	Threads	procesamiento distribuido (strong computer units), Bulk Synchronous Process, etc. . .

Modelo PRAM En este curso consideramos en particular el modelo PRAM.
* Mucha unidad de CPU, una sola memoria RAM
cada procesador tiene un identificador unico, y puede utilizarlo en el programa
* Ejemplo:

- if $p2 = 0$ then
  - $A[p] + = A[p-1]$
- else
  - $A[p] + = A[p+1]$
- $b \leftarrow A[p]$;
- $A[p] \leftarrow b$;

* Problema: el resultado no es bien definido, puede tener **conflictos** si los procesadores estan asinchronos. Las soluciones a este problemas dan varios submodelos del modelo PRAM:

1. EREW Exclusive Read. Exclusive Write
2. CREW Concurrent Read, Exclusive Write
3. CRCW Concurrent Read, Concurent Write En este caso hay variantes tambien:
   - todos deben escribir lo mismo
   - arbitrario resultado
   - priorizado
   - alguna $f()$ de lo que se escribe

Como medir el "trade-off.^{en}tre recursos (cantidad de procesadores) y tiempo? * DEFINICION:

- $T^*(n)$ es el **Tiempo secuencial** del mejor algoritmo no paralelo en una entrada de tamano n (i.e. usando 1 procesador).
- $T_A(n,p)$ es el **Tiempo paralelo** del algoritmo paralelo A en una entrada de tamano n usando p procesadores.
- El **Speedup** del algoritmo A es definido por
  $S_A(n,p) = \frac{T^*(n)}{T_A(n,p)} \leq p$
  Un algoritmo es mas efectivo cuando $S(p) = p$, que se llama **speedup perfecto**.
- La **Eficiencia** del algoritmo A es definida por
  $E_A(n,p) = \frac{S_A(n,p)}{p} = \frac{T^*(n)}{pT_A(n,p)}$
  El caso optima es cuando $E_A(n,p) = 1$, cuando el algoritmo paralelo hace la misma cantidad de trabajo que el algoritmo secuencial. El objetivo es de **maximizar la eficiencia**.

  (Nota estas definiciones en la pisara, vamos a usarlas despues.)

1.5.3. LEMMA de Brent, Trabajo y Consecuencias

PROBLEMA: Calcular $\max(A[1, ..., N])$

- Solucion Secuencial
  * Algoritmo:
    - $m \leftarrow 1$
    - for $i \leftarrow 2$ to n
      - if $A[i] > A[m]$ then $m \leftarrow i$
    - return $A[m]$
- tiempo $O(n)$, con 1 procesador, entonces:
- $T^*(n) = n$.
- Solucion Parallela con n procesadores
  * Algoritmo:
    - $M[p] \leftarrow A[p]$
    - for $l \leftarrow 0$ to $\lceil \lg p \rceil - 1$
      - if $p2^{l+1} = 0$ y $ p+2^l¡n$
        - $M[p] \leftarrow \max(M[p], M[p + 2^l])$
    - if $p = 0$
      - $max \leftarrow M[1]$
- tiempo $O(\lg n)$ con n procesador, i.e. en nuestras notaciones:
- $T(n, n) = \lg n$
- $S(n, n) = \frac{n}{\lg n}$
- $E(n, n) = \frac{n}{n \lg n} = \frac{1}{\lg n}$
  * Nota: no se puede hacer mas rapido, pero hay mucho procesadores poco usados: quizas se puede calcular el max en el mismo tiempo, pero usando menos procesadores?
- Solucion general con p procesadores
  * Idea:
    - reduce la cantidad de procesadores, y hace "load balancing" sobre $n/\lg n$ procesadores.
    - Divida el input en $n/\lg n$ grupos,
    - asigna cada grupo de $\lg n$ elementos a un procesador.
    - En la primera fase, cada procesador encontra el max de su grupo
    - En la segunda fase, utiliza el algoritmo precedente.

- tiempo $O(\lg n)$ con n procesador, i.e. en nuestra notaciones:
  - $T(n, \frac{n}{\lg n}) = 2 \lg n \in O(\lg n)$
    - $T(n,p) = \frac{n}{p} + \lg p$
  - $S(n,p) = \frac{n}{\frac{n}{p}+\lg p} = p(1 - \frac{p \lg p}{n+p \lg p}) \to p$ si $n \to \infty$
  - $E(n,p) = \frac{n}{\frac{n}{\lg n} \lg n} = 1/2$
- El parametro de $\lg n$ procesadores es optima?
  - Para que? Que significa ser optima?
    - en energia
    - en el contexto donde los procesadores libres pueden ser usados para otras tareas.
  - Si, es optimo para la eficiencia, se puede ver estudiando el grafo en funcion de p.
- Eso es un algoritmo EREW, CREW, o CRCW?
  - EREW (Exclusive Read. Exclusive Write): no dos procesadores lean o escriben en la misma cedula al mismo tiempo.
- Nota:
  - Hay un algoritmo CRCW que puede calcular el max en $O(1)$ tiempo en paralelo, ilustrando el poder del modelo CRCW (y el costo de las restricciones del modelo EREW) [REFERENCIA: Section 12.3.2 of "Introduction to Algorithms, A Creative Approach", Udi Manber, p. 382]]

LEMA de Brent El algoritmo previo illustra un principo mas general, llamado el "Lemma de Brent":
Si un algoritmo

- consigue un tiempo T(n,p)=C, entonces
- consigue tiempo T(n,p/s)= sC ∀ s¿1
- (bajo algunas condiciones, tal que hay suficientamente memoria para cada procesador)

DEFINICION "Trabajo

- Usando el Lema de Brent, podemos exprimir el rendimiento de los algoritmos paralelos con solamente dos medidas:
  - $T(n)$, el tiempo del mejor algoritmo paralelo usando cualquier cantidad de procesadores que quiere.

    Nota las diferencias con
    - $T^*(n)$, el tiempo del mejor algoritmo secuencial, y
    - $T_A(n,n)$, el tiempo del algoritmo A con n procesadores.

- $W(n)$, la suma del total trabajo ejecutado por todo los procesadores (i.e. superficia del arbol de calculo, a contras de su altura (tiempo) o hancho (cantidad de procesadores).

- INTERACCION: Cual son estas valores para el algoritmo de Max?
  - $T(n) =?$
  - $W(n) =?$
- INTERACCION: Puedes ver como desde $T(n)$, $W(n)$ se puede deducir las valores de
  - $T(n, p)$? (solucion en el corolario)
  - $S(n, p)$? (trivial desde $T(n, p)$)
  - $E(n, p)$? (solucion en el corolario)

COROLARIO

- Con el lema de Brent podemos obtener:
  - $$T(n, p) = T(n) + \frac{W(n)}{p}$$
  - $$E(n, p) = \frac{T^*(n)}{pT(n) + W(n)}$$

EJEMPLO * Para el calculo del maximo:

- $T(n) = \lg n$
- $W(n) = n$

* Entonces

- se puede obtener
  - $T_B(n, p) = \lg n + \frac{n}{p}$
  - $$E(n, p) = \frac{n}{p \lg n + n}$$

* (Nota que eso es solamente una cota superior, nuestro algoritmo da un mejor tiempo.)

1.5.4. PROBLEMA: Ranking en listas

1. DEFINICION
   - dado una lista, calcula el rango para cada elemento.
   - En el caso de una lista tradicional, no se puede hacer mucho mejor que lineal.
   - Consideramos una lista en un arreglo A,
     - donde cada elemento $A[i]$ tiene un puntero al siguiente elemento, $N[i]$, y

- calculamos su rango $R[i]$ en un arreglo R.

2. DoublingRank()
   - $R[p] \leftarrow 0$
   - if $N[p] = null$
     - $R[p] \leftarrow 1$
   - for $d \leftarrow 1$ to $\lceil \lg n \rceil$
     - if $N[p] \neq NULL$
       - if $R[N[p]] > 0$
         - $R[p] \leftarrow R[N[p]] + 2^d$
       - $N[p] \leftarrow N[N[p]]$
3. Analisis
   - $T(n) = \lg n$
   - $W(n) = n + W(n/2) \in O(n)$
   - $T(n,p) = T(n) + W(n)/p = \lg n + n/p$
   - $p^*\ T(n) = W(n)/p^*\ pn/\lg n$
   - $E(n,p^*) = \frac{T^*(n)}{p^*T(n)+W(n)} = \frac{n}{n/\lg n \lg n+n} \in \Theta(1)$
4. El algoritmo es EREW o CREW?
   - es EREW si los procesadores estan sincronizados, com en RAM aqua.

1.5.5. PROBLEMA: Prefijos en paralelo ("Parallel Prefix")

* DEFINICION: Problema "Prefijo en Paralelo"
Dado $x_1, \ldots, x_n$ y un operador asociativo $\times$, calcular

- $y_1 = x_1$
- $y_2 = x_1 \times x_2$
- $y_2 = x_1 \times x_2 \times x_3$
- $\ldots$
- $y_n = x_1 \times \ldots \times x_n$

* Solucion Secuencial
Hay una solucion obvio en tiempo $O(n)$.

Solucion paralela 1 * Concepto:

- Hipotesis: sabemos solucionarlo con $n/2$ elementos
- Caso de base: $n = 1$ es simple.
- Induccion:

  1. recursivamente calculamos en paralelo:
     - todos los prefijos de $\{x_1, \dots, x_{n/2}\}$ con $n/2$ procesadores.
     - todos los prefijos de $\{x_{n/2}, \dots, x_n\}$ con $n/2$ procesadores.
  2. en paralelo agregamos $x_{n/2}$ a los prefijos de $\{x_{n/2}, \dots, x_n\}$

* Observacion: en cual modelo de parallelismo es el ultimo paso?
  * ParallelPrefix1(i,j)

- if $i_p = j_p$
  - return x_{i_p}
- $m_p \leftarrow \lfloor \frac{i_p + j_p}{2} \rfloor$;
- if $p \leq m$ then
  - algo(i_p, m_p)
- else
  - algo($m + 1, j_p$)
  - $y_p \leftarrow y_m.y_p$

* Otra forma de escribir el algoritmo (de p. 384 de "Introduction to Algorithms, A Creative Approach", Udi Manber):

- ParallelPrefix1(left,right)
  - if $(right - left) = 1$
    - $x[right] \leftarrow x[left].x[right]$
  - else
    - $middle \leftarrow (left + right - 1)/2$
    - do in paralel
      - $ParallelPrefix1(left, middle)$\{ assigned to $\{P_1 to P_{n/2}\}$\}
      - $ParallelPrefix1(middle + 1, right)$\{ assigned to $\{P_{n/2+1} to P_n\}$\}
    - for $i \leftarrow middle + 1$ to $right$ do in paralel

◇ $x[i] \leftarrow x[middle].x[i]$

* Notas:

- este solucion **no** es EREW (Exclusive Read and Write), porque los procesadores pueden leer y_m al mismo tiempo.
- este soluciono es CREW (Concurrent Read, Exclusive Write).
- Complejidad:
  - $T_{A_1}(n,n) = 1 + T(n/2, n/2) = \lg n$
    (El mejor tiempo en paralelo con cualquier cantidad de procesadores.)
  - $W_{A_1}(n) = n + 2W_{A_1}(n/2) - n \lg n$
  - $T_{A_1}(n,p) = T(n) + W_{A_1}(n)/p = \lg n + (n \lg n)/p$
  - Calculamos p^*, la cantidad optima de procesadores para minimizar el tiempo:
    - $T(n) = W_{A_1}(n)/p^*$
    - $p^* = \frac{W(n)}{T(n)} = n$
  - Calculamos la eficiencia
    - $E_{A_1}(n,p^*) = \frac{T^*(n)}{p^*T(n)+W(n)} = \frac{n}{n \lg n} = \frac{1}{\lg n}$
    - es poco eficiente =(
    - Podriamos tener un algoritmo con
      - ◇ la eficiencia del algoritmo secuencial
      - ◇ el tiempo del algoritmo paralelo?

Solucion paralela 2: mismo tiempo, mejor eficiencia

- Idea:
  El concepto es de dividir de manera diferente: par y impar (en vez de largo o pequeno).
- Concepto:
  1. Calcular en paralelo $x_{2i-1}.x_{2i}$ en x_{2i} para $\forall i, 1 \leq i \leq n/2$.
  2. Recursivamente, calcular todos los prefijos de $E = \{x_2, x_4, \ldots, x_{2i}, \ldots, x_n\}$
  3. Calcular en paralelo los prefijos en posiciones impares, multiplicando los prefijos de E por una sola valor.
- algo2(i,j)
  - for $d \leftarrow 1$ to $(\lg n) - 1$
    - if $p = 02^{d+1}$

- ◇ if $p + 2^d < n$
- ◇ $x_{p+2^d} \leftarrow x_p.x_{p+2^d}$

- for $d \leftarrow 1$ to $(\lg n) - 1$
  - if $p = 02^{d+1}$
    - if $p - 2^d > 0$
    - $x_{p-2^d} \leftarrow x_{p-2^d}.x_p$

- visualizacion
  1.
  2. (0,1)
  3.
  4. (2,3) (0,3)
  5.
  6. (4,5)
  7.
  8. (6,7) (4,7) (0,7)
  9.
  10. (8,9)
  11.
  12. (10,11) (8,11)
  13.
  14. (12,13)
  15.
  16. (14,15) (12,15) (8,15) (0,15)

- Notas:
  - Este algoritmo es EREW

- Analisis
  - $T_2(n) = 2 \lg n$
  - $W(n) = n + W(n,2) = n$
  - $T_2(n,p) = T(n) + W(n)/p = 2 \lg n + \frac{n}{p}$
  - Calculamos la cantidad optima de procesadores para obtener el tiempo optima:
    - $T_2(n) = W(n)/p^*$ entonces
    - $p^* = \frac{W(n)}{T(n)} = \frac{n}{\lg n}$
  - $E_2(n,p^*) = \frac{T^*(n)}{p^*T(n)+W(n)} \in O(1)$

1.5.6. Moralidad del Parallelismo:

1. Cual es la consecuencia del Lemma de Brent?

    - Concentrarse en $T(n)$ y $E(n)$
    - Pero saber aplicar el Lemma de Brent para programmar $T(n,p)$

2. Cual (otra) tecnica veamos?

    - Mejorar la efficiencia con una parte secuencial (en parallelo) del algoritmo.

1.6. Conclusion Unidad

* Vimos

1. Aleatorizacion
2. Aproximabilidad
3. Paralelizacion / Distribucion

* Contexto

- son **extenciones** del contenido del curso
- hay muchas otras
    - cryptografia
    - quantum computing
    - parameterized complexity
    - ...
- La metodologia entre todas tiene una parte en comun:
    - Formalismo
        - Cotas superiores
        - Cotas inferiores
    - Adecuacion a la practica.

[1] FOOTNOTE DEFINITION NOT FOUND: 0