Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Tarea 2
KD-Trees

Autores: Claudio Berroeta
Sebastián Ferrada
Profesor: Pablo Barceló
Auxiliar: Miguel Romero
Ayudantes: Javiera Born
Giselle Font
5 de junio de 2014
Santiago, Chile.

Índice

1. Introducción

En el presente informe se pretende mostrar un análisis sobre la performance de diferentes algoritmos clásicos de ordenamiento, sometiéndolos a diversos experimentos para comparar así la cantidad de comparaciones esperadas con las efectuadas por los algoritmos al ordenar arreglos de diferentes tamaños y con distintas topologías, para luego relacionarlas con el tiempo que toma en ejecutarse.

1.1. Hipótesis

1. Se espera que Bubblesort sobre un arreglo aleatoriamente desordenado tome n° inversiones · largo del arreglo + operaciones swap. La cantidad esperada de inversiones es de orden N al igual que el largo del arreglo, luego esperamos que requiera aproximadamente N^2 comparaciones y un tiempo proporcional a eso.

2. Por otro lado, si el arreglo está casi ordenado, se espera que la cantidad de comparaciones disminuya dramáticamente hasta n° inversiones $\cdot N$ e incluso mejor.

3. Se espera que Insetionsort, que toma $O(n)$ comparaciones, en un arreglo aleatorio tome muchas menos, cerca de $\frac{3}{4}N^2$. Y que sea de $O(n + n^\circ inversiones)$ en el caso casi ordenado. Tomando en ambos casos un tiempo mucho menor que Bubblesort.

4. En el caso de Quicksort, esperamos que al seleccionar el pivote al azar, los casos aleatorio y casi ordenado no presenten grandes diferencias de performance, siendo ambos $O(nlog(n))$. Cabe decir que al seleccionar determinísticamente un pivote, provoca la aparición de peores casos, elevando la complejidad del algoritmo hasta $O(n^2)$.

5. Para mergesort, en cualquier caso se espera que para el caso aleatorio, la cantidad de comparaciones vaya con el caso promedio teórico $nlog(n) - n + O(log(n))$, pero en tiempo se espera que sea levemente mayor, esto por la cantidad extra de escrituras/lecturas a memoria que causan las copias de los sub arreglos en las llamadas recursivas.

1.2. Ambiente Operacional

Acá Claudin habla de su computador!!

2. Diseño de Algoritmos y Experimentos

Los cuatro algoritmos fueron implementados en sus formas clásicas y están disponibles en dos versiones, una que cuenta la cantidad de comparaciones que realiza y otra que no, para así medir el tiempo y la cantidad de comparaciones de forma independiente.

2.1. Bubble Sort

Bubblesort, recorre el arreglo, llevando cada vez a uno o más elementos hasta lo más alto que puedan llegar en el orden final. Repite el procedimiento hasta que logra recorrer el arreglo sin detectar ninguna inversión. A continuación, la implementación de este algoritmo:

```
static public int[] sort(int[] A){
        boolean isSorted = false;
        while (!isSorted){
            isSorted = true;
            for(int i=0; i<A.length-1; i++){
                if(A[i]>A[i+1]){
                    swap(A, i, i+1);
                    isSorted = false;}
            }
        }
        return A;
}
```

2.2. Insertion Sort

El ordenamiento por inserción mantiene siempre el siguiente invariante: al ordenar el elemento en la posición A_j el arreglo $A[1, j-1]$ ya está ordenado. Luego, debe buscar la posición de A_j hacia su izquierda. Acá la implementación:

```
static public int[] sort(int[] A){
        for(int i=1; i<A.length; i++){
            for(int j=i-1; j>=0 && A[i]<A[j]; j--){
                swap(A, i, j);
                i--;
            }
        }
        return A;
}
```

2.3. Merge Sort

El algoritmo de ordenamiento por mezcla, utiliza dividir para reinar, dividiendo el arreglo en dos, ordenando cada mitad y luego mezclando los resultados. La división del arreglo ocurre hasta llegar a sub arreglos de tamaño 1. A continuación se presenta el cuerpo del algoritmo, la función de mezcla puede encontrarse como anexo:

```
static public int[] sort(int[] A) {
        if(A.length == 1) return A;
        int mid = A.length/2;
        int[] A1 = Arrays.copyOfRange(A, 0, mid);
        int[] A2 = Arrays.copyOfRange(A, mid, A.length);
        return merge(sort(A1), sort(A2));
}
```

2.4. Quicksort

El algoritmo Quicksort, se basa en escojer un pivote al azar dentro del arreglo y particionarlo entre los elementos mayores y los menores a este pivote. Esto se repite recursivamente hasta que las particiones tienen tamaño 1. A continuación se presenta el esqueleto de la implementación, la función de particionamiento puede ser encontrada en los anexos:

```
static public int[] sort(int[] A){
        return quicksort(A, 0, A.length-1);
}

private static int[] quicksort(int[] A, int start, int end) {
        if(end-start<1) return A;
        int mid = partition(A,start, end);
        quicksort(A, start, mid);
        quicksort(A, mid+1, end);
        return A;
}
```

2.5. Diseño de los Experimentos

Los experimentos consistieron en someter a los algoritmos a ordenar arreglos sucesivamente más grandes (tamaño $2^i, i \in \{11, ..., 20\}$) y medir su tiempo (en nanosegundos) y cantidad de comparaciones promedio, usando las dos versiones de cada algoritmo, para no afectar la performance debido al conteo. Cada experimento se repite las veces necesarias para que el error sea menor al 5 % (i.e., hasta que $\frac{stdv}{mean} \leq 0{,}05$). Para detalles de la implementación de la batería de experimentos, referirse al código fuente adjunto.

3. Resultados

very successful

4. Conclusiones

wow

5. Anexos

5.1. Merge function

```
static private int[] merge(int[] A, int[] B){
        int i=0, j=0, n=A.length+B.length;
        int[] res = new int[n];

        for(int k=0; k<n; k++){
           if(i<A.length && j<B.length){
               if(A[i]<=B[j]){
                   res[k] = A[i];
                   i++;
               }else{
                   res[k] = B[j];
                   j++;
               }
           } else if(i == A.length){ //si se acaban los elems de B
               for(;k<n;){
                   res[k] = B[j];
                   k++; j++;
               }
           }else{
               for(;k<n;){ //si se acaban los elems de A
                   res[k] = A[i];
                   k++; i++;
               }
           }
        }
        return res;
    }
}
```

5.2. Partition function

```
private static int partition(int[] A, int start, int end) {
    int index = start + new Random().nextInt(end-start);
    int pivot = A[index];
    int i = start, j = end;
    while(i<j){
        while(A[i] < pivot) i++;
        while(A[j] > pivot) j--;
        swap(A, i, j);
    }
    return i;
}
```

5.3. Experimentos

5.3.1. Experimento 1

5.3.2. Experimento 2

5.3.3. Experimento 3

5.3.4. Experimento 4

5.4. Resultados

5.4.1. Resultado 1

5.4.2. Resultado 2

5.4.3. Resultado 3

5.4.4. Resultado 4