Concurso EDA 2010-2011

Carlos Escriche Izquierdo

11 de mayo de 2011

Índice

1. Introducción

La siguiente memoria del concurso está planteada a modo de histórico. La memoria está dividida en subproblemas que hacen referencia a una parte en concreto del problema general. Para cada subproblema se iŕan contando las optimizaciones realizadas en orden cronológico, así como alguna solución que haya aumentado el coste temporal. Los subproblemas serán los siguientes:

- **Lectura**: Hace referencia al modo de leer los píxeles.
- **Patches y cálculo de repetidos**: Se trata del cálculo de los patches y de cuantas veces aparecen en la imagen.
- **Distancia entre imágenes**: Aquí se habla de la distancia entre imágenes antes de insertar en el grafo.
- **Grafo y Dijkstra**: Se muestra la inserción de las k-menores distancias en el grafo y del algoritmo de Dijkstra utilizado para conseguir el camino de menor coste.

2. Lectura

2.1. Modo normal. Pixel a pixel.

La primera forma de leer es abriendo el fichero con el modo por defecto. Los píxeles ya quedaban con el formato adecuado y es muy sencillo de implementar. Es más lento que si se lee en binario.

2.2. Modo binario. Buffer.

La siguiente manera de leer es mucho más rápida, pero después de leer hay que dar formato a los píxeles. Primero se abre el fichero en modo binario y se guarda toda la imagenen un vector o buffer de caracteres y después se convierten dichos caracteres a enteros. Mejora bastante el coste temporal. En el oráculo mejoraba alrededor de 10 segundos.

3. Patches y cálculo de repetidos

3.1. Calculo de patches

3.1.1. Calcular los patches de manera secuencial leyendo los píxeles necesarios para cada patch

Es tal y como se explica en el enunciado del problema.

$$d(I,J) = \sum_{q \epsilon D_{IJ}} N_{Iq} + \sum_{q \epsilon D_{JI}} N_{Jq} + \sum_{q \epsilon C_{IJ}} abs(N_{Iq} - N_{Jq}) \quad (1)$$

Para cada patch se recorre $w * w$ pixeles y se calcula la suma de las componentes RGB. Al principio la implementación era regular y funcionaba bastante lento. Posteriormente, con una implementación mejor, haciéndolo de la misma manera, se reducía bastante el tiempo.

3.1.2. Leer píxeles secuencialmente y sumarlos a los patches que corresponda

Para cada pixel se calculan los patches a los que pertenece dicho pixel. Teóricamente debería funcionar mejor porque no se accede varias veces a un píxel. Calcular los patches a los que pertenece un píxel no es trivial y requería de muchos cálculos. Esta implementación aumentaba unos segundos el tiempo de la anterior, por lo que fue un paso atrás.

3.2. Representación de las sumas de las componentes RGB

Al principio, para representar en cada patch las sumas de las componentes RGB, utilizaba 3 variables. Esto aumentaba el coste temporal de todo lo demás. Porque cuando comparaba patches tenía que comparar las tres variables. Posteriormente esas 3 componentes se redujeron a una sola mediante desplazamientos. Aunque solo se puede hacer después de saber que los valores w y c son los adecuados. Convirtiendo las 3 variables en 1 una, se puede mejorar casi todo lo demás, pero pierde la funcionalidad cuando w es grande y c es pequeña.

3.3. Elementos repetidos

3.3.1. Repetidos ordenando

Después de calcular los patches hay que contar y eliminar los repetidos. Se cuenta el número de patches con la misma clave representando los elementos repetidos como uno sólo y con una cuenta. Esto se puede implementar con un algoritmo de ordenación. Primero se ordena el vector de repetidos por la clave. Consecutivamente, se recorre el vector y mientras la clave no sea distinta, se suma el número de apariciones. El algoritmo de ordenación utilizado es el Sort.

3.3.2. Repetidos con tabla hash

En vez de utilizar un vector y luego ordenarlo, se pueden insertar los patches en una tabla hash a medida que se calculan. Antes de insertar se busca en la tabla, si no está se inserta y se marca su cuenta a 1. Si ya está insertado, se suma 1 a su cuenta. Los elementos únicos quedan en la tabla hash con su respectiva cuenta. Está implementado con una hash map.

4. Distancia entre imágenes

El cálculo de las distancias admite muchas soluciones. Estas son algunas de las implementaciones que he realizado.

4.1. Algoritmos anteriores

Antes de utilizar merge para el cálculo de distancias, ultilicé muchos otros. Aquí no los nombro porque ninguno de ellos era eficiente ni pasaba el concurso.

4.2. Merge

Para poder utilizar estas versiones basadas en merge hay que ordenar antes los vectores de patches por la clave, lo que incrementa un poco el coste. Los elementos comunes que se nombran después son los patches que pertenecen a dos imágenes. Los elementos diferentes pertenecen a una imagen pero no a la otra.

4.2.1. Comunes y no comunes

Es el primer algoritmo de cálculo de distancias con el que pasé el concurso. Esta versión utiliza para el cálculo de la distancia tanto los elementos comunes como los no comunes y siempre incrementa la distancia. Utiliza la fórmula tal cual está en el enunciado del problema. La idea es igual que es el merge de MergeSort. Si la clave de uno es menor que la del otro se suma la cuenta de la clave menor. Cuando las claves son iguales se suma el valor absoluto de la resta de las cuentas. Al terminar uno de los dos vectores hay que sumar a la distancia, la cuenta de todos los elementos sobrantes del otro vector.

4.2.2. Solo comunes

Esta modificación se debe a que los elementos comunes son muchos menos que los no comunes. La modificación respecto al anterior es en la suma de los no comunes que se elimina y también se quita el último bucle que

recorre los elementos que quedan en un vector al terminar el otro. Mejora significativamente. En el concurso eran 15 o 20 segundos.

4.3. Intersección con AND lógica

Otro algoritmo abordado es el de la intersección de vectores mediante AND lógica. Primero insertaba todos los elementos en una tabla hash. Si el elementos ya estaba insertado, ya no lo insertaba. De esta forma conseguía un vector con todo el rango de parches. Por cada imagen creaba un vector de bits con la talla igual al rango de parches. Y marcaba con un 1 si el patch estaba en la imagen y con 0 si no estaba. Para calcular la distancia entre dos imágenes hacía una AND con los dos vectores de bits. Finalmente recorría el vector resultado de la AND y donde hubiera unos (elementos comunes) calculaba la distancia. El coste era bastante más lento que los elementos comunes. Hubiera sido muy rápido si el rango de elementos fuera pequeño, pero era grande.

4.4. Tabla hash perfecta

Con esta idea acaba el cálculo de distancias. Es la más óptima de las aquí propuestas para el caso particular del concurso y sus datos. Calculando la clave como un solo valor que cabe en un entero se pueden dispersar todos los patches posibles en un vector de manera que no haya colisiones. Esto funciona bien porque con los datos del concurso se puede crear la clave con una sola variable. Y además este vector no sale demasiado grande. El tamaño es aproximadamente 1 millón elementos para los datos del concurso. El bucle general recorre todas las imágenes. Por cada patch no repedido de la imagen i-ésima se mira si en la cubeta dada por la clave del patch hay patches ya insertados de otras imágenes. Si los hay, por cada elemento que haya en el std vector se calcula una distancia auxiliar de la siguiente manera: Lo que sumarían los patches si no fueran comunes menos lo que ganan comunes. Es decir, lo que dejan de ganar por ser comunes.

$cuentaI = cuentapatchqueseinserta$
$cuentaSTD = cuentapatchenelstdvector$

$$distanciaAux = cuentaI + cuentaSTD - abs(cuentaI - cuentaSTD) \quad (2)$$

Esta distancia auxiliar se resta a la distancia de la imagen del patch del vector std a la imagen del i-ésima. Cuando concluye el bucle general se debe replicar la distancia en la triangular inferior y sumar a cada distancia el número total de patches de las dos imágenes que intervienen. La mejora de tiempo era abrumadora. De 50 segundos a 11.

5. Grafo y Dijkstra

5.1. Grafo representado como lista de adyacencia

El grafo utilizado está representado mediante una lista de adyacencia debido a que es muy disperso y de esta manera se reduce el coste espacial y más tarde el temporal con el algoritmo de Dijkstra.

5.2. Inserción de las k-menores distancias

Para insertar las distancias en grafo se buscan las k-menores distancias a cada imagen mediante selección.

5.3. Dijkstra básico

Algoritmo de Dijkstra implementado de la manera vista en clase. Con vector de visitados. Como el grafo es disperso este algoritmo no es tan bueno como el implementado con cola de prioridad. El coste asintótico $\mathcal{O}(|V|^2)$.

5.4. Dijkstra Cola de prioridad

El algoritmo Dijkstra con cola de prioridad mejoraba aproximadamente 10 segundos respecto al Dijkstra simple. Está implementado mediante una cola de prioridad de la STL priority queue. El coste asintótico es $\mathcal{O}(|A|log|V|)$.