



FACULTAD DE INGENIERIA

Universidad de Buenos Aires

Teoría de los Algoritmos (75.29)

Reentrega Trabajo Practico Nro 1

2do Cuatrimestre 2016

Padrón	Apellido y Nombre	email
89563	Ferreya, Oscar	fferreyra38@gmail.com
90697	Eisner, Ariel	aeisnerg@gmail.com

Fecha de reentrega: 25/11/2016

Sumario

ESTADÍSTICOS.....	4
Modo Ejecución.....	4
Cálculo de Orden de los Algoritmos Solicitados.....	5
Fuerza Bruta.....	5
Orden.....	5
Complejidad.....	5
Mejores casos.....	5
Peores casos.....	5
Ordenar y Seleccionar.....	6
Orden.....	6
Complejidad.....	6
Mejor y peor caso.....	6
K-Selecciones.....	6
Orden.....	7
Complejidad.....	7
Mejor caso.....	7
Peores casos.....	7
K-HeapSort.....	7
Orden.....	8
Complejidad.....	8
Mejores casos.....	8
Peores casos.....	8
K-HeapSelect.....	8
Complejidad.....	9
Mejores casos.....	9
Peores casos.....	9
GRAFOS.....	10
BFS.....	10
Complejidad.....	10

Mejores casos.....	10
Peores casos.....	10
Dijkstra.....	10
Complejidad.....	10
Mejores casos.....	10
Peores casos.....	10
BFS Heurístico.....	11
Complejidad.....	11
Mejores casos.....	11
Peores casos.....	11
A*.....	11
Complejidad.....	11
Mejores casos.....	11
Peores casos.....	11

ESTADÍSTICOS

Modo Ejecución

El programa de inferencias estadísticas recibe 4 parámetros

Ej: datafake.txt 3 8 9

Nombre del Archivo de Datos

Texto plano con los números separados por una “,”

Numero de Método

- 1.FuerzaBruta
2. Ordenar y Seleccionar
3. K-Selecciones
- 4.K_HeapSort
- 5.HeapSelect
- 6.QuickSelect

La posición a consultar dentro del vector

El valor en si

Cálculo de Orden de los Algoritmos Solicitados

Fuerza Bruta

```
public boolean calcularPorFuerzaBruta(int k, int valor) {  
    int cantidadDatos = contenedorDatos.size(); //1  
  
    boolean seEncontroElemento = false; // 1  
  
    int indice = 0; // 1  
  
    while (!seEncontroElemento) { // K  
  
        int elementoActual = contenedorDatos.elementAt(indice); // 2  
  
        int menores = 0; // 1  
  
        for (int i = 0; i < cantidadDatos; i++) { // N  
            if (i != indice) {//para que no analice el actual // 1  
                int elementoAComparar = contenedorDatos.elementAt(i); // 2  
                //no se contemplan caso de valores repetidos  
                if (elementoAComparar < elementoActual) { // 1  
                    menores++; // 2  
                }  
            }  
        }  
        seEncontroElemento = ((menores + 1) == k); // si se encontro // 3  
        indice++; // 2  
    }  
    return (contenedorDatos.elementAt(indice-1) == valor); // 3  
}
```

Orden

El algoritmo de fuerza bruta recorre los elementos del arreglo, y a cada uno lo compara contra todos los demás para determinar la cantidad de elementos menores al considerado, en cada. Si encuentra alguno para el cual dicha cantidad sea k, corta el ciclo en ese momento y lo devuelve como resultado.

Complejidad

$O(n^2)$

Mejores casos

$O(n)$ si el elemento con k menores se encuentra en la primera

Peores casos

$O(n^2)$ si el elemento con k menores se encuentra en la última posición.

$$T(n) = 1 + 1 + 1 + \sum_{i=1}^K 2 + 1 \left[\sum_{i=1}^n 1 + 2 + 1 + 2 \right] + 3 + 2 = 3 + \sum_{i=1}^K 8 \left[\sum_{i=1}^n 6 \right]$$

$$T(n) = 3 + 8 K (6 N) = 3 + 48 K N =$$

$$\in O(K N) \text{ y si } K \text{ es el ultimo elemento } \in O(n^2)$$

Ordenar y Seleccionar

```
public boolean calcularPorOrdenarSeleccionar(Integer pos, Integer valor) {
    mergeSort(0, contenedorDatos.size()-1);          //N log(N)

    if (contenedorDatos.elementAt(pos-1) == valor) { // 3
        return true;                                // 1
    }
    return false;                                    // 1
}
```

Orden

Este algoritmo tiene dos partes. La primera, que ordena el conjunto por el metodo de mergesort y la segunda, que selecciona un elemento de ese conjunto. El orden del algoritmo dependerá completamente del orden del algoritmo de búsqueda.

Complejidad

$O(n \log n)$

Mejor y peor caso

$O(n \log n)$

$$T(n) = n \cdot \log_2(n) + 3 + 1 = n \cdot \log_2(n) + 4 \in O(n \cdot \log_2(n))$$

K-Selecciones

```
public boolean calcularPorKSelecciones(Integer pos, Integer valor){
    Integer minimo = Integer.MAX_VALUE;          //1

    Integer indiceMin = 0;                        //1

    for (int i = 0; i < pos; i++) {                //K
        for (int j = 0; j < contenedorDatos.size(); j++) { //n
            if(contenedorDatos.elementAt(j) < minimo){ //2
                minimo = contenedorDatos.elementAt(j); //2
                indiceMin = j;                          //1
            }
        }
        selecciones.set(i,minimo);                  //1
        contenedorDatos.removeElementAt(indiceMin); //1
        minimo = Integer.MAX_VALUE ; indiceMin = 0; //2
    }
    return verificarPosPorSeleccion(pos-1,valor); //1 + T(VerPos)
}

private boolean verificarPosPorSeleccion(Integer pos, Integer valor) {
    if(selecciones.elementAt(pos) == valor) //1
        return true;                        //1
}
```

```

        return false;    //1
    }

```

Orden

El ordenamiento por selección es $O(n^2)$ porque recorre todo el conjunto para hallar el menor, lo coloca en el primer lugar y vuelve a repetir el proceso para todos los elementos restantes. Dado que no se necesita ordenar todo el arreglo, sino solo encontrar el k-esimo elemento menor, deberemos recorrer el conjunto de tamaño n, k veces.

Complejidad

$O(n \cdot k)$

Mejor caso

$O(n)$ si k es uno

Peores casos

$O(n^2)$ si k es igual a n

$$T(n) = 2 + \sum_{i=1}^K \sum_{j=1}^n (2 + 2 + 1) + 1 + 1 + 2 + 1 + 2 = 5 + k(n + 5)$$

$kn + 5k \in O(kn)$ y si K es el ultimo elemento $\in O(n^2)$

K-HeapSort

```

public boolean calcularPorKSeleccionesEnHeap(Integer k, Integer valor){
    cargarDatosAlheapMinimo();    // nlog2(n)

    return buscarEnHeapMinimo(k,valor); // klog2(n)
}

private void cargarDatosAlheapMinimo() {
    for (int i = 0; i < contenedorDatos.size(); i++) { //n
        heapMinimo.agregar(contenedorDatos.elementAt(i)); // log2(n)
    }
}

private boolean buscarRefEnHeap(Integer k, Integer valor) {
    --k;    //1
    while(k > 0){    //K
        heap.eliminarMin();    //log2(n)
        --pos;    //1
    }
    if (heap.obtenerMin() == valor)    //2
        return true;    //1

    return false;    //1
}

```

Orden

Cargar los datos en el heap tiene una complejidad de $O(n \log n)$. El extraer un elemento de este obliga a realizar un downheap, el cual se realiza en un tiempo de $O(\log n)$. Si se realizan k extracciones, se deberá realizar un downheap k veces. Acceder al elemento en la raíz se realiza en orden constante. Ergo la complejidad del K-Heapsort será $O(n \log n + k \log n)$, el tiempo de la carga del heap mas el de la búsqueda en él mismo. Como k puede ser a lo sumo igual a n , el orden del algoritmo será como máximo

$O(n \log n + n \log n) \rightarrow O(2n \log n)$, es decir $O(n \log n)$.

Complejidad

$O(n \log n)$

Mejores casos

$O(n \log n)$ si k es igual a 1. Porque solo debo realizar el ordenamiento de heapsort y evito los downheap de cada extracción.

Peores casos

$O(2n \log n)$ si k es igual a n .

$$T(n) = \sum_{i=1}^n \log_2(n) + 1 + \sum_{i=1}^k [\log_2(n) + 1] + 2 + 1 =$$
$$T(n) = \sum_{i=1}^n \log_2(n) + \sum_{i=1}^k [\log_2(n) + 1] + 4$$

$$n \log_2(n) + k \log_2(n) \in O((k+n) \log_2(n))$$

K-HeapSelect

```
public boolean calcularPorHeapSelect(int k, int valor) {
    for (int i = 0; i < k; i++) { // K
        heapMaximo.agregar(contenedorDatos.elementAt(i)); //  $\log(K)$ 
    }

    for (int i = k; i < contenedorDatos.size(); i++) { // N - K
        if(heapMaximo.obtenerMax() > contenedorDatos.elementAt(i)){
            heapMaximo.eliminarMax(); //  $\log(K)$ 

            heapMaximo.agregar(contenedorDatos.elementAt(i)); //  $\log(K)$ 
        }
    }
    return heapMaximo.obtenerMax() == valor; // 2
}
```


HeapSelect usa un heap de maximo, pero este heap tiene un total de k elementos. Esto implica que las operaciones, como la de remover la raiz, seran $O(\log k)$. Para determinar los menores elementos se comparan todos contra el maximo del heap, el cual será el k -iesimo valor buscado. En caso de que alguno sea menor que el máximo, se lo agrega al heap, removiendo el máximo anterior. Esto implicar dos operaciones $O(\log k)$.

Complejidad

$O(n \log k)$

Mejores casos

$O(k \log k)$ si los k elementos menores están en las primeras posiciones del conjunto, lo que evita modificar y cargar el heap luego de su creación. En este caso, nunca se cumpliría el if del segundo bucle for.

Peores casos

$O(n \log k)$ si los k elementos menores están en las últimas posiciones del conjunto, lo que evita que implica que todos los elementos del conjunto serán agregados al heap y la $n - k$ serán removidos.

GRAFOS

BFS

Es el algoritmo de recorrido del grafo en anchura. Pasa por todos los vértices y aristas. Su complejidad es $O(|V| + |A|)$ siendo $|V|$ la cantidad de vértices del grafo y $|A|$ la cantidad de aristas.

Complejidad

$O(|V| + |A|)$

Mejores casos

- $O(1)$ si el origen es igual al destino.
- $O(|V_m| + |A_m|)$ siendo $|V_m|$ la cantidad de vértices pertenecientes al camino mínimo y $|A_m|$ la cantidad de aristas cuando el grafo es lineal.
- $O(|V_n| + |A_n|)$ con $|V_n|$ la cantidad de vértices en niveles menores o iguales a n y $|A_n|$ la cantidad de aristas correspondientes, si el primer camino evaluado es el camino mínimo. En el ejemplo se muestran dos caminos, pero solo hay uno mínimo (rojo) y será el primero que se considerará.

Peores casos

- $O(|V| + |A|)$ si el destino es el último vértice en ser evaluado por estar en el mayor nivel posible.
- Si el grafo es ponderado puede no escoger el camino mínimo.

Dijkstra

Se implementa el algoritmo utilizando como estructura para ir almacenando los vértices aún no recorridos una cola de prioridad, de modo que la complejidad estará afectada por el costo de las operaciones de dicha estructura, que son $O(n \log n)$.

En cada iteración se elimina un vértice de la cola ($O(|V| \log |V|)$), se analiza si sus adyacentes están visitados y en caso contrario se los agrega a la cola ($O(|A| \log |V|)$). Es decir, se realizan $O((|A| + |V|) \log |V|)$. Esto es equivalente a $O(|A| \log |V|)$.

Complejidad

$O(|A| \log |V|)$

Mejores casos

- $O(|V| \log |V|)$ si cada vértice encolado es único por iteración, y dicho vértice se encola una sola vez y luego queda agregado al camino.

Peores casos

- $O(|A| \log |V|)$ si los vértices se encolan y desencolan varias veces.

BFS Heurístico

En vez de encolarse directamente el nodo adyacente más próximo a lo ancho, sucesivamente, lo que se hace es agregarlo según un análisis de la adyacencia y una distancia ponderada del destino, utilizando una heurística. En este caso, se usó como heurística el cálculo de la distancia Manhattan entre dos vértices. Debido al uso de la cola de prioridad en el cual se agregan solo una vez todos los vértices, la complejidad será $O(|V| \log |V|)$

Complejidad

$O(|V| \log |V|)$

Mejores casos

- $O(|V| \log |V|)$ si el grafo es cuadrado y no ponderado.

Peores casos

- $O(|A| \log |V|)$ si la heurística no es buena y estima distancias erróneas. Puede ocurrir en grafos que no tengan una forma adecuada, lo que puede hacer necesario recorrer todo el grafo para encontrar un nodo que puede haberse encontrado antes.
- En el caso de grafos ponderados, no se toma en cuenta el peso de las aristas, por lo que no se puede esperar el camino mínimo, sino solo uno de los existentes.

A*

Se implementa el mismo algoritmo que para BFS Heurístico, pero teniendo en cuenta los pesos de las aristas.

Complejidad

$O(|V| \log |V|)$

Mejores casos

- $O(|V| \log |V|)$ si el grafo es cuadrado y la heurística es buena, es decir no hay gran diferencia entre la distancia estimada y la real.

Peores casos

- $O(|A| \log |V|)$ si la heurística no es buena y estima distancias erróneas.