Estructuras de Datos y Algoritmos

Facultad de Informática
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Práctica 2

Algoritmo de Kruskal

Parte (b):

Implementación de un Montículo o "Heap"

Curso 2003-2004

1. Introducción

En este boletín continuamos con el desarrollo del algoritmo de Kruskal que calcula el Árbol de Expansión de mínimo peso (MST) para un grafo dado. El algoritmo de Kruskal se basa, entre otras cosas, en considerar la lista de aristas del grafo en orden creciente de peso. Esto puede hacerse fácilmente mediante una ordenación (*sort*) de las aristas por su peso, como veremos en el siguiente boletín de prácticas. Otra alternativa, preferible en muchos casos, es estructurar las aristas como un *montículo de mínimos* (*min-heap*) organizado por pesos.

Un vector (de punteros a aristas) puede estructurarse como un montículo con un coste *lineal* ($O(a)$) con el número de elementos (aristas). Una vez estructurado, se puede extraer el mínimo (y eliminarlo, manteniendo la estructura de montículo) con coste $O(\log a)$. Como el número de aristas que forman parte de un MST es frecuentemente mucho menor que que el número total de aristas del grafo dado, esta solución es preferible en muchos casos.

Este boletín describe las actividades a realizar (en dos sesiones de prácticas) para implementar y estudiar la estructura de datos *montículo* y comprobar su correcto funcionamiento y eficiencia. En particular, se trata de implementar un *min-heap* de aristas, con las operaciones de *construir heap* y *extraer el mínimo*,

2. Montículo de mínimos o *min-heap*

Un *árbol parcialmente ordenado* (de menor a mayor) es un árbol binario casi-completo, en el que el valor de cualquier nodo es o menor o igual que el de sus nodos hijos. A partir de esta definición se derivan las siguientes propiedades:

- Todas las ramas del árbol son secuencias ordenadas.
- La raíz del árbol es el nodo de mínimo valor.
- Todo subárbol es, a su vez, un árbol parcialmente ordenado.

Una de las implementaciones mas eficientes de árbol parcialmente ordenado, y quizás la mas simple, es la representación vectorial, también llamada *(min-)heap*. Según esta representación, si A es un vector con n elementos, $A[1]$ es la raíz del árbol y, dado un nodo $A[i]$,

- si $2i \leq n$, $A[2i]$ es su hijo izquierdo
- si $2i + 1 \leq n$, $A[2i + 1]$ es su hijo derecho
- si $i \neq 1$, $A[\lfloor i/2 \rfloor]$ es su padre

Con esta estructura, un vector cualquiera con n elementos se puede convertir en *min-heap* "in situ" y con un coste temporal $O(n)$. Denominaremos *buildMinHeap* a la operación que hace esta conversión. Por otra parte, dado un vector estructurado como *min-heap* se puede saber cuál es el mínimo elemento del vector en $O(1)$: basta consultar $A[1]$. Para la práctica totalidad de las aplicaciones de un *min-heap*, siempre que se consulta el mínimo también interesa eliminarlo. Esta eliminación no es tan simple, aunque también puede realizarse muy eficientemente, con un coste temporal en el peor de los casos de $O(\log n)$. Denominaremos *extractMin* a la operación que combina consulta y eliminación del mínimo. Ambas operaciones, *buildMinHeap* y *extractMin*, se basan a su vez en una operación "interna", generalmente denominada *heapify*, que consigue restaurar la estructura de *min-heap* cuando ésta falla localmente entre un nodo y sus dos hijos, pero ambos hijos tienen estructura de *min-heaps*.

Aunque innecesaria para sus uso en el algoritmo de Kruskal, otra operación eficiente en un vector estructurado como *min-heap* es la *inserción* de un elemento, que puede realizarse también en $O(\log n)$. Pueden encontrarse mas detalles sobre esta interesante estructura de datos en la mayoría de libros de algoritmos y estructuras de datos como, por ejemplo [1].

Para su uso en el algoritmo de Kruskal, se propone implementar una *clase minHeap* de punteros a aristas con los siguientes atributos y métodos:

```
class minHeap {         // min-heap de elementos de tipo pArista
  int  maxNelem;                        // máxima talla de elHeap
  int  nElem;     // n.elem. de elHeap estructurados en min-heap
  pArista *elHeap;      // copia, decrementada en 1, del puntero
                           // (vector) a estructurar en min-heap
public:
  minHeap(pArista *h, int numElem); // constructor: elHeap = h-1
  ~minHeap();                                        //  destructor
  int  numElem();
  void buildMinHeap(); // convierte el vector elHeap en min-heap
  pArista extractMinHeap();        // extrae y elimina el mínimo
};
```

Este esquema permite aprovechar la interesante propiedad de la operación *buildMinHeap* según la cual un vector cualquiera se se estructura *"in situ"*, sin necesidad de copiar o duplicar su contenido. Para ello se propone que el atributo privado `elHeap` sea un simple puntero donde se mantendrá la dirección del primer elemento del vector h suministrado al constructor de la clase `minHeap` en su inicialización. De esta manera, el programa que use la clase `minHeap` deberá invocar al constructor `minHeap(h,n)` con el vector `h` (de punteros a arista en nuestro caso) que se quiera manejar como heap. Obviamente, este programa será responsable de reservar o asignar la memoria del vector `h`. El código de implementación de `minHeap(h,n)` deberá copiar `h` en `elHeap` (y también hacer `maxNelem = nElem = n`) para que el resto de código de la clase pueda trabajar directamente con `elHeap` dejando los resultados realmente en `h`.

Por otra parte, conviene recordar que la estructuración implícita de un vector en árbol binario (basada en productos y cocientes de los índices por 2) exige que el primer elemento útil del vector, que coincide con el nodo raíz, sea el de índice 1, y no 0 como generalmente se asume en el lenguaje C o C++. Una solución simple es despreciar la posición 0, pero ésto no parece una buena idea ya que exigiría que todos los programas que usen la clase *minHeap* lo tengan en cuenta, con el consiguiente riesgo de errores. Una alternativa preferible consiste en decrementar en 1 la *dirección* de `h` al copiarla a `elHeap`; es decir, hacer `elHeap = h-1`.

3. Actividades de laboratorio

La actividades que se proponen a continuación tienen como objetivo producir las rutinas necesarias para convertir un vector de punteros a aristas en un *min-heap* estructurado según los pesos de las aristas representadas:

1. Implementar la clase *minHeap* de punteros a arista con los métodos *buildMinHeap* y *extractMin*.
2. Escribir un programa principal de prueba de la clase *minHeap*. Este programa leerá un grafo (lista de aristas) por la entrada estándar, haciendo uso del método *leer* de la clase *grafo* desarrollada en sesiones anteriores. A continuación, mediante el método *buildMinHeap*, deberá estructurar el vector de aristas en montículo de mínimos. Finalmente, mediante sucesivas llamadas a *extractMin* debe escribir por la salida estándar la lista de aristas, ordenadas por pesos crecientes. Para obtener los nombres alfanuméricos de los nodos de cada arista puede utilizarse el método *index2key* de la clase *hashTable*.
3. Modificar el programa anterior para obtener el tiempo que necesita el programa para realizar las operaciones *buildMinHeap* y (todas las llamadas a) *extractMin* con grafos de distintas tallas. Ver detalles sobre cálculo de tiempos en el apéndice A.

4. Cuestiones

1. Modifica la estructura de datos para trabajar con montículos de máximos en lugar de montículos de mínimos.
2. Escribe un método que permita insertar elementos en un heap.
3. Escribe un método que permita cambiar el peso de un elemento existente en un heap, manteniendo la estructura de montículo.
4. Usando el concepto de *plantilla* (“*template*”) de C++, modifica la clase *minHeap* desarrollada anteriormente para que trabaje con cualquier tipo de datos (no necesariamente punteros a arista).
5. Escribe un programa principal para probar la clase *genérica* implementada en el punto anterior, instanciando el tipo de base a enteros (`int`) y a reales (`double`).

Referencias

[1] T. Cormen, Ch. Leiserson, R. Rivest, “Introduction to Algorithms”, MIT, 1990.

Anexo A

Determinación de tiempos de ejecución: primitiva `clock`

Medir el tiempo de ejecución de una rutina es una forma empírica de determinar la calidad de una implementación del algoritmo correspondiente. En el lenguaje C++ se dispone de la librería entandar `ctime` que corresponde a la librería `time.h` del lenguaje C. Esta librería nos ofrece algunos tipos de datos y funciones:

`clock_t`	Tipo similar a un `long`
`clock_t clock()`	Retorna un número de tics de reloj
`CLOCKS_PER_SEC`	Constante para transformar tics a segundos

Con ellos podemos medir el tiempo de ejecución entre dos llamadas a la función `clock()`. Este tiempo no se corresponde con tiempo realmente transcurrido, sino más bien es el tiempo de procesador consumido por el proceso que invoca `clock`. De esta manera las mediciones resultan bastante independientes de la carga que tenga el sistema debida a otros procesos. Por ejemplo veamos el siguiente código, donde `{B}` es cualquier bloque de instrucciones cuyo tiempo de ejecución se desea determinar:

```
t1 = clock();
{B}
t2 = clock();
```

Después de la ejecución de este fragmento de programa el tiempo de proceso consumido por el bloque {B} será `t2-t1` (en tics).

La precisión de *clock* es limitada. En sistemas Unix/Linux estándar un "*tic*" equivale a 0.01 segundos (100 tics/s). Dada la gran potencia computacional que en la actualidad tienen la mayoría de procesadores, esta precisión es realmente *muy* limitada. Por ejemplo, en un modesto procesador PentiumIII a 450Mhz, una buena implementación del algoritmo *Quicksort* es capaz de ordenar 200 vectores de 10,000 enteros de 16 bits en 1 segundo; es decir el tiempo necesario para realizar una ordenación es de unos 0.005 segundos. Si quisiéramos medir este tiempo mediante `clock` obtendríamos 0 *tics*[1].

Aunque existen otros métodos de medición que permiten obtener resoluciones por debajo del microsegundo, de momento, por simplicidad, trataremos de arreglárnoslas con la función *clock*.

Obviamente, dadas las limitaciones indicadas, si queremos medir el tiempo de proceso de una instancia deberemos recurrir a medir el tiempo de procesar muchas instancias y dividir este tiempo por el número de instancias procesadas. Esto plantea el problema de decidir *cuántas son las instancias necesarias para obtener una precisión aceptable*. En el ejemplo anterior, si realizamos 200 ordenaciones obtendremos una medición de tiempo de aproximadamente 1 segundo; es decir, 100 *tics*. Esto resulta en un tiempo por instancia de $1/200 = 0{,}005$ segundos, con una resolución de aproximadamente el 1 %.

En general, el número adecuado de instancias a procesar depende de lo que tarde en procesarse una instancia. Por ejemplo, con el mismo procesador e implementación del *Quicksort* anteriores, el tiempo de ordenar un vector de 1,000,000 de enteros de 16 bits es de 1 segundo aproximadamente. En este caso, el tiempo por instancia se obtiene directamente con una precisión del 1 %, procesando una sola instancia. Si decidiéramos procesar en este caso el mismo número de instancias que en el caso de talla 10,000, obtendríamos una resolución del 0.01 % (mucho mas fina de lo necesario), pero el proceso de medición tardaría muchísimo más: unos 200 segundos para ser mas precisos.

Por todas estas razones, el método ideal para medir tiempos con la función *clock* consiste en *fijar un tiempo total* (mínimo) *de proceso*, tt, y determinar cuántas instancias completas pueden procesarse en ese tiempo (o algo más). Sea i el número de instancias procesadas y $t \geq tt$ el tiempo realmente requerido. El tiempo medio por instancia será $ti = t/i$ y la resolución será proporcional a t (concretamente $100 * 1/(t \cdot CLOCKS\_PER\_SEC)$ %, usualmente $(1/t)$ %).

[1] En realidad se obtendrá aleatoriamente 0 o 1 *tics*, dependiendo del instante exacto en el que se invoque a *clock*