Taller 8

Andrés Cocunubo

April 8, 2018

1. Construcción Árbol Óptimo

   Para la resolución del problema se utilizaron los pseudocódigos brindados por el docente.

   (a) Estructura Óptima del problema

   $$e\left[i,j\right]=\begin{cases} q_{i-1} & ;\quad j=i-1 \\ \min\limits_{i\leq r\leq j}\left\{e\left[i,r-1\right]+e\left[r+1,j\right]+w\left(i,j\right)\right\} & ;\quad i\leq j \end{cases}$$

   $$w\left(i,j\right)=\sum_{l=i}^{j}p_l+\sum_{l=i-1}^{j}q_l$$

   (b) Algoritmo Recurrente

Algorithm 1 Algoritmo Recurrente

```
procedure BUILD_OPTBINTREE(P, Q)
    return BUILD_OPTBINTREE_REC (P, Q, 1, |P|)
end procedure
```

```
procedure BUILD_OPTBINTREE_REC(P, Q, i, j)
    if j = i − 1 then
        return Q [i − 1]
    else
        w ← DUMMY_WEIGHT (P, Q, i, j)
        e ← ∞
        for r ← i to j do
            v_l ← BUILD_OPTBINTREE_REC (P, Q, i, r − 1)
            v_r ← BUILD_OPTBINTREE_REC (P, Q, r + 1, j)
            v ← v_l + v_r + w
            if v < e then
                e ← v
            end if
        end for
        return e
    end if
end procedure
```

```
procedure DUMMY_WEIGHT(P, Q, i, j)
    w ← Q [i − 1]
    for l ← i to j do
        w ← w + P [l] + Q [l]
    end for
    return w
end procedure
```

(c) Algoritmo Recurrente <<memoizado>>

Algorithm 2 Memoizado

```
procedure Build_OptBinTree(P, Q)
    let W be a matrix [1..|P|] × [1..|P|]
    let M be a matrix [0..|P|] × [0..|P|]
    W ← 0
    M ← 0
    for i ← 1 to |P| do
        W[i, i] ← Q[i − 1] + P[i] + Q[i]
        M[i, i] ← Q[i − 1]
        for j ← i + 1 to |P| do
            W[i, j] ← W[i, j − 1] + P[j] + Q[j]
            M[i, j] ← ∞
        end for
    end for
    return Build_OptBinTree_Rec(P, Q, 1, |P|, M, W)
end procedure
```

```
procedure Build_OptBinTree_Rec(P, Q, i, j, M, W)
    if M[i, j] = ∞ then
        if j = i − 1 then
            M[i, j] ← Q[i − 1]
        else
            for r ← i to j do
                v_l ← Build_OptBinTree_Rec(P, Q, i, r − 1, M, W)
                v_r ← Build_OptBinTree_Rec(P, Q, r + 1, j, M, W)
                v ← v_l + v_r + W[i, j]
                if v < M[i, j] then
                    M[i, j] ← v
                end if
            end for
        end if
    end if
    return M[i, j]
end procedure
```

La tabla de memoización tiene la siguiente forma:

$M[i,j]$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	$Q[0]$	$M[1,1]$	$M[1,2]$	$M[1,3]$	$M[1,4]$	$M[1,5]$
2	0	$Q[1]$	$M[2,2]$	$M[2,3]$	$M[2,4]$	$M[2,5]$
3	0	0	$Q[2]$	$M[3,3]$	$M[3,4]$	$M[3,5]$
4	0	0	0	$Q[3]$	$M[4,4]$	$M[4,5]$
5	0	0	0	0	$Q[4]$	$M[5,5]$

Y w puede ser pre-calculado:

$W[i,j]$	1	2	3	4
1	$Q[0]+P[1]+Q[1]$	$W[1,1]+P[2]+Q[2]$	$W[1,2]+P[3]+Q[3]$	$W[1,3]+P[4]$
2	0	$Q[1]+P[2]+Q[2]$	$W[2,2]+P[3]+Q[3]$	$W[2,3]+P[4]$
3	0	0	$Q[2]+P[3]+Q[3]$	$W[3,3]+P[4]$
4	0	0	0	$Q[3]+P[4]+$
5	0	0	0	0

(d) Algoritmo <<*Bottom-up*>> con la solución

Algorithm 3 Bottom-up

```
procedure Build_OptBinTree(P, Q)
    let W be a matrix [1..|P|] × [1..|P|]
    let M be a matrix [0..|P|] × [0..|P|]
    let R be a matrix [1..|P|] × [1..|P|]
    W ← 0
    M ← 0
    R ← 0
    for i ← 1 to |P| do
        W[i, i] ← Q[i − 1] + P[i] + Q[i]
        M[i, i] ← Q[i − 1]
        for j ← i + 1 to |P| do
            W[i, j] ← W[i, j − 1] + P[j] + Q[j]
            M[i, j] ← ∞
        end for
    end for
    for l ← 1 to |P| do
        for i ← 1 to |P| − l + 1 do
            j ← i + l − 1
            for r ← i to j do
                v ← M[i, r − 1] + M[r + 1, j] + W[i, j]
                if v < M[i, j] then
                    M[i, j] ← v
                    R[i, j] ← r
                end if
            end for
        end for
    end for
    return R
end procedure
```

(e) Complejidad:

i. Hasta el bottom-up

- Para encontrar el árbol óptimo después de realizar el *bottom-up* se encontró que el llenado de la tabla M que contiene los porcentajes de cada árbol es llenada de la siguiente manera:

- Para realizar el recorrido anterior se necesitaron tres ciclos, en los cuales los dos más exteriores recorrían el vector de probabilidades P y el ciclo más interno realiza un recorrido de en el peor de los casos $\frac{n}{2}$ debido a que recorre sesgadamente una diagonal.
- A partir de lo dicho anteriormente, se determina por inspección que la solución que contiene los porcentajes del árbol óptimo tiene una complejidad de $O(n^{\mathbf{3}})$.

ii. Backtracking

- En este paso, se tomaron los valores obtenidos en la tabla de resultados R, para construir el árbol óptimo
- Iniciando desde la posición $R[0][n]$, donde se encuentra la raíz del árbol óptimo, se utilizó la estructura de datos *queue* de la STL, para agregar los diferentes nodos y calcular sus hijos con los índices dados, hasta que se completaran la cantidad de nodos equivalente a la cantidad de palabras que se quieren almacenar.
- Por lo tanto, como se recorren los elementos hasta n, la complejidad por inspección es $O(n)$.

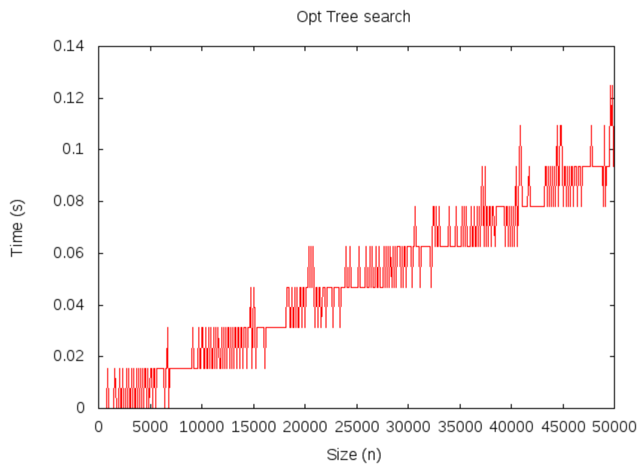
iii. Búsqueda

- Como el árbol posee un orden en el cual los elementos de la izquierda son menores a los de la derecha, la búsqueda recurrente consiste en comparar con la raíz y de acuerdo a esto de decide si buscar al lado izquierdo o derecho, lo que según el teorema maestro tiene un complejidad $O(logn)$, debido a que solo hay un llamado recurrente dada una condición.

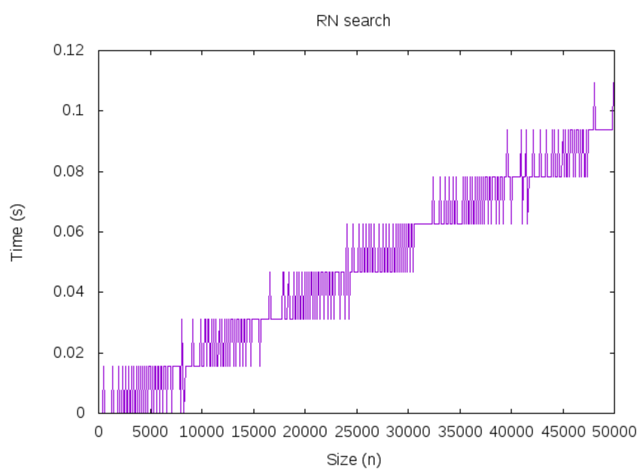
2. Comparación Árbol Óptimo y Rojo y negro

- Para la realización del experimento se midieron los tiempos que tardaban cada algoritmo en realizar búsquedas de palabras aleatorias. Las siguientes gráficas fueron los resultados obtenidos para cada uno y además la comparación de la dos.

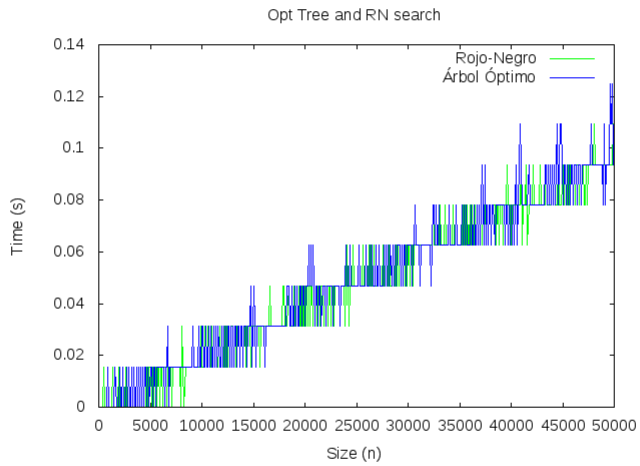
(a) Árbol Óptimo



(b) Rojo y negro



(c) Comparación de ambas estructuras



- Como se observa en las gráficas en algunas casos el árbol óptimo presenta una mejora mínima.

3. Solución Voraz

$$e\,[i,j] = \begin{cases} w(i,j) & ; \quad i = r \wedge j = r \\ w(i,j) + e[r+1,j] & ; \quad i = r \\ e[i,r-1] + w(i,j) & ; \quad j = r \\ e[i,r-1] + e[r+1,j] & ; \quad i \wedge j \neq r \end{cases}$$

$$r = max(i,...,j)$$

$$w\,(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

4. Modo de uso

Se debe ejecutar taller.cxx para el árbol óptimo y RNTree.cxx para el rojo y negro con c++11.

Ejemplo ejecución en ubuntu:

1. Abrir una terminal en linux en el directorio del programa.
2. Compilar en la terminal "g++ -std=c++11 taller.cxx -o test ".
3. Ejecutar "./test"