Estructuras de Datos y Algoritmos

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

Práctica 2

Algoritmo de Kruskal

Parte (a):

Introducción, Tablas de dispersión y lectura/representación de grafos

Curso 2003-2004

1. Introducción

Son muchos los problemas de interés en las ciencias y las ingenierías que pueden ser planteados en términos de problemas sobre *grafos*. Un grafo es un conjunto de *nodos* o *vértices* y un conjunto de *aristas* o *arcos* que "conectan" estos vértices entre sí. En muchos casos de interés, las aristas suelen estar *ponderadas*; es decir, cada arista tiene asignado un valor que de alguna manera mide la importancia de esa arista en el grafo o en el problema que éste modela. Formalmente un grafo ponderado se denota mediante una tupla $G = (V, A, p)$, donde V es el conjunto de nodos, $A \subseteq V \times V$ es el conjunto de aristas (pares de nodos) y p es una función definida en A tal que $\forall (u, v) \in A$, $p(u, v)$ es el peso de la arista (u, v).

Entre los problemas sobre grafos, uno de los de mayor interés por su enorme abanico de aplicaciones es la obtención del *Árbol de recubrimiento de peso mínimo*. En lo que sigue nos referiremos a éste árbol por las siglas de su nombre en inglés, MST (*Minimum Spanning Tree*.

La determinación del MST tiene innumerables aplicaciones en diversas ramas de la ciencia y la tecnología. Por ejemplo, este tipo de árboles pueden usarse directamente como base para optimizar el diseño de redes de transmisión de energía o de información cuando la topología de las conexiones debe ser (o conviene que sea) arborescente. En otras áreas, el MST puede usarse para determinar *agrupamientos naturales* de datos. Si tenemos una colección de datos y alguna manera de medir la disimilitud o "distancia" entre ellos, podemos ver esta colección como el conjunto de nodos de un grafo y las disimilitudes entre objetos como aristas de un grafo completamente interconectado. El MST de un grafo de este tipo tiene la propiedad de que si se eliminan sus aristas de mayor peso (disimilitud) se obtiene un "bosque" en el que cada (sub-)árbol *agrupa* todos los datos similares o "próximos" entre sí.

Entre los algoritmos para obtener el MST de un grafo, uno de los mas interesantes es el algoritmo de Kruskal. Este algoritmo se basa en la simple idea "voraz" de construir incrementalmente un

bosque, seleccionando en cada paso una arista que *no cree ningún ciclo* y que produzca el *menor incremento de peso posible* en ese paso.

Aunque el desarrollo directo de esta idea conduce programas muy poco eficientes, mediante el uso de estructuras de datos adecuadas se logra un algoritmo muy rápido, capaz de procesar eficientemente grafos de gran talla. Este aspecto hace que el algoritmo de Kruskal sea de gran interés en la asignatura de Estructuras de Datos y Algoritmos.

El objetivo final de esta serie de prácticas es realizar implementaciones eficientes del algoritmo de Kruskal y comparar experimentalmente las prestaciones de estas implementaciones. Para ello convendrá comenzar con la implementación y comprobación de los distintas componentes del algoritmo; cada una de ellos relacionada con una estructura de datos y algoritmos correspondientes.

Aplicación propuesta

Como ejemplo de aplicación del MST y del algoritmo de Kruskal se propone obtener trabajar sobre grafos en los que los nodos son ciudades y las aristas son distancias entre ellas. Con este planteamiento se pueden resolver problemas tales como obtener una red arborescente óptima de interconexión entre un conjunto de ciudades dado, o agrupar este conjunto en subconjuntos de ciudades próximas entre sí.

Para cada ciudad se dispone de las siguientes informaciones:

- País al que pertenece la ciudad
- Nombre
- Latitud geográfica
- Longitud geográfica
- Altura sobre el nivel del mar

Para una correcta implementación del algoritmo de Kruskal (así como cualquier otro algoritmo sobre grafos) se requiere que los nodos queden identificados por números enteros entre 0 y $n-1$, donde n es el numero de nodos. Sin embargo, en nuestros datos, los nodos (ciudades) vienen dados por sus nombres, que son cadenas de caracteres ASCII de 8 bits. Por esta razón, el primer paso para la realización de esta práctica consiste en implementar un método adecuado para asociar cadenas (nombres de ciudades) con números enteros (nodos). Una buena solución a este problema nos la ofrecen las tablas de dispersión (*hashing*).

A partir de la latitud y longitud (y altura, si conviene) se puede calcular la distancia (esférica) entre dos ciudades. En base a este cálculo, y restringiéndonos a las ciudades de unos pocos países se construirán una serie de grafos en los que se conectarán entre sí distintos tipos de ciudades. Estos grafos estarán disponibles en sendos ficheros de texto en los que cada línea corresponde a una arista, con el formato:

`<nombre-ciudad-1>|<nombre-ciudad-2>|distancia`

Estos grafos son los que utilizaremos para las pruebas y experimentos con nuestras implementaciones de los distintos componentes del algoritmo de Kruskal, y del algoritmo finalmente construido.

2. Planteamiento formal y desarrollo del algoritmo de Kruskal

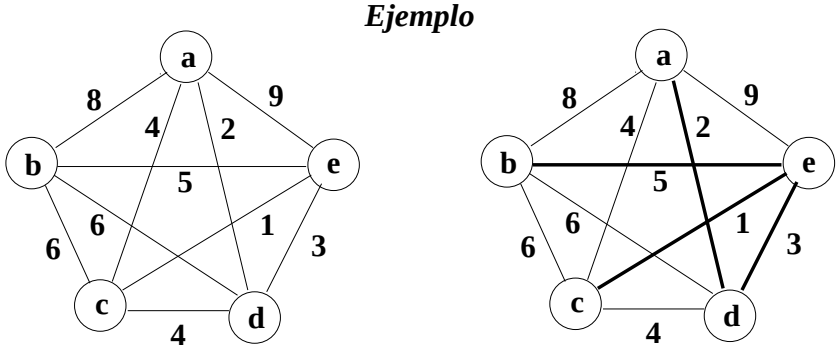
En esta sección desarrollaremos el algoritmo de Kruskal mediante refinamientos de una simple idea básica. De esta manera surgen dos versiones: una basada en pre-ordenación (*sort*) de las aristas del grafo y otra basada en la estructuración de estas aristas como un montículo (*heap*). En ambos casos aparece la necesidad de realizar eficientemente operaciones de búsqueda y unión en subconjuntos disjuntos. Para ésto, se adopta la estructura de datos por excelencia para este proceso: el MFSET.

2.1. Árbol de Recubrimiento de peso mínimo

Árbol de Recubrimiento de un grafo *no dirigido* $G = (V, A, p)$ es un árbol libre $T = (V'A')$ tal que $V' = V$ y $A' \subseteq A$.

Problema: Dado un Grafo Conexo Ponderado No Dirigido $G = (V, A, p)$, encontrar un Árbol de Recubrimiento de G, tal que la suma de los pesos de sus $|V| - 1$ aristas sea mínimo.

Ejemplo



Grafo Ponderado No Dirigido Árbol de Recubrimiento de peso mínimo (= 11)

Solución basada en una estrategia voraz: construir incrementalmente un *bosque*, seleccionando en cada paso una arista $(u, v) \in A$ tal que:

- *No* cree ningún ciclo,
- Produzca el menor incremento de peso posible.

El resultado se obtiene cuando se han seleccionado exactamente $|V| - 1$ aristas válidas, que es la condición de se ha de cumplir para que el bosque sea un grafo conexo (y, por tanto, un árbol) con exactamente $|V|$ nodos.

En el apéndice A se muestra una traza de ejecución de un algoritmo basado en esta idea.

2.2. Esquema básico del algoritmo de Kruskal

Dado un grafo $G = (V, A, p)$ dirigido y ponderado por $p : A \rightarrow \mathbb{R}^{\geq 0}$, ($p(u, v) = +\infty \;\; si \; (u, v) \notin A$), la función KRUSKAL obtiene un árbol $T = (V, A')$ cuyo peso total D es mínimo.

función KRUSKAL(G=(V,A,p)) **{**

  $A'' = A;\; A' = \phi;\; D = 0;$

  mientras $(|A'| < |V| - 1)$ **{**

    $(u, v) = argmin_{(x,y)\in A''}\, p(x, y)$

    $A'' = A'' - \{(u, v)\}$

    si ((u, v) no crea un ciclo en (V, A')**)**

     {$A' = A' \cup \{(u, v)\}$**;** $D = D + p(u, v)$ **}**

  }

  retornar(T,D)

}

Una implementación trivial de este algoritmo calcularía el *argmin* a base de recorrer en cada paso todas las aristas disponibles y seleccionar una cuyo peso sea mínimo. Por otra parte, determinar si (u, v) *crea un ciclo en* (V, A') requeriría, por ejemplo, recorrer todos los vértices del árbol parcial al

que pertenece u para ver si entre ellos se encuentra v. Estas operaciones tendrían un coste temporal de $O(n^4)$ o superior, donde $n = |V|$. Incluso utilizando computadores de los mas rápidos disponibles actualmente, esta implementación resultaría inservible en la práctica para grafos con un número bastante pequeño de nodos (aproximadamente 500 o más).

En cualquier caso, está claro que el coste de la implementación Kruskal está dominado por la operación $argmin$ y a la comprobación de "*no crear ciclo*". A continuación estudiaremos como realizar eficientemente estas operaciones.

3. Estructuras de datos para el algoritmo de Kruskal

Para conseguir una versión eficiente del algoritmo esbozado en la sección anterior hay dos problemas a resolver:

- Cómo verificar eficientemente la condición de "*no crear ciclo*"
- Cómo realizar rápidamente a operación de selección (*argmin*)

Una buena solución al primer problema consiste en mantener una colección de subconjuntos (disjuntos) con los vértices incluidos en cada árbol parcial: *Una arista* (u, v) *no creará ciclo si* u *y* v *están en distintos subconjuntos*. Existe una estructura de datos ideal para mantener y consultar con gran eficiencia *colecciones de subconjuntos disjuntos*. Esta estructura, conocida generalmente con el nombre de *MFSET* (*Merge/Find Set*), consigue tiempos prácticamente unitarios (1 paso) para las dos operaciones necesarias: buscar (*find*) el subconjunto (árbol parcial) al que pertenece un elemento (nodo) y unir (*merge)* dos subconjuntos (árboles parciales) disjuntos.

Por otra parte, la operación de selección (*argmin*) puede realizarse eficientemente construyendo una *lista ordenada* por peso con las aristas de A''. Una alternativa interesante y en la práctica preferible al *sort* consiste en mantener A'' como un *montículo* (*Min-Heap*) de aristas organizadas según su peso. Esta estructura permite montar un montículo con un coste *lineal* ($O(a)$) con el número de elementos (aristas) y extraer el mínimo del conjunto con coste $O(\log a)$, donde a es el número de elementos (aristas).

3.1. Algoritmo de Kruskal mediante ordenación de aristas

función KRUSKAL-S(G=(V,A,p)) **{**

$Q = \text{minSort}(p, A);\ S = \text{BuildMFSET}(V);\ A' = \phi;\ D = 0$

mientras ($|A'| < |V| - 1$**) {**

$(q, (u, v)) =$ **cabeza**(Q)**;** *// leer el mínimo de la cola Q*

descabezar(Q**)** *// eliminar el mínimo de Q*

$x =$**FIND**(u)**;** $y =$**FIND**(v)

si ($x \neq y$**) {** **MergeSet**$(x, y);\ A' = A' \cup (u, v);\ D = D + q$ **}**

}

retornar(T,D) *//* $T = (V, A')$

}

El coste de este algoritmo está dominado por el coste de la ordenación. Si *minSort* se implementa de forma que su coste sea $O(n \log n)$, donde n es la talla de la lista a ordenar, el coste será $O(a \log a)$, donde $a = |A|$.

3.2. Algoritmo de Kruskal mediante Heap de aristas

función KRUSKAL-H(G=(V,A,p)) **{**

$H = \text{BuildHeap}(p, A);\ S = \text{BuildMFSET}(V);\ A' = \phi;\ D = 0$

mientras ($|A'| < |V| - 1$**) {**

$(q, (u, v)) =$ **extraeMinHeap**(H) *// leer y eliminar el mínimo*

$x =$**FIND**(u)**;** $y =$**FIND**(v)

si ($x \neq y$**) {** **MergeSet**$(x, y);\ A' = A' \cup (u, v);\ D = D + q$ **}**

}

retornar(T,D) $// T = (V, A')$

}

Es fácil ver que el coste de este algoritmo en el peor de los casos también se puede acotar por $O(a \log a)$, donde $a = |A|$. Sin embargo, en este caso el coste real es generalmente bastante inferior a dicha cota.

Si m es el número de iteraciones del bucle *mientras*, el coste real será $O(a + m \log a)$. Como $a = O(n^2)$ ($n = |V|$) y $O(\log n^2) = O(\log n)$, el coste es $O(a + m \log n)$. En grafos densos $a \gg n$, pero el bucle principal suele terminar mucho antes de haber visitado todas las aristas; típicamente con $m \approx n$. Por tanto el coste está mas cercano a $O(a)$ que a $O(a \log a)$. Si a es grande, la ventaja con respecto al coste de la versión basada en *sort* es clara. En grafos muy poco densos $m \approx n \approx a$ y el coste es $O(a \log a)$. En estos casos, la versión basada en *sort* puede llegar a ser más rápida.

4. Planificación de las actividades de laboratorio

Para lograr una correcta compresión del algoritmo de Kruskal y para facilitar su implementación paso a paso, se propone una subdivisión del trabajo a realizar en las siguientes sesiones:

- Grafos, lectura/escritura, tablas de dispersión (*hash*): 4 sesiones
- Montículos (*heaps*): 2 sesiones
- Ordenación (*sort*): 2 sesiones
- Subconjutos Disjuntos (*MFSETs*): 2 sesiones
- Kruskal, medición de prestaciones: 4 sesiones.

La primera práctica se destinará a familiarizarse con el problema a resolver e implementar las operaciones de lectura, representación y escritura de los grafos de (nombres de) ciudades. Dado que las ciudades vienen dadas por sus nombres (cadenas de caracteres), para representar las aristas en memoria será necesario implementar una asociación entre nombres y números enteros. Como se ha comentado anteriormente, ésto se puede realizar adecuadamente mediante *hashing*.

En la segunda práctica se procederá a implementar un montículo (*heap*) de aristas, con las operaciones de *construir heap* y *extraer el mínimo*, y a comprobar su correcto funcionamiento y eficiencia.

La tercera práctica se destinará a realizar una implementación y estudio empírico de algún algoritmo clásico de ordenación tal como, por ejemplo, *quicksort*.

En la cuarta práctica se realizará una implementación de la estructura *MFSET* con sus operaciones de buscar (*find*) y unir (*merge*). Esta estructura tiene por sí misma gran interés y es útil en muchas aplicaciones. En este caso la usaremos únicamente para soportar las colecciones disjuntas de nodos de los distintos árboles parciales que se van obteniendo en las distintas iteraciones del algoritmo de Kruskal.

En la quinta y última parte de esta serie de prácticas utilizaremos todo el código implementado en las sesiones anteriores para *montar* dos versiones del algoritmo de Kruskal; una basada en ordenación y otra en montículo. También se deberá implementar un *programa principal* suficientemente flexible para comprobar el correcto funcionamiento del algoritmo implementado y para medir sus prestaciones sobre grafos de gran talla; típicamente con tallas hasta algunas decenas de millones de aristas.

Los detalles de cada una de estas partes se describirán en sendos boletines que aparecerán sucesivamente a continuación del presente.

5. Tablas de dispersión: introducción

Para desarrollar el algoritmo de Kruskal es necesario implementar eficientemente las operaciones de lectura, representación y escritura de los grafos de (nombres de) ciudades. Dado que las ciudades vienen dadas por sus nombres, para representar las aristas en memoria será necesario implementar una asociación entre nombres y números enteros. Ésto se puede realizar adecuadamente mediante tablas de dispersión. El objetivo de esta práctica es implementar las funciones de lectura y escritura de grafos, lo que implícitamente supone la implementación de tablas de dispersión. En primer lugar nos centraremos en las tablas de dispersión y después implementaremos las funciones sobre grafos.

Los objetivos concretos de esta parte son los siguientes:

- realizar un estudio comparativo de diferentes funciones de dispersión para cadenas de caracteres,
- estudiar experimentalmente la influencia del tamaño de la tabla y la función de dispersión en las prestaciones alcanzables con distintos tipos de texto.

En la siguiente sección se comentan brevemente las características más importantes de las tablas de dispersión y en posteriormente se explica cuál es el trabajo a realizar en la práctica. Finalmente se plantean una serie de cuestiones referentes al trabajo realizado en el laboratorio, así como actividades de ampliación del mismo.

6. Tablas de dispersión

Una tabla de dispersión es una estructura de datos apropiada para representar un conjunto de elementos cuando las operaciones a optimizar son *insertar*, *eliminar* y comprobar si un elemento *pertenece* al conjunto, esto es, para representar un *diccionario*.

En una tabla de dispersión, los elementos se distribuyen en un conjunto de m cubetas utilizando para ello una clave y una *función de dispersión*. Cuando varios elementos se dispersan en la misma cubeta decimos que se produce una *colisión*. Existen diversas propuestas para tratar el problema de las colisiones: una posibilidad es utilizar una representación con *direccionamiento abierto* (ver referencias), mientras que otra posibilidad es utilizar una representación con *encadenamiento separado*. Esta última consiste básicamente en un vector donde cada componente apunta a una lista enlazada que está asociada a un determinada cubeta. La representación con encadenamiento es la propuesta que vamos a estudiar en esta práctica.

Un punto importante es la elección del número de cubetas, m, según la talla, N, del conjunto de datos que se espera almacenar en la tabla. Obviamente conviene que el *factor de carga*, $\alpha = N/m$, sea pequeño. Como regla general, para ello basta utilizar valores grandes de m, próximos a N. No obstante, esta regla general puede fallar si la *función de dispersión* es inadecuada.

Una buena *función de dispersión* debe distribuir los datos entre las diferentes cubetas *equitativamente*. Una visión cualitativa de esta distribución nos la da el *histograma de ocupación de cubetas*. Un histograma muy concentrado alrededor (de un valor cercano) del *factor de carga* es señal de una buena distribución, mientras que un histograma disperso significa una falta de equilibrio en la

carga. Una medida escalar de la calidad de un histograma de este tipo es la *desviación típica de la ocupación de las cubetas*. Un valor grande de desviación típica ocurre cuando la dispersión es poco uniforme, mientras que un valor pequeño corresponde a un buen equilibrio en la carga de las cubetas disponibles.

En resumen, el número de cubetas debe decidirse básicamente en función de la talla del conjunto a almacenar, procurando el menor valor posible del factor de carga, mientras que la función de dispersión debe elegirse en función del tipo de datos a cargar en la tabla, procurando obtener desviaciones típicas pequeñas.

Finalmente, el parámetro decisivo para el diseño definitivo es el tiempo medio requerido por operación. Se espera que este tiempo disminuya con:

- el factor de carga,
- la desviación típica de la ocupación de cubetas,
- el tiempo requerido para el cálculo de la función de dispersión.

Para reducir el factor de carga basta aumentar el tamaño de la tabla. La optimización relativa a los dos últimos ítem es mas conflictiva, ya que éstos suelen ser contrapuestos: una buena función de dispersión generalmente requiere un mayor tiempo de cálculo. Por tanto, en la práctica, hay que llegar a ciertos compromisos en la elección o diseño de la función de dispersión.

Funciones de dispersión

Una función simple

Una función de dispersión simple es la siguiente:

```
unsigned int func1(char *cadena) {
  unsigned int h=0;
  for (char* p=cadena; *p!='\0'; p++)
    h += (unsigned int)(*p);
  return(h % m); // m representa el número de cubetas
}
```

Esta función se basa en el método de la división [2]. En primer lugar la clave alfabética se transforma en una clave numérica sumando los códigos ASCII de cada carácter y, a continuación, se acota el resultado a un rango entre 0 y $m-1$ mediante el cálculo del resto de la división entera por m (operación *módulo*). Se caracteriza porque hace uso de todos los caracteres de la clave alfabética para realizar la transformación, pero no es capaz de aprovechar las ventajas potenciales de usar un número elevado de cubetas.

Método divisivo

La función:

```
unsigned int func2(char * x) {            // -- método 'de la división'
#define firstChar 32   // Código del primer caràcter a considerar (' ')
#define rangChar 227     // Número de caracteres a considerar (' '-'ÿ')
#define base rangChar

  unsigned int h=0;
  for (char* p=x; (*p)!='\0'; p++)
    h = (h*base + (unsigned int)(*p)-firstChar) % m;
  return(h);
}
```

también se basa en el método de la división. En este caso, la clave alfabética se transforma en una clave numérica considerando que los caracteres son como "dígitos" en una base adecuada, y el resultado se acota módulo m. Como la base adecuada es grande (menor pero aproximadamente igual a 256 si se esperan letras mayúsculas y minúsculas, posiblemente acentuadas), para longitudes de cadena usuales el resultado de la transformación suele ser un número astronómicamente grande. Aunque la operación resto lo acotaría adecuadamente, la transformación inicial no puede realizarse con aritmética de precisión limitada. Afortunadamente, la operación resto puede calcularse iterativamente, limitando así el rango de valores a manejar y evitando que se produzca desbordamientos aritméticos

Método multiplicativo

La siguiente función:

```
unsigned int func3(char *cadena) {          // Método  multiplicativo
   // cte_hash= 2^32/proporcio_aurea, proporcio_aurea = (sqrt(5)+1)/2
  const unsigned int cte_hash = 2654435769U;

  unsigned int h = 0;
  for (char* p=cadena; *p!='\0'; p++)
    h = (h + (unsigned int)(*p)) * cte_hash;
  return(h % m);
}
```

se basa en el método de la multiplicación [2], también calculado iterativamente como en el caso anterior, pero tiene la ventaja de no requerir una base explícita para la transformación. Además la operación resto se calcula "implícitamente" en cada iteración con suma eficiencia por desbordamiento de los 32 bits con que se representa un número entero. Como la clave numérica obtenida de esta transformación suele ser un entero muy grande ($\leq 2^{32}$), se requiere una operación módulo final para acotarla entre 0 y $m-1$. La conveniencia de utilizar la constante que aparece en la función se discute en [1].

7. Actividades de laboratorio sobre tablas de dispersión

En primer lugar se propone estudiar dos aspectos importantes de una tabla de dispersión: la elección del tamaño de la tabla y la elección de una función de dispersión. Los datos a almacenar en las tablas de dispersión a estudiar serán cadenas de caracteres (nombres de ciudades).

En `/labos/asignaturas/FI/eda/` se encuentran los datos necesarios para realizar la práctica. Los fichero de datos pueden contener valores repetidos, por lo que si utilizan para representar conjuntos sólo de almacenarse una repetición de cada cadena.

7.1. Implementación de tablas de dispersión

El primer paso es implementar una clase para la tabla de dispersión con la estrategia de resolución de colisiones conocida como "encadenamiento separado". Las operaciones que debe soportar la clase son la inserción de un elemento y la búsqueda. Ambas operaciones se pueden fundir en una única función. La operación de borrado no será necesaria.

Puesto que esta tabla de dispersión se utilizará más adelante para realizar un *mapeo* entre cadenas de caracteres y enteros, es conveniente que se tome esto en consideración para la implementación. Una vez implementada la clase, se propone implementar un programa de prueba que haga uso de esa clase. Este programa debe proporcionar un histograma de ocupación de cubetas así como la media y la desviación típica de la ocupación de las cubetas. Esto permitirá estudiar las características de la tabla de dispersión tal como se describe a continuación.

7.2. Influencia de la función de dispersión

Se propone usar el programa implementado para estudiar las propiedades de las tres funciones de dispersión descritas. Para cada función hay que obtener el correspondiente histograma de ocupación de cubetas para diferentes números de cubetas. Los histogramas obtenidos pueden almacenarse en un archivo y visualizarse mediante el programa de representación gráfica `gnuplot`. Si "`histo.gnp`" es el nombre de este archivo, una directiva de `gnuplot` adecuada para obtener una gráfica de estilo histograma es: "`plot "histo.gnp" with boxes`".

Para tablas grandes, se puede observar como la función mas simple, `func1`, produce cargas extremadamente desequilibradas, con un número enorme de cubetas vacías. La explicación de este comportamiento hay que buscarla en el limitado rango de valores de la función de dispersión que `func1` puede producir (estudia `func1` hasta que este comportamiento te resulte evidente). Aún a pesar de este desequilibrio, los tiempos que se observan no son tan malos; el bajo coste computacional de esta simple función llega a compensar bastante bien el coste extra de recorrer las largas listas asociadas a las pocas cubetas ocupadas. El comportamiento estadístico de `func2` y `func3` es muy similar, pero solo `func3` (cuyo coste computacional es significativamente menor), llega a superar en eficacia temporal a la función mas simple y de peor comportamiento estadístico `func1`.

7.3. Influencia del tamaño de la tabla

Usando el programa implementado, se propone realizar sendas gráficas en las que se represente la eficacia de la tabla de dispersión en función del número de cubetas, para las tres funciones de dispersión estudiadas en la sección anterior. Se debe observar como el coste temporal por operación decrece monótonamente con el número de cubetas, m, hasta aproximadamente $m = N$, donde N es la talla del conjunto almacenado en la tabla. A partir de este punto, ulteriores incrementos de m ya no producen beneficios significativos.

Además de las gráficas de coste temporal, también tiene interés representar los parámetros estadísticos (la desviación típica en particular), en función del número de cubetas.

8. Cuestiones sobre tablas de dispersión

1. Modifica la estructura de datos para que se lleve un registro del número de elementos insertados en cada una de las cubetas.
2. ¿Qué supondría tener una función de dispersión para la cual la talla de las cubetas tuviera una media $\mu > 0$ y una varianza 0?
3. El buen funcionamiento de una función de dispersión es siempre independiente del tipo y distribución de los datos que se dispersan. ¿Crees que esta afirmación es cierta? ¿Por qué?
4. ¿Qué sucede con la función de dispersión `func1` si todas las cadenas no pasan de longitud 3, el código ASCII más grande es 256 y el número de cubetas elegido es mucho más grande de 768?
5. ¿Qué sucede con la función `func1` con las cadenas *sirena* y *resina*?
6. Modifica el programa principal de la práctica para que una vez construida la tabla de dispersión con un fichero, elimine las palabras leídas de otro fichero que coincidan con alguna línea almacenada en la tabla; esto es, se trata de calcular la diferencia entre los conjuntos de cadenas contenidas en ambos ficheros.

9. Lectura y representación de aristas: introducción

Un primer paso para la implementación del algoritmo de Kruskal es la lectura y representación de las aristas del grafo a procesar. Tal como se discutió en la sección 1 (página 2), los grafos que uti-

lizaremos para las pruebas y experimentos vendrán dados en ficheros en los que cada línea contendrá una arista con el formato:

```
<nombre-ciudad-1>|<nombre-ciudad-2>|distancia
```

Sin embargo, para su representación y manejo en base a las estructuras de datos necesarias, se requiere que los nodos del grafo sean números naturales. Por tanto, en el proceso de lectura del grafo, se necesita establecer una relación biunívoca entre nombres de ciudades y números enteros. Para esto podemos utilizar una tabla de dispersión, adecuadamente modificada para mantener un identificador entero asociado a cada vértice diferente leído como parte de alguna arista. Esto significa que deberemos disponer de una función que tome como entrada una cadena (una ciudad) y la busque en la tabla. Si se encuentra, deberá devolver su identificador entero asociado; sino, la incluirá en la tabla, asignándole un nuevo entero con el que quedará identificada en posteriores llamadas.

Adicionalmente, para el proceso de escritura es conveniente disponer de una función tal que, dado un identificador entero, devuelva la ciudad asociada. De esta manera es posible escribir los nodos de un grafo sin más que conocer sus identificadores. Nótese que hay que modificar la tabla de dispersión para que soporte y mantenga esta información adicional de manera eficiente.

Concretamente, conviene implementar una clase con los siguientes atributos y métodos:

```
class hashTable{
  ... // atributos y métodos privados necesarios para la
      // tabla de dispersión.
public:
  hashTable (int s);
  ~hashTable ();
  int keyToIndex (char *newKey);
  char *indexToKey (int i);
};
```

Esta clase facilitará la implementación de las rutinas de lectura y representación de aristas que discutiremos a continuación.

10. Lectura y representación de aristas

Conceptualmente, una arista es un par (u, v), donde u y v son los nodos involucrados. Podemos denotar una *arista ponderada* como $((u, v), w)$, donde (u, v) es una arista y w es su peso.

Mediante la tabla de dispersión discutida anteriormente al leer una arista podemos obtener un identificador numérico para cada uno de los dos nodos leídos. Así pues, una arista la podemos representar en memoria como un *registro*:

```
typedef struct arista {int u; int v; float w;};
```

Un grafo nos vendrá dado como un fichero en el que la primera línea contiene dos valores enteros que representan, respectivamente, el número de nodos y el número de aristas. A continuación cada línea contiene una arista ponderada. Una adecuada representación en memoria de la colección de aristas debe cumplir los siguientes requisitos:

- facilitar la implementación de las rutinas de ordenación (*Sort*) y *Heap*
- evitar el movimiento innecesario de datos (aristas) en memoria

Esto se puede conseguir fácilmente manteniendo una estructura estática con todas las aristas leídas y un *vector de punteros a arista* que es el que manejarán las rutinas de *Sort/Heap*. Mas concretamente, podemos declarar un tipo puntero a arista y luego un vector de aristas y un vector de punteros a estas aristas:

```
typedef arista* pArista; // puntero a arista
arista* todasLasAristas;
pArista* pa;
```

Durante la lectura, tras obtener los identificadores de los nodos de la arista i-ésima, ésta se almacenará (secuencialmente) en `todasLasAristas[i]` y en el vector de punteros `pa` pondremos un puntero al registro `todasLasAristas[i]`. Por tanto podemos definir una *clase* `grafo` del siguiente modo:

```
   class grafo {
     ... // atributos y métodos privados necesarios para representar
         // el grafo
   public:
     grafo(int maxAristas);        // constructor
     ~grafo();                     // destructor
     pArista* leer(char* fichEnt); // asigna memoria para un vector de
                                   // p. a arista y lo rellena de
                                   // punteros a las aristas leidas
     escribir(char* fichSal);      // escribe el grafo en un fichero
     escribirAristas(char* fichSal, pArista* vectorParistas, int nA);
                                   // escribe en un fichero el vector de
                                   // punteros a aristas
     ...
};
```

Una construcción alternativa consiste en declarar las aristas como una *clase* `arista` en lugar de un *registro* (`struct`). Esto tiene la ventaja de facilitar la sobrecarga de los operadores de comparación $<, \leq, =$, de manera que podamos comparar dos aristas según el valor de su peso w. Estas comparaciones se necesitarán posteriormente para las rutinas de *Sort* y *Heap*.

11. Actividades de laboratorio sobre lectura y representación de aristas

La actividades que se proponen a continuación tienen como objetivo que al finalizar las mismas el alumno tenga la rutinas necesarias para la lectura y escritura de grafos:

1. modifica adecuadamente la clase de la tabla de dispersión tal como se ha descrito anteriormente, para que establezca una correspondencia biunívoca (“*mapping” uno a uno*) entre cadenas de caracteres y enteros;

2. escribe una clase para la lectura y escritura de grafos eligiendo una de las dos maneras de representar aristas que se han descrito;

3. escribe un rutina principal de prueba de la clase de lectura escritura, cuyas únicas acciones sean leer un grafo de la entrada estándar y escribirlo por la salida estándar. Prueba el programa escrito con los fichero de prueba que encontrarás en `/labos/asignaturas/FI/eda/`.

Referencias

[1] D.E. Knuth “Sorting and Searching”, vol. 3, “The Art of Computer Programming”, Addison-Wesley, 1973.

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

[3] B.J. McKenzie, R. Harries, T. Bell “Selecting a Hashing Algorithm”, Software–Practice and Experience, Vol. 20(2), pp. 209–224, 1990. (Demanar al professor de pràctiques)

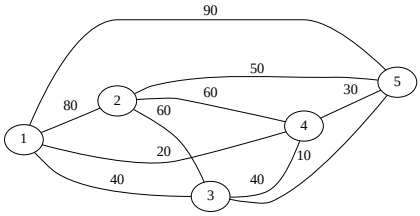
[4] G.H. Gonnet, R. Baeza-Yates, “Handbook of algorithms and data structures: in Pascal and C”, Addison-Wesley, 2nd ed., 1991.

[5] M.A. Weiss, “Estructuras de datos y algoritmos”, Addison-Wesley, 1995.

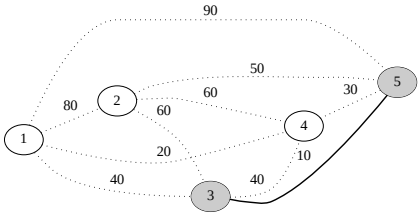
[6] G.L. Heileman “Data structures, algorithms and object-oriented programming”, McGraw-Hill, 1996.

A. Ejemplo de ejecución del algoritmo de Kruskal

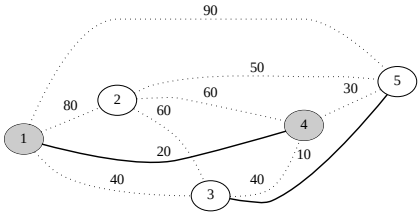
Las lineas gruesas representan las aristas sucesivamente seleccionadas y las lineas discontinuas (en los pasos 4 y 5) corresponden a aristas que se descartan por no cumplir la condición de “no crear ciclo”.



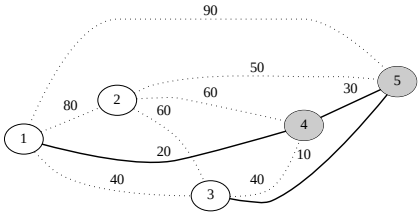
Grafo Inicial



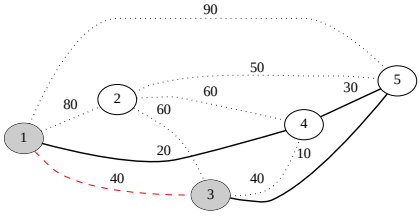
Traza Kruskal: (u,v) = (3,5); peso=10; find(3) != find(5)



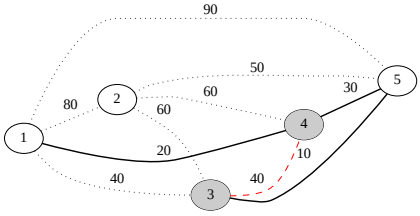
Traza Kruskal: (u,v) = (1,4); peso=20; find(1) != find(4)



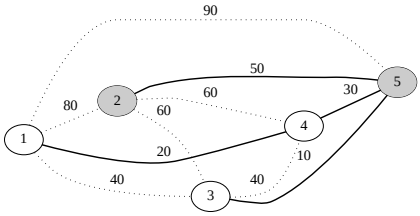
Traza Kruskal: (u,v) = (4,5); peso=30; find(4) != find(5)



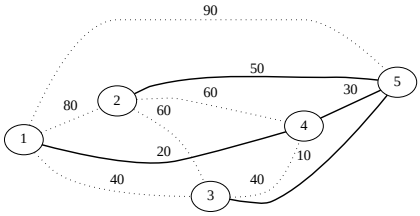
Traza Kruskal: (u,v) = (1,3); peso=40; find(1) == find(3)



Traza Kruskal: (u,v) = (3,4); peso=40; find(3) == find(4)



Traza Kruskal: (u,v) = (2,5); peso=50; find(2) != find(5)



Resultado Final: pesoTotal=110