Algoritmos Aleatorizados

Gabriel Diéguez Franzani

November 29, 2011

Contenidos

Motivación

- Existen situaciones en que los algoritmos convencionales presentan problemas
  - Alta complejidad
    - En el sentido "clásico", no el de Ciencia de la Computación
  - Casos extremadamente malos... como Quicksort con una entrada ordenada
- Parece ser que el determinismo no siempre es nuestro aliado
- **Idea:** usar una componente aleatoria!

Algoritmos aleatorizados

Definición

Un **algoritmo** se dice **aleatorizado** si usa algún grado de aleatoriedad como parte de su lógica.

- Dos grandes ventajas:
  - Simpleza
  - Rapidez
- Pero hay un costo...
  - El tiempo de ejecución del algoritmo, su output o ambos se convierten en **variables aleatorias**.

Simpleza y rapidez

Ejemplo: RandQS

- El “mejor” algoritmo de ordenación?
  - Quicksort!
- Su mayor dificultad: elegir el pivote...
- Y si lo elegimos aleatoriamente?
  - Mucho más simple!
  - Tiempo esperado de ejecución: $O(n \log n)$, **independiente** del input!

Las Vegas y *Monte Carlo*

Hablamos del tiempo de ejecución y del output de los algoritmos aleatorizados como variables aleatorias. A partir de esto definimos dos tipos de AAs:

- **Las Vegas:** algoritmos que siempre entregan el resultado correcto, pero cuyo tiempo de ejecución varía (incluso para el mismo input).
    - Un algoritmo Las Vegas se dice **eficiente** si su tiempo esperado de ejecución es polinomial para cualquier entrada.
- **Monte Carlo:** algoritmos que pueden entregar resultados incorrectos con una probabilidad acotada.
    - Podemos disminuir esta probabilidad de error repitiendo el algoritmo... a expensas del tiempo de ejecución.
    - Un algoritmo Monte Carlo se dice **eficiente** si su tiempo de ejecución en el peor caso es polinomial para cualquier entrada.

Un poco de formalidad

- El modelo subyacente a los algoritmos aleatorizados es la **Máquina de Turing probabilística**.
- Usa un string de bits aleatorios distribuidos uniformemente como input auxiliar.
- Idea: alcanzar un buen rendimiento en el **caso promedio** sobre todas las posibles elecciones de bits aleatorios.
- Las variables aleatorias que definen el tiempo de ejecución y el output del algoritmo están determinadas por estos bits aleatorios.

RP (Randomized Polynomial time)

Lenguajes que tienen un algoritmo aleatorizado A tal que:

- Su tiempo de ejecución es polinomial en el peor caso
- Para cualquier entrada x en Σ^*:
  - $x \in L \Rightarrow Pr[A \text{ acepta } x] \geq \frac{1}{2}$
  - $x \notin L \Rightarrow Pr[A \text{ acepta } x] = 0$

Observación

Un algoritmo en RP es un algoritmo *Monte Carlo* que sólo falla cuando $x \in L$. Esto se conoce como *one-sided error*.

Algunas clases de complejidad

ZPP (Zero-error Probabilistic Polynomial time)

Lenguajes que tienen un algoritmo aleatorizado A tal que:

- No fallan en el output
- Su tiempo de ejecución es polinomial en el caso promedio

En otras palabras, ZPP contiene a los algoritmos *Las Vegas* que corren en tiempo polinomial esperado.

Observación

$ZPP = RP \cap co - RP$

Algunas clases de complejidad

PP (Probabilistic Polynomial time)

Lenguajes que tienen un algoritmo aleatorizado A tal que:

- Su tiempo de ejecución es polinomial en el peor caso
- Para cualquier entrada x en Σ^*:
  - $x \in L \Rightarrow Pr[A \text{ acepta } x] > \frac{1}{2}$
  - $x \notin L \Rightarrow Pr[A \text{ acepta } x] < \frac{1}{2}$

Observación

Un algoritmo en PP es un algoritmo *Monte Carlo* que puede fallar tanto cuando $x \in L$ como cuando $x \notin L$. Esto se conoce como *two-sided error*.

RP v/s PP

- Es fácil notar que como RP representa a los algoritmos *Monte Carlo one-sided error*, la repetición de un algoritmo de esta clase puede reducir arbitrariamente su probabilidad de error.
- Pero para un algoritmo PP, esto no es necesariamente cierto con una cantidad polinomial de repeticiones, ya que no sabemos qué tan lejos de $\frac{1}{2}$ están las probabilidades:

Proposición

Una cantidad polinomial de repeticiones de un algoritmo en PP puede no ser suficiente para reducir la probabilidad de error a $\frac{1}{4}$.

Y entonces?

Una clase más útil para *two-sided error*

BPP (Bounded-error Probabilistic Polynomial time)

Lenguajes que tienen un algoritmo aleatorizado A tal que:

- Su tiempo de ejecución es polinomial en el peor caso
- Para cualquier entrada x en Σ^*:
  - $x \in L \Rightarrow Pr[A \text{ acepta } x] \geq \frac{3}{4}$
  - $x \notin L \Rightarrow Pr[A \text{ acepta } x] \leq \frac{1}{4}$

Proposición

La probabilidad de error de un algoritmo en BPP puede reducirse a $\frac{1}{2^n}$ con una cantidad polinomial de repeticiones.

Algunas propiedades

- $P \subseteq RP \subseteq NP$
- $RP \subseteq BPP \subseteq PP$
- $PP = co - PP$, $BPP = co - BPP$
- $NP \subseteq PP \subseteq PSPACE$

Ejercicio

Demuestre las propiedades anteriores.

Aplicación: Estructuras de datos

El problema fundamental de las estructuras de datos

Mantener conjuntos $\{S_1, S_2, ...\}$ de objetos pertenecientes a un universo ordenado U, de manera de soportar operaciones de búsqueda, actualización y otras eficientemente.

- Supondremos que los objetos están representados por su clave k.
- Cada objeto pertenece sólo a un conjunto, y su clave es única.

Operaciones

- $MAKESET(S)$: crear un nuevo conjunto S (vacío).
- $INSERT(i, S)$: insertar el objeto i en un conjunto S.
- $DELETE(k, S)$: borrar el objeto indexado con clave k del conjunto S.
- $FIND(k, S)$: retornar el objeto indexado con clave k en el conjunto S.
- **Otras:** $JOIN(S_1, i, S_2)$, $PASTE(S_1, S_2)$, $SPLIT(k, S)$.

Solución estándar: **Árboles de búsqueda binarios**

Árboles de búsqueda binarios

- Guardamos claves en los nodos.
- Cada nodo tiene exactamente dos hijos.
- Se cumple la propiedad de ABB:

Propiedad de ABB

Para cada nodo v, sea $L(v)$ su hijo izquierdo y $R(v)$ su hijo derecho:

- Si $L(v)$ existe, entonces $L(v).k < v.k$
- Si $R(v)$ existe, entonces $R(v).k > v.k$

- Las operaciones mencionadas pueden realizarse en tiempo proporcional a la altura del árbol.

Altura?

- Entonces sólo debemos asegurar una altura razonable!
- Pero es muy fácil idear una secuencia de inserción malévola.
  - Basta con insertar en orden...
- Una solución: mantener un balance en el árbol con tal de asegurar una altura logarítmica en la cantidad de nodos.
  - Implica hacer rotaciones al insertar o borrar elementos.
  - Puede llegar a ser muy costoso para ciertas secuencias de operaciones.
  - Asegura tiempo de ejecución $O(\log n)$ para las operaciones descritas.
- Otra solución: aplicar algoritmos aleatorizados!

Treaps

- En cada nodo guardamos dos valores: una clave $k(v)$ y una prioridad $p(v)$.
- Las claves cumplen la propiedad de ABB.
- Las prioridades cumplen la siguiente propiedad:

Propiedad de heap

Para cada nodo v, $p(v) > p(u)$, para todo hijo u de v.

Ejercicio

Construir un treap para el conjunto
$\{(2,13),(4,26),(6,19),(7,30),(9,14),(11,27),(12,22)\}$

Existencia

Para un conjunto dado de pares (clave, prioridad), existe siempre un Treap?

Teorema (Existencia y Unicidad de Treaps)

Sea $S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$ un conjunto de pares clave-prioridad cualquiera tal que las claves y prioridades son todas distintas. Entonces, existe un único Treap $T(S)$.

Demostración: usamos una prueba constructiva y recursiva.
Es claro que el teorema se cumple para $n = 0$ y $n = 1$. Supongamos ahora que $n \geq 2$, y que (k_1, p_1) tiene la mayor prioridad en S. Entonces, un treap para S puede construirse usando el primer objeto como raíz. A continuación, construimos un treap para los elementos en S con clave menor a k_1 recursivamente, y todo este treap será el subárbol izquierdo de la raíz. De manera análoga se construye el treap para los demás elementos. Finalmente, es fácil ver que todo treap para S tendrá exactamente la misma composición al mirarlo desde la raíz. $\square$

Estructura de un treap

- La estructura de un treap depende de las prioridades relativas de los pares del conjunto.
- De hecho, podemos forzar su estructura eligiendo las prioridades convenientemente.
- Entonces, si somos capaces de generar un conjunto de prioridades adecuado, podemos obtener un treap de profundidad baja... y resolver el problema fundamental de las estructuras de datos!
  - Alguna idea?
  - Prioridades aleatorias!

Random treaps

- En un *random treap*, las prioridades se eligen aleatoriamente de una distribución que asegure no repetición de prioridades.
- Como el ordenamiento de las prioridades será completamente independiente del de las claves, la estructura del treap permanecerá balanceada, y tendrá una profundidad esperada de $O(\log n)$.
- Además, como la elección de las prioridades se mantiene "escondida", es imposible para un adversario invocar una secuencia de operaciones que desbalancee el treap.
- Suena lindo, pero se puede demostrar?
  - **Sí**, pero necesitamos una herramienta: los Juegos de Mulmuley.

Juegos de Mulmuley

Participantes:

- **Jugadores:** $P = \{P_1, ..., P_p\}$
- **Stoppers:** $S = \{S_1, ..., S_s\}$
- **Gatilladores:** $T = \{T_1, ..., T_t\}$
- **Espectadores:** $B = \{B_1, ..., B_b\}$

Reglas:

- El conjunto $P \cup S$ se obtiene de un universo totalmente ordenado.
- Todos los jugadores son menores que todos los *stoppers*: para todo i, j, $P_i < S_j$.
- Los conjuntos son disjuntos a pares.

Juegos de Mulmuley

Dependiendo de los personajes que participan, se definen cuatro juegos, pero primero veamos una propiedad útil.

Propiedad

Sea $H_k = \sum_{i=1}^{k} \frac{1}{i}$ el k-ésimo número armónico.
Se cumple que $\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1)$

Recordatorio

$H_k = \ln k + O(1)$

Juegos de Mulmuley: Juego A

Juego A

- Se comienza con el conjunto de personajes $X = P \cup B$.
- El juego consiste en sacar personajes de X sin reemplazo, hasta que X queda vacío.
- Cada personaje se elige aleatoriamente de manera uniforme entre los que quedan en X.
- Sea V una variable aleatoria, definida como el número de veces que un jugador P_i es elegido de manera tal que es mayor que todos los jugadores elegidos anteriormente.
- Definimos el **valor** del juego A_p como $E[V]$.

Juegos de Mulmuley: Juego A

Lema

Para todo $p \geq 0$, $A_p = H_p$.

Demostración: Suponemos que el conjunto de jugadores está ordenado como $P_1 > P_2 > ... > P_p$.

- Notemos que los espectadores son absolutamente irrelevantes: el valor del juego no está influenciado por la cantidad de espectadores. Podemos asumir entonces que la cantidad de espectadores es $b = 0$.
- Sea P_i el primer jugador obtenido. Tenemos entonces que el valor esperado del juego es $1 + A_{i-1}$. Esto es porque los demás jugadores $(P_{i+1}, ..., P_p)$ ya no pueden aportar al valor del juego, y en la práctica se convierten en espectadores.

Juegos de Mulmuley: Juego A

- Ya que i distribuye uniformemente sobre $\{1, ...p\}$, obtenemos la recurrencia $A_p = \sum_{i=1}^{p} \frac{1+A_{i-1}}{p} = 1 + \sum_{i=1}^{p} \frac{A_{i-1}}{p}$.
- Reacomodando los términos, y sabiendo que $A_0 = 1$, tenemos que $\sum_{i=1}^{p-1} A_i = pA_p - p$.
- Finalmente, usando la propiedad de los números armónicos enunciada anteriormente, concluimos que $A_p = H_{p\Box}$

Juegos de Mulmuley: Juego C

Juego C

- Se comienza con el conjunto de personajes $X = P \cup B \cup S$.
- El juego se desarrolla igual que antes, considerando a los *stoppers* como jugadores.
- La diferencia radica en que una vez que se elige el primer stopper, se acaba el juego.
- Dado que los stoppers son todos mayores que los jugadores, el primer stopper siempre agregará 1 al valor del juego.
- Definimos el **valor** del juego como $C_p^s = E[V+1] = 1 + E[V]$, con V definida como en el juego A.

Juegos de Mulmuley: Juego C

Lema

Para todo $p, s \geq 0$, $C_p^s = 1 + H_{s+p} - H_s$.

Demostración: Asumimos el mismo orden de antes $(P_1 > P_2 > ... > P_p)$ y que la cantidad de espectadores es 0.

- Si la primera jugada es un stopper, el valor del juego es 1. La probabilidad de que esto ocurra es $\frac{s}{s+p}$.
- Si la primera jugada es un jugador P_i, el valor del juego es $1 + C_{i-1}^s$. Esto pasa con probabilidad $\frac{1}{s+p}$.

Así, obtenemos la siguiente recurrencia:

$$C_p^s = (\frac{s}{s+p} \times 1) + (\frac{1}{s+p} \times \sum_{i=1}^{p} (1 + C_{i-1}^s))$$

Reordenando y teniendo en cuenta que $C_0^s = 1$:

$$C_p^s = \frac{s+p+1}{s+p} + \frac{\sum_{i=1}^{p-1} C_i^s}{s+p}$$

y equivalentemente:

$$\sum_{i=1}^{p-1} C_i^s = (s+p)C_p^s - (s+p+1)$$

Entonces, usando nuevamente la propiedad de los números armónicos, concluimos que la solución de la recurrencia es

$$C_p^s = 1 + H_{s+p} - H_s \square$$

Juegos de Mulmuley: Juegos D y E

Juegos D y E

- Muy similares a los juegos A y C.
- La diferencia es que los conjuntos de personajes son $X = P \cup B \cup T$ y $X = P \cup B \cup S \cup T$ respectivamente.
- El conteo empieza después de sacar el primer gatillador.
- Entonces, un jugador o stopper contribuye a V si y sólo si son sacados después de un gatillador, antes de cualquier stopper y si es mayor que todos los jugadores escogidos previamente.

Sean D_p^t y $E_p^{s,t}$ los valores esperados de los juegos D y E respectivamente.

Lema

Para todo $p, t \geq 0$, $D_p^t = H_p + H_t - H_{p+t}$.

Lema

Para todo $p, s, t \geq 0$, $E_p^{s,t} = \frac{t}{s+t} + (H_{s+p} - H_s) - (H_{s+p+t} - H_{s+t})$.

Ejercicio

Demuestre los lemas anteriores.

Análisis de Treaps

Para aplicar los Juegos de Mulmuley al análisis de desempeño de *random treaps*, será útil la siguiente propiedad:

Propiedad (Carencia de memoria)

- Sea un *random treap* obtenido tras insertar los elementos de un conjunto S en un treap inicialmente vacío.
- Como las prioridades aleatorias de los elementos en S se eligen de manera independiente, podemos asumir que se asignan antes de iniciar la inserción de ellos.
- Una vez que hemos fijado las prioridades, el teorema de Existencia y Unicidad de treaps implica que existe un único treap T para los elementos de S con las prioridades asignadas.

Análisis de Treaps

Propiedad (Carencia de memoria)

- Entonces, el orden en que se insertan los elementos de S no altera la estructura del treap T.
- Luego, sin pérdida de generalidad, podemos asumir que **los elementos de S fueron insertados en T en orden decreciente respecto a la prioridad**.
- Esto significa que todas las inserciones se realizan en las hojas, y que además no necesitamos rotaciones para conservar la propiedad de treap.

Análisis de Treaps

Sea la **profundidad** de un nodo x en un treap su distancia a la raíz, denotada como $depth(x)$.
Sea el **rango** de un elemento en un conjunto su posición en un orden creciente del conjunto.

Lema

Sea T un random treap para un conjunto S con n elementos. Dado un elemento $x \in S$ con rango k:

$$E[depth(x)] = H_k + H_{n-k+1} - 1$$

Demostración: Sean los conjuntos

$$S^- = \{y \in S \mid y \leq x\},\ S^+ = \{y \in S \mid y \geq x\}$$

Dado que x tiene rango k, es claro que

$$|S^-| = k,\ |S^+| = n - k + 1$$

Sea $Q_x \subseteq S$ el conjunto de elementos que están guardados en nodos en el camino desde la raíz de T hasta el nodo que contiene a x; en otras palabras, Q_x contiene a los ancestros de x en T. Sean además

$$Q_x^- = S^- \cap Q_x,\ Q_x^+ = S^+ \cap Q_x$$

La idea de la demostración es la siguiente: supongamos que

$$E[|Q_x^-|] = H_k$$

Por simetría, el tamaño esperado de Q_x^+ será H_{n-k+1}. Esto implicará que el largo esperado del camino de la raíz a x será

$$H_k + H_{n-k+1} - 1$$

como queremos demostrar, dado que $Q_x^- \cap Q_x^+ = \{x\}$.

Luego, sólo nos basta demostrar nuestra suposición inicial!

PD: $E[|Q_x^-|] = H_k$

- Tomemos un ancestro $y \in Q_x^-$ de x. Por carencia de memoria, y debió haber sido insertado antes que x, y sus prioridades satisfacen $p_x > p_y$.
- Como $y < x$, x debe estar en el sub-árbol derecho de y. De hecho, todo elemento z tal que $y < z < x$ está en el sub-árbol derecho de y.
- En otras palabras, y es ancestro de cada nodo que contenga un elemento con clave entre y y x.
- Usando el orden de inserción que asumimos anteriormente, deducimos que todo elemento cuya clave esté entre y y x debió haber sido insertado después de y, y luego tiene menor prioridad que y.

Análisis de Treaps

- El razonamiento anterior es equivalente a decir que un elemento $y \in S^-$ es un ancestro de x (o pertenece a Q_x^-) si y sólo si era el elemento con mayor clave en S^- al momento de su inserción.
- Dado que el orden de inserción está determinado por el orden de las prioridades, y éstas distribuyen uniformemente, podemos decir que el orden de inserción está dado por un muestreo sin reemplazo del conjunto S.
- Entonces, podemos ver que **la distribución de $|Q_x^-|$ es la misma que la del valor del Juego de Mulmuley A**, haciendo

$$P = S^- \text{ y } B = S \setminus S^-$$

- Finalmente, como $|S^-| = k$, concluimos que $E[|Q_x^-|] = H_k \square$

Análisis de Treaps

Ya dijimos que el tiempo que toman las operaciones $FIND$, $INSERT$ y $DELETE$ es proporcional a la altura del treap. Sin embargo, podemos establecer algo más fuerte: Sea la **columna derecha** de un árbol el camino obtenido al empezar en la raíz y moverse repetidamente hacia la derecha, hasta llegar a una hoja. Se define la **columna izquierda** similarmente.

Proposición

El número de rotaciones necesarias durante la eliminación de un nodo v es igual a la suma de los largos de la columna izquierda del sub-árbol derecho y de la columna derecha del sub-árbol izquierdo de v.

Ejercicio

Demuestre la proposición y enuncie un resultado equivalente para la operación $INSERT$.

Análisis de Treaps

Sea ahora L_x el largo de la columna izquierda del sub-árbol derecho de un elemento x en un treap, y R_x el largo de la columna derecha del sub-árbol izquierdo de x.

Lema

Sea T un random treap para un conjunto S con n elementos. Dado un elemento $x \in S$ con rango k:

$$E[R_x] = 1 - \frac{1}{k}$$

$$E[L_x] = 1 - \frac{1}{n-k+1}$$

Demostración: demostraremos que la distribución de R_x es la misma que la del valor del Juego de Mulmuley D, tomando

$$P = S^- \setminus \{x\},\ T = \{x\},\ B = S^+ \setminus \{x\}$$

con S^- y S^+ definidos como antes.

Como $p = k - 1$, $t = 1$ y $b = n - k$, del lema para el valor del Juego D obtenemos que

$$E[R_x] = D^1_{k-1} = H_{k-1} + H_1 - H_k = 1 - \frac{1}{k}$$

Para relacionar el largo de la columna derecha del sub-árbol izquierdo de x al Juego D, hacemos la siguiente proposición:

Proposición

Un elemento $z < x$ está en la columna derecha del sub-árbol izquierdo de x si y sólo si z es insertado después de x, y todos los elementos cuyo valor está entre z y x son insertados después de z.

Demostración:

($\Leftarrow$): Tomemos el camino recorrido por el procedimiento de inserción de z, al localizar la hoja donde z será insertado. Este camino debe pasar por el nodo que contiene a x, dado que la única manera de distinguir entre z y x es comparándolos con un elemento que esté entre ellos, los cuales son todos insertados después de z.

Análisis de Treaps

Como z es menor que x y es insertado después que x, debe estar en el sub-árbol izquierdo de x. Además, dado que todos los elementos en el sub-árbol izquierdo de x son menores que x, y z es el mayor de ellos al momento de su inserción, z debe estar en la columna derecha de este sub-árbol.

(⇒): Dado que z está en el sub-árbol izquierdo de x, debe haber sido insertado después de x y ser menor que x. Además, todos los elemenos con valor entre z y x deben estar en el sub-árbol izquierdo de x, y como z está en la columna derecha del sub-árbol, estos elementos deben haber sido insertados después de z. $\square$

Ejercicio

Demostrar la segunda parte del lema.

Análisis de Treaps

Finalmente, usando todos los lemas y proposiciones enunciadas, es fácil demostrar el siguiente teorema:

Teorema (Desempeño de un *Random Treap*)

1. El tiempo esperado para las operaciones $FIND$, $INSERT$ y $DELETE$ en un random treap T es $O(\log n)$.
2. El número esperado de rotaciones requeridas durante una inserción o eliminación es a lo más 2.
3. El tiempo esperado para las operaciones $JOIN$, $PASTE$ y $SPLIT$ que involucren conjuntos S_1 y S_2 de tamaño n y m respectivamente es $O(\log n + \log m)$.