Algorítmica: práctica 2

Mezclando k vectores ordenados

Grupo 2

Sofía Almeida Bruno Antonio Coín Castro María Victoria Granados Pozo
Miguel Lentisco Ballesteros José María Martín Luque

6 de abril de 2017

Introducción

El objetivo de esta práctica es diseñar un algoritmo *divide y vencerás* que se encargue de combinar *k* vectores ordenados. Además tenemos que analizar su eficiencia, implementarlo y compararlo con un algoritmo clásico.

Algoritmo clásico

```
/**
 * EJERCICIO 4: mezcla de k vectores ordenados
 *
 * Generador de ejemplo para el problema de mezcla de k vectores ordenados. Para obtener vectores
 * ordenados de forma creciente, cada uno con n elementos, se genera un vector de tamaño k*n
 * con todos los enteros entre 0 y k*n-1 ordenados.
 *
 * Se lanzan entonces k iteraciones de un algoritmo
 * de muestreo aleatorio de tamaño n para obtener los k vectores. Están ordeados porque el
 * algoritmo de muestreo mantiene el orden.
 *
 */

#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

using namespace std;

#define PRINT_VECTOR 0

/**
 * Genera un número uniformemente distribuido en el
 * intervalo [0,1)
 */
double uniforme() {
  int t = rand();
  double f = ((double) RAND_MAX + 1.0);
  return (double) t/f;
}

/**
 * Imprime el contenido de k vectores
 */
void imprimir_vector(int* T, int n) {
  for (int i = 0; i < n; i++) {
    cout << T[i] << " ";
  }
  cout << " " << endl;
}

/**
 * Mezcla dos vectores ordenados en un tercero
```

```
 */
void merge(int T1[], int T2[], int S[], int n1, int n2) {
  int p1 = 0, p2 = 0, p3 = 0;

  while (p1 < n1 && p2 < n2) {
    if (T1[p1] <= T2[p2]) {
      S[p3] = T1[p1];
      p1++;
    }
    else {
      S[p3] = T2[p2];
      p2++;
    }

    p3++;
  }

  while (p1 < n1) {
    S[p3++] = T1[p1++];
  }

  while (p2 < n2) {
    S[p3++] = T2[p2++];
  }
}

/**
 * Mezcla k vectores ordenados en un vector solución
 */
int* mezcla_vectores(int** T, int k, int n) {
  int* S = new int[k*n];  // Vector mezcla
  assert(S);

  if (k > 1) {
    int* aux = new int [k*n];
    assert(aux);

    // Primera mezcla
    merge(T[0], T[1], S, n, n);

    // Resto de mezclas
    for (int i = 2; i < k; i++) {
      merge(S, T[i], aux, i*n, n);
      swap(S, aux);   // Intercambiamos punteros
    }

    delete [] aux;
  }

  else {
    for (int i = 0; i < n; i++) {
      S[i] = T[0][i];
    }
  }

  return S;
```

```
}

int main(int argc, char * argv[]) {
  if (argc != 3) {
    cerr << "Formato " << argv[0] << ": <num_elem>" << " <num_vect>" << endl;
    return -1;
  }

  int n = atoi(argv[1]);  // número de elementos
  int k = atoi(argv[2]);  // número de vectores

  int** T;
  T = new int* [k];
  assert(T);

  for (int i = 0; i < k; i++)
    T[i]= new int [n];

  int N = k*n;
  int* aux = new int [N];
  assert(aux);

  // Genera todos los enteros entre 0 y k*n-1
  srand(time(0));
  for (int j = 0; j < N; j++)
    aux[j] = j;

  // Para cada uno de los k vectores se lanza el algoritmo S (sampling) de Knuth
  for (int i = 0; i < k; i++) {
    int t = 0;
    int m = 0;

    while (m < n) {
      double u = uniforme();

      if ((N - t) * u >= (n - m)) {
        t++;
      }

      else {
        T[i][m] = aux[t];
        t++;
        m++;
      }
    }
  }

  delete [] aux;

  #if PRINT_VECTOR
    // Imprimir los k vectores
    for (int i = 0; i < k; i++) {
      cout << "Vector " << i + 1 << ": ";
      imprimir_vector(T[i], n);
    }
  #endif
```

```
158
159   int* S;
160   clock_t t_antes = clock();
161   // Mezclamos los k vectores por el algoritmo clásico
162   S = mezcla_vectores(T, k, n);
163   clock_t t_despues = clock();
164   cout << k << " " << ((double)(t_despues - t_antes)) / CLOCKS_PER_SEC << endl;
165
166   #if PRINT_VECTOR
167     cout << endl << "Vector mezcla: ";
168     imprimir_vector(S, k*n);
169   #endif
170
171   delete [] S;
172
173   for (int i = 0; i < k; i++)
174     delete [] T[i];
175
176   delete [] T;
177
178   return 0;
179 }
```

Eficiencia teórica

Eficiencia empírica

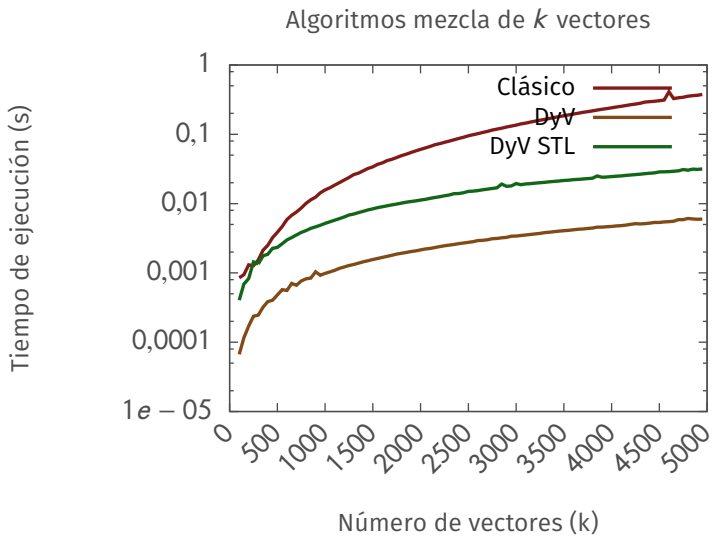
Eficiencia *híbrida*

Algoritmo *divide y vencerás*

Eficiencia teórica

Eficiencia empírica

Eficiencia *híbrida*



Algoritmos de ordenación

Eficiencia $O(n^2)$

Burbuja

Revisa cada elemento de la lista con el siguiente, intercambiándose de posición si no están en el orden correcto. Su eficiencia teórica es $O(n^2)$.

Inserción

Consideramos el elemento N-ésimo de la lista y lo ordenamos respecto de los elementos desde el primero hasta el N-1-ésimo. Su eficiencia teórica es $O(n^2)$.

Selección

Consiste en encontrar el menor de todos los elementos de la lista e intercambiarlo con el de la primera posición. Luego con el segundo, y así sucesivamente hasta ordenarlo todo. De nuevo, su eficiencia teórica es $O(n^2)$.

Eficiencia $O(n log n)$

Mergesort

Se basa en la técnica de divide y vencerás. Consiste en dividir la lista en sublistas de la mitad de tamaño, ordenando cada una de ellas de forma recursiva. Si el tamaño de la lista es 0 ó 1 la lista ya está ordenada. Para acabar juntamos todas las sublistas en una sola. Su eficiencia teórica es $O(n \log n)$.
En este caso, podemos observar ciertas anomalías en la gráfica, pues presenta unos *picos* para ciertos tamaños del vector. No sabemos bien a qué puede ser debido, pues hemos realizado cuidadosamente la ejecución en varias máquinas, así como en distintos sistemas operativos, obteniendo siempre el mismo resultado.

Quicksort

También se basa en la técnica de divide y vencerás. En primer lugar elegimos un elemento de la lista, que llamaremos *pivote*. A continuación los elementos de la lista se ordenarán de forma que la derecha del pivote queden los mayores que él, y a la izquierda los menores. De esta forma dividimos la lista en dos sublistas, la de la derecha y la de la izquierda. Repetiremos el proceso mientras las sublistas tengan más de un elemento. Su eficiencia teórica también es $O(n \log n)$.

Heapsort

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en una estructura de datos llamada montículo (*heap*). Luego, se extrae el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él. Su eficiencia teórica es $O(n \log n)$.

Otros algoritmos

Floyd

Es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos ponderados. El algoritmo compara todos los posibles caminos a través del grafo entre cada par de vértices. Es un ejemplo de **programación dinámica**, y su eficiencia teórica es $O(n^3)$.

Hanoi

Hay tres pilas de discos, llamadas origen, auxiliar y destino. La primera de ellas está ordenada según tamaño creciente de los discos, de arriba hacia abajo. Se moverá un disco de la pila origen a la destino si hay un único disco en la pila origen. En caso contrario, se moverán todos los discos a la auxiliar, excepto el más grande. Por último, moveremos el disco mayor al destino, y movemos los $n - 1$ restantes encima del mayor. El número de pasos crece exponencialmente con el número de discos, y su eficiencia teórica es $O(2^n)$.

Cálculo de la eficiencia empírica

Puesto que la eficiencia teórica de cada algoritmo es diferente, no podemos realizar las mediciones para los mismos valores de entrada en todos los algoritmos. Así, los agrupamos según su orden de eficiencia.

Para el cálculo de la eficiencia empírica, hemos utilizado un *script* que realiza tantas ejecuciones de cada algoritmo como le indiquemos, tomando como parámetros el valor inicial, el incremento, y el valor final de los datos de entrada.

Las gráficas anteriores han sido realizadas a partir de los datos recogidos en las siguientes tablas.

Tablas

Gráficos comparativos

A continuación se muestra una serie de gráficos que corresponden a la comparación de aquellos algoritmos que son de un mismo orden.

Algoritmos que son $O(n^2)$

Observamos que el algoritmo **burbuja** es claramente más lento para estos valores. Esto se verá reflejado cuando calculemos los valores de las constantes ocultas en la eficiencia híbrida.

Algoritmos que son $O(n \log n)$

Estos algoritmos se ejecutan en tiempos similares, aunque **quicksort** es ligeramente más rápido, manteniendo de forma uniforme la diferencia con los otros dos algoritmos a medida que aumenta el número de valores.

Algoritmos de ordenación

En este último gráfico, puede observarse claramente la tendencia de los algoritmos cuyo orden de eficiencia es $O(n \log n)$ a ser más rápidos que aquellos que son $O(n^2)$. Se ha utilizado una escala logarítmica para poder representar todos los algoritmos en un mismo gráfico.

Cálculo de la eficiencia híbrida

A continuación se recogen los gráficos que muestran tanto la eficiencia empírica como la función ajustada o *eficiencia híbrida* de cada algoritmo. Para el ajuste, se ha utilizado la función que corresponde en cada caso a la eficiencia teórica de cada algoritmo. Además, utilizamos el mayor número de constantes ocultas posibles, a fin de que el ajuste sea mejor.

El valor de estas constantes ocultas puede verse reflejado también en las gráficas. Son precisamente estas constantes las que hacen que un algoritmo se ejecute más rápidamente que otro, aunque tengan el mismo orden de eficiencia teórica.

Eficiencia $O(n^2)$

Para ajustar los algoritmos de ordenación cuadráticos hemos usado una función $f(x)$ de la forma:

$$f(x) = a_0 x^2 + a_1 x + a_2$$

Con ayuda de `gnuplot` hemos calculado los valores de las constantes ocultas, así como la gráfica que muestra tanto el ajuste como los valores experimentales.

Eficiencia $O(nlogn)$

La función utilizada para ajustar los valores en los algoritmos de ordenación de orden superlineal es:

$$f(x) = a_0 x log(x) + a1x + a2$$

Eficiencia $O(n^3)$

En el caso del algoritmo de Floyd la función $f(x)$ utilizada es:

$$f(x) = a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

.

Eficiencia $O(2^n)$

Por último, ajustamos el algoritmo de Hanoi mediante la función:

$$f(x) = a_0 2^{a_1 x + a_2}$$

obteniendo el resultado mostrado a continuación.
En general, las constantes ocultas tienen un valor muy pequeño, en algunos casos del orden de 10^{-9}. Esto es porque el ordenador ejecuta los algoritmos en cuestión de segundos o milésimas de segundo, para los tamaños de entrada relativamente “pequeños” que hemos introducido.

Comparativa según optimización y sistema operativo

Hemos elegido un representante de cada orden de eficiencia, y hemos realizados una comprativa para analizar cómo varían los tiempos de ejecución según el sistema operativo y el nivel de optimización.

Para cada algoritmo seleccionado veremos dos gráficas: una que compara la ejecución sin optimización en los sistemas operativos *macOS* y *Linux*; y otra que compara la ejecución en *Linux* con los distintos tipos de optimización del compilador.

Representante de $O(n^2)$: Inserción

Representante de $O(n \log n)$: Quicksort

Representante de $O(n^3)$: Floyd

Representante de $O(2^n)$: Hanoi

Podemos observar en los cuatro casos que las variaciones entre ejecuciones en distintos sistemas operativos es prácticamente inapreciable, salvo alguna excepción. Del mismo modo, los tiempos de ejecución disminuyen significativamente, en general, de manera proporcional al nivel de optimización, tal y como cabía esperar.

Ajuste con otras funciones

Veamos por último lo que ocurre si intentamos ajustar los datos recogidos experimentalmente a una función que no coincide con la eficiencia teórica del algoritmo.
Observamos que la curva ajustada y los datos experimentales no coinciden apenas que en unos cuantos puntos. Esto era de esperar, pues hemos ajustado los datos con funciones que difieren notablemente de su eficiencia teórica.

Anexo

Características de los ordenadores donde se ha ejecutado

1. Apple MacBook Pro, Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz, 8GB RAM.
   Compilador: clang-800.0.38
   Sistema operativo: macOS Sierra

2. Dell XPS 13, Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 8GB RAM.
   Compilador: g++ 6.3.1
   Sistema operativo: Arch Linux