MARP - Práctica 1

Diego Atance Sanz

12 de diciembre de 2019

Índice

1. Detalles de implementación

Se ha implementado el algoritmo de Dijkstra apoyándose en montículos sesgados con la operación `decrecerClave()` con el objetivo de operar sobre grafos implementados mediante listas de adyacencia. Dicha implementación se ha realizado en C++ en los archivos aportados.

1.1. Ficheros principales

1. **dijkstra.cpp:** Main del programa. Implementa las siguientes funciones:

   - readGraph(...): Función implementada con el objetivo de leer grafos desde un archivo de entrada, con el formato siguiente para cada una de las aristas a incluir, tomando una arista por linea.

     ```
     vertice_origen vertice_destino peso_arista
     ```

   - createGraph(int N_VERTICES, int MAX_WEIGHT, float EDGE_CHANCE): Genera un grafo de N_VERTEX vértices, con un peso inferior a MAX_WEIGHT y con una probabilidad de generar una arista de CHANCE para cada uno de los vértices del grafo, es decir, con un 10 % de probabilidad, de cada vértice saldrán aristas aproximadamente a una décima parte de los demás vértices del grafo. En el grafo generado no se generan aristas al propio vértice origen.
   - dijkstra(...): Implementación del algoritmo de dijkstra utilizando montículos sesgados y la operación decrecerClave, más información aportada en el propio código.

2. **leftistHeap.cpp:** Implementación de los montículos sesgados. En su cabecera se definen 2 clases, Node y LeftistHeap. `Node` implementa los métodos necesarios para la construcción de cada uno de los elementos que posteriormente se gestionarán en el montículo. `LeftistHeap` implementa todos los métodos necesarios para gestionar el montículo a nivel interno y permitir su uso mediante métodos públicos. La implementación de la operación `decrecerClave()` se explicará mas adelante en este documento. Ver el archivo `leftistHeap.h` para más información al respecto de los métodos propios del montículo.

3. **WDGraph.cpp:** Implementación de un grafo dirigido y valorado mediante listas enlazadas. Contiene 3 clases, `EdgeNode`, encargada de gestionar las aristas del grafo, `VertexNote`, encargada de la gestión de los vértices de forma individual, y `WDGraph`, encargada de mantener la lista de vértices. En cada vértice se mantiene un puntero a la primera arista que sale del mismo y un puntero al siguiente vértice del grafo. La única operación destacable es `addEdge()`, encargada de la inserción de las aristas en el grafo y, en caso de ser necesario, los vértices que no estuviesen ya incluidos en el mismo, dicha operación se encuentra comentada en la propia implementación.

4. **leftistHeap.h:** Encabezados del montiículo sesgado. Contiene comentarios informativos.

5. **WDGraph.h:** Encabezados del grafo. Contiene comentarios informativos.

1.2. decreaseKey()

```
// Permite decrementar el valor de la distancia del nodo que contiene al elemento "elem"
void LeftistHeap::decreaseKey(int elem, float newDist) {
// Busca en el mapa el valor indicado por "elem"
   std::unordered_map<int, Node*>::iterator iter = _map->find(elem);
// Comprueba que exista en el monticulo
   if (iter != _map->end()) {
// Si se trata de la raiz, decrece el valor correspondiente
      if (iter->second == _root) {
         _root->_elem.first = newDist;
      }
// En caso de no ser la raiz, crea un nuevo nodo con los valores apropiados
      else {
         Node *aux = new Node(
               std::make_pair(newDist, iter->second->_elem.second),
               0,
               iter->second->_parent,
               iter->second->_left,
               iter->second->_right
               );
// Comprueba si se trata del hijo izquierdo o derecho de su padre
         if ((aux->_parent->_left->_elem.second == elem)) {
// Si se trata del izquierdo, el puntero _left del padre se apunta a NULL
            aux->_parent->_left = NULL;
// Se intercambian los hijos del padre que el derecho no debe ser hijo unico
            swapChildren(aux->_parent);
         } else
// Si se trata del derecho, el puntero _right del padre se apunta a NULL
            aux->_parent->_right = NULL;
// Perdemos el puntero _parent, descolgando de forma efectiva nuestro nodo a decrementar del
    monticulo
         aux->_parent = NULL;
// Se indica a nuestros hijos _left y _right que aux es el nuevo nodo padre, ya que es una
    direccion de memoria distinta
         if (aux->_left != NULL)
            aux->_left->_parent = aux;
         if (aux->_right != NULL)
            aux->_right->_parent = aux;
// Llamamos a mergeHeaps() con la raiz original de nuestro monticulo y el nuevo nodo que
    hemos descolgado
         _root = mergeHeaps(_root, aux);
// Eliminamos la antigua direccion del nodo del mapa e insertamos la nueva
         _map->erase(elem);
         _map->insert(std::make_pair(elem, aux));
      }
   }
}
```

2. Casos de prueba y costes

2.1. Casos de prueba

Para la generación de casos de prueba se ha utilizado el código aportado en el fichero `graph_generator.cpp`, el cual recorre una matriz de adyacencia imaginaria y, para cada una de las posibles aristas del grafo, decide en función de un valor aleatorio si se genera o no dicha arista. El coste de dicho proceso es $O(v^2)$, resultando tremendamente lento para casos de prueba grandes (más de 10.000 vértices). Esta función ha resultado incluida finalmente en el propio ejecutable de la entrega ya que la generación de un grafo en un archivo y su posterior lectura tiene varios problemas.
En primer lugar, se debe realizar una escritura de todas las aristas del grafo en un fichero, y posteriormente el programa debe leer el fichero con tal de insertar dichas aristas en un grafo, lo cual conlleva 2 recorridos completos a las aristas del grafo desde un fichero, requiriendo un tiempo extremadamente alto.
En segundo lugar, el coste en memoria de un grafo grande (de 25.000 vértices con una probabilidad de generación de arista de un 33 % en cada uno de sus vértices) es realmente alto, llegando a ocupar 15GB o más.

```
Generating a graph with:
10 MAX_VERTICES
25 MAX_WEIGHT
0.15 EDGE_CHANCE
1 4 11.9438
1 7 1.67102
2 0 18.4473
2 1 15.4167
2 6 5.2207
3 5 19.4965
3 8 3.7929
4 0 15.7367
6 2 14.7927
6 4 9.28923
7 0 9.72129
8 4 23.3855
9 4 11.1458
9 5 20.8837
9 6 12.8217
```

Figura 1: Ejemplo de fichero de entrada generado con el comando **./generate_cases 10 25 0.15**

En el fichero de ejemplo se ha generado un grafo con 10 vértices, un peso máximo por arista de 25 y una proabilidad del 15 % de generar una arista en cada caso. Cabe destacar que las 4 primeras líneas son simplemente a título informativo y no llegan al fichero final que procesa nuestro código, por lo que son mostradas por la salida de errores `stderr`.

2.2. Costes

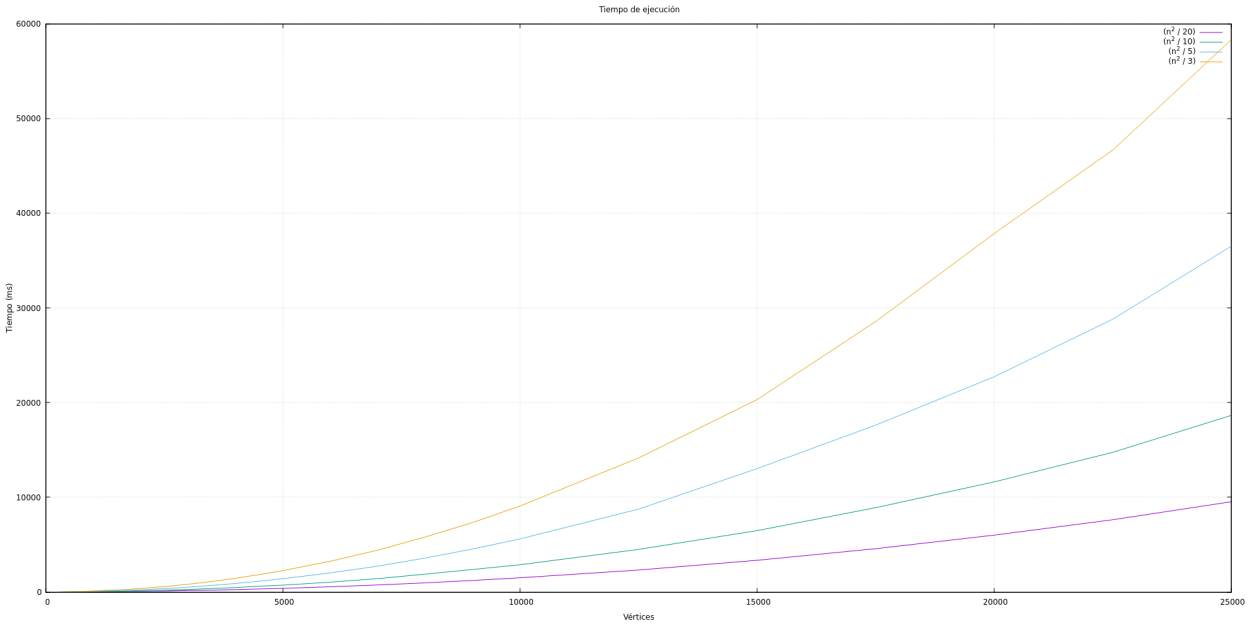

Figura 2: Tiempos de ejecución para los distintos casos de prueba generados

En la imagen adjuntada (e incluida en la entrega final de la práctica) se puede observar de forma gráfica los diferentes tiempos de ejecución del algoritmo de dijkstra con la implementación requerida. En el eje Y se muestra el tiempo en milisegundos que ha tardado la ejecución, y en el eje X se observa el número de vértices de cada caso ejecutado.
Cada una de las líneas representa una probabilidad de generación de aristas diferente, o, lo que vendría a ser lo mismo, una diferente densidad del grafo, siendo muchísimo más costoso el algoritmo para grafos densos que para grafos dispersos.
Para realizar la gráfica se ha ejecutado el código múltiples veces mediante el script `generate_data.py`, escrito en python y adjuntado en la entrega. Dicho script ejecuta 3 veces el algoritmo para cada número de vértices dado, difiriendo la distancia entre estos en 250, 500, 1.000 y 2.500 con tal de obtener datos fiables pero no sobrecargando a la máquina. También realiza la media aritmetica de los tiempos de ejecución, obtenidos directamente desde nuestro código C++ mediante la función $std :: chrono :: steady\_clock :: now()$ y $std :: chrono :: duration < double, std :: milli > (end - start).count()$, las cuales nos permiten obtener el tiempo del sistema antes y después de la llamada a la función dijkstra, que como hemos detallado previamente, ejecuta el algoritmo en sí, y presentar el output en la medida de tiempo deseada, en nuestro caso, y por mayor precisión en casos de prueba pequeños, milisegundos.
En cuanto a complejidad, la teoría se corresponde con la práctica, pudiendo observar un coste $O((a+n)logn)$, siendo **a** el número de aristas y **n** el número de vértices. Un mayor número de vértices influye de forma razonable al tiempo de ejecución, para los casos tratados, un aumento de complejidad lineal de 25.000 vértices es indiferente en comparación con el número de aristas que pueden surgir entre dicho número de vértices. En algunos casos examinados manualmente, para 25.000 vértices se generan alrededor de 200 millones de aristas, siendo este segundo factor mucho más representativo. En cualquiera de los casos, se debe tomar en cuenta que la máquina en la que ha sido ejecutado el algoritmo se trata de un portátil de trabajo bajo carga mínima, la cual aporta una capacidad de cómputo limitada. La gran mayoría del tiempo de ejecución no queda reflejado en ninguna de las gráficas, ya que corresponde a la generación de los casos de prueba y, en caso de no evitarlo, la entrada y salida.

DIEGO ATANCE SANZ
DICIEMBRE 2019
Ult. modificación 12 de diciembre de 2019
Creado con \LaTeX{}
Lic. CC-BY-NC-4.0

Esta obra está bajo una licencia Creative Commons "Reconocimiento-NoCommercial 4.0 Internacional".