Práctica MARP: Algoritmo de Dijkstra con Montículo Sesgado

Guillermo García Patiño Lenza – MARP Grupo A

22 de enero de 2021

1 CONTENIDO

2 IMPLEMENTACIÓN:

En esta práctica se ha implementado el algoritmo de Dijkstra de manera que emplee montículos sesgados con la operación decrease_Key(). Además, se han implementado una estructura auxiliar para representar un grafo con listas de adyacencia y una clase que genera grafos de manera aleatoria ajustándose a unos parámetros que se indicarán posteriormente. Todo lo descrito en esta sección se ha implementado en C++.

2.1 MODOS DE EJECUCIÓN:

1. Showmode: este modo se emplea para realizar una ejecución de prueba del algoritmo sobre un grafo generado aleatoriamente. Muestra por pantalla en grafo generado como una lista de adyacencia y los mapas de distancias y predecesores que resultan de ejecutar el algoritmo de Dijkstra.

2. Timemode: este modo se empleó para realizar la toma de tiempos automáticamente. Siguiendo las indicaciones que aparecen en el guion de la práctica, imprime por pantalla una línea con el número de vértices del grafo y la media de los tiempos de 3 ejecuciones del algoritmo para ese grafo. Para recoger los resultados de este modo se redirigía la salida estándar del programa empleando un pipe.

2.2 FICHEROS ENTREGADOS:

En esta sección se detalla cada fichero que se ha incluido en la entrega, junto con lo que implementa cada uno

1. practica.cpp:
    a. main (int argc, char * argv[ ]): se trata de la función principal del programa. Recibe los siguientes argumentos:
        i. Modo: admite los valores ‘-s’ o ‘-t’. Determina si se ejecuta una muestra del algoritmo o si se van a tomar tiempos de ejecución respectivamente.
        ii. Initial-nodes: entero que se interpreta como el número de nodos del grafo para el que se va a ejecutar la muestra en showmode o como el número de nodos que tendrá el primer grafo con el que se va a tomar tiempos en timemode.
        iii. Max-nodes: entero que se interpreta como el número máximo de nodos del grafo con el que se va a tomar tiempos en timemode. En showmode no se emplea para nada.
        iv. Inc: entero que representa el paso al que se incrementa el contador del bucle que se ejecuta en timemode.
        v. Max-Weight: entero que representa el peso máximo de una arista de cualquiera de los grafos que se van a generar en cualquiera de los modos.
        vi. Edge-chance: número con coma flotante (float) que representa la probabilidad con la que se generará una arista entre dos vértices

de un grafo en showmode. En timemode se ignora.

b. showmode(int initial_nodes, int max_weight, float edge_chance): esta función se encarga de realizar la funcionalidad del modo de ejecución que tiene su mismo nombre.

c. timemode(int initial_nddes, int max_nodes, int inc, int max_weight, float edge_chance): función que realiza la funcionalidad descrita en el modo que tiene el mismo nombre.

d. Dijkstra(grafo<int> const & g, grafo<int>::vertice const & v0, bool m): ejecuta el algoritmo de Dijkstra sobre el grafo recibido empezando por el vértice v0 que recibe como parámetro. Muestra el resultado por pantalla si recibe True en el parámetro 'm' (solo en showmode).

2. graph_generator.h : implementación de una clase que genera grafos aleatoriamente de acuerdo a los parámetros que recibe en su constructora.
3. grafo.h: implementación de una clase que representa un grafo mediante una lista de adyacencia. En el mismo fichero hay además un struct 'node' que se usa como un nodo de las listas de adyacencia del grafo. Los métodos de esta clase sirven fundamentalmente para insertar vértices o aristas en el grafo y realizar consultas sobre ellos.
4. skew_heap.h: implementación de los montículos sesgados. Incluye las operaciones necesarias para emplear el montículo como una cola de prioridad (insertar, borraMin, min). Además se incluye el método decrease_Key() que se explicará posteriormente.

2.3 Método Decrease_Key():

Para implementar este método ha sido necesario incluir las siguientes modificaciones al montículo sesgado visto en clase de teoría:

1. Mapa Clave-Nodo:
   Para poder decrecer una clave determinada, es necesario tener un puntero a esa clave dentro del montículo. El mapa que se ha incluido en el montículo permite acceder al nodo correspondiente a la clave que se desea decrecer en tiempo constante.

2. Puntero al padre:
   Una vez tengo el puntero al nodo que contiene la clave que deseo decrecer, he de descolgarlo del montículo (junto con todos sus hijos) y después unirlo con el montículo original. Para ello, guardo en cada nodo un puntero a su padre, pudiendo descolgarlo así de manera sencilla.

Una vez se han explicado estas dos modificaciones, se presenta a continuación el código completo del método decrease_key(). (Para más información ver skew_heap.h)

```
void decrease_key(clave c, valor vn){
    auto it = mapa.find(c);
    if(it != mapa.end()){
        //Obtengo un puntero al nodo que almacena la clave a decrecer
        Link l = it->second;
        if(l == root){ root -> v =  vn ;}
        else{
            if(l->v > vn){
                //Descuelgo el nodo que tiene la clave que quiero decrecer del
                // monticulo principal
                if(l->pad->iz == l){
                    l->pad->iz = nullptr;
                    l->pad = nullptr;
                }
                else{
                    l->pad->dr = nullptr;
                    l->pad = nullptr;
                }
                //Decrezco la clave
                l->v = vn;
                //Uno los dos montículos que quedan
                Link r = unir(root,l,root);
                root = r;
            }
            else{
                throw domain_error("Intentas cambiar un valor por otro mayor");
            }
        }
    }
    else{
        throw domain_error("La clave no se encuentra en el monticulo");
    }

}
```

3 CASOS DE PRUEBA Y ESTUDIO DE COSTES

3.1 CASOS DE PRUEBA:

Para generar casos de prueba automáticamente se ha creado la clase graph_generator que fundamentalmente se responsabiliza de crear grafos aleatoriamente según los parámetros num_vertices (el numero de vértices del grafo que se generará), prob_arista (la probabilidad de que se genere una arista entre dos vértices) y coste_maximo (peso máximo de una arista dentro del grafo).

Además, la función generar (la función de la clase que se encarga de crear un grafo y devolverlo como resultado) recibe como parámetro un booleano ‘casoPeor’ que indica si el grafo que se va a generar supone uno de los casos peores para el algoritmo de Dijkstra (de cada vértice sale una arista al resto de vértices) o no. En caso de serlo, se ignora el parámetro de prob_arista, y si no lo es se genera una arista de acuerdo a un número aleatorio entre 0 y 1.

Por otro lado, para asignar un peso a una arista, se genera un entero aleatorio y se calcula el módulo con el parámetro coste_máximo.

De esta manera, se podría emplear el modo showmode para visualizar uno de los grafos sencillos con los que ejecutar el algoritmo. En concreto, el siguiente caso de prueba ha sido generado con ‘./practica -s 7 15 15 20 0.5’ .

```
7 (2 , 0 ) ,(4 , 9 ) ,(5 , 0 ) ,(6 , 2 )
2 (1 , 11 ) ,(4 , 18 )
1 (2 , 8 ) ,(5 , 6 )
3 (2 , 12 ) ,(4 , 3 ) ,(5 , 14 ) ,(6 , 4 )
4 (1 , 15 ) ,(2 , 14 ) ,(5 , 7 ) ,(6 , 16 ) ,(7 , 15 )
5 (1 , 19 ) ,(4 , 3 ) ,(6 , 15 ) ,(7 , 15 )
6 (1 , 0 ) ,(2 , 8 ) ,(3 , 0 )
```

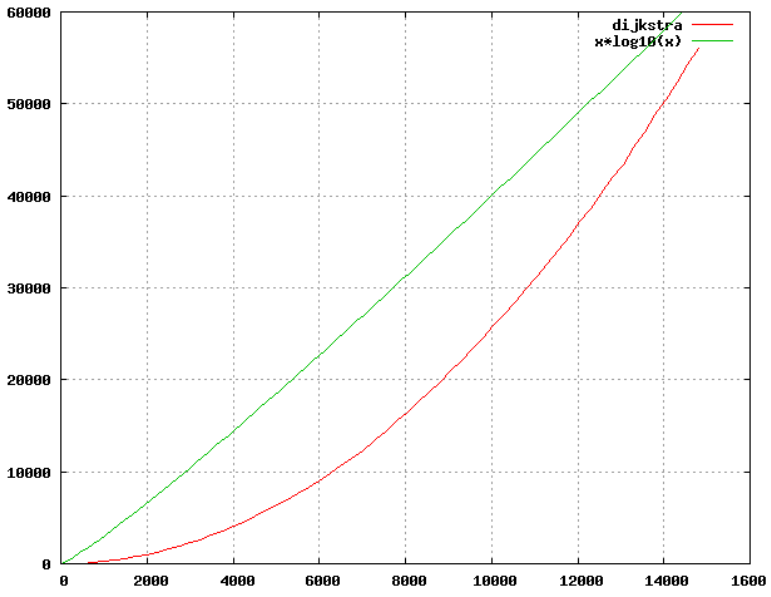
De esta manera, se ha configurado el graph_generator para crear grafos con 7 vértices en los que las aristas tienen peso máximo 20 y con 0.5 de probabilidad de generar una arista.

3.2 ESTUDIO DE LOS COSTES:

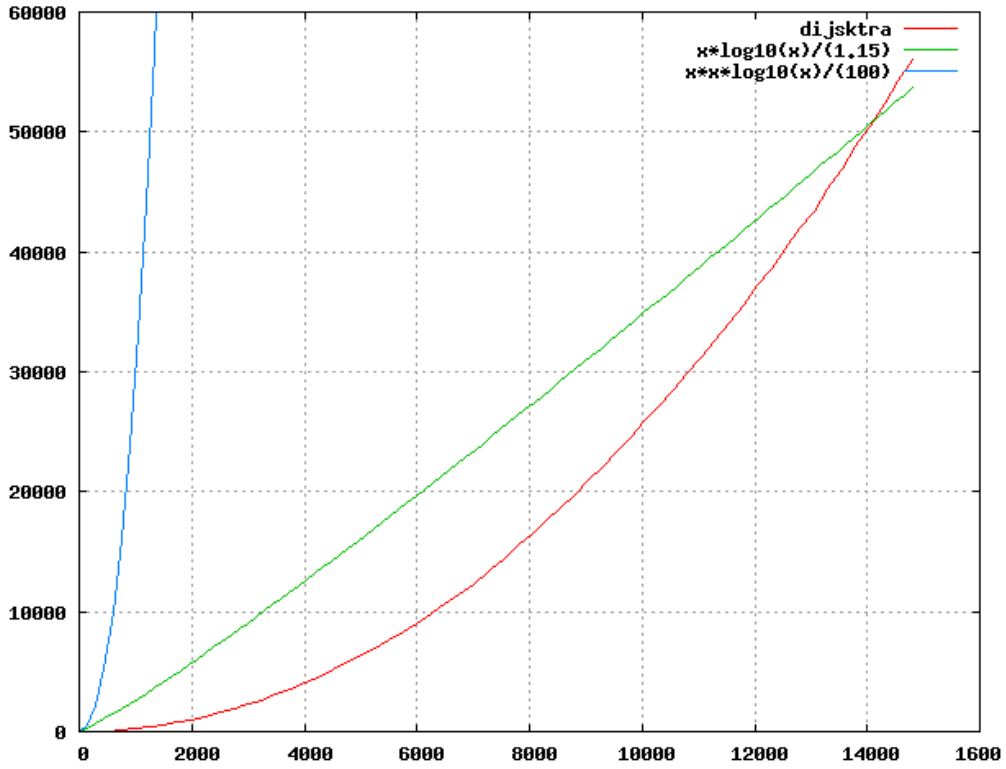
Para generar la gráfica que se adjunta al final del documento se ha lanzado el programa en timemode de la siguiente manera ‘./practica -t x y 250 1000 _ > tiempos.txt’ donde ‘x’ e ‘y’ (la cantidad inicial de vértices y la cantidad máxima de vértices) han ido variando (los tiempos que aparecen en el fichero ‘tiempos.txt’ no han sido tomados en una sola ejecución, sino que han sido tomados en varias ejecuciones donde ‘x’ e ‘y’ tomaron diferentes valores y luego reunidos en un mismo fichero. Por eso si se observa detenidamente el fichero de tiempos, se encuentran saltos en el numero de nodos que no tienen 250 nodos de diferencia).

Por otro lado, para medir el tiempo de una ejecución del algoritmo, se ha recurrido a la librería ‘chrono’ , más concretamente de la función std::chrono::steady_clock::now(). Con esta herramienta, se ha obtenido el tiempo de ejecución guardando el valor de esa función antes y después de ejecutar el algoritmo, permitiendo que al realizar una resta con std::chrono::duration<double, std::milli>(end – start).count() se obtenga el tiempo de una ejecución del algoritmo. Así, para cada valor de ‘x’ (el número de vértices del grafo) , se puede calcular el tiempo de ejecución de 3 casos de ese tamaño, y obtener la media aritmética.

Teniendo esto, es sencillo configurar el programa para obtener un fichero ‘tiempos.txt’ con el formato exacto que espera gnuplot para poder graficarlo de forma sencilla con ‘plot “tiempos.txt” with lines’. Junto con los tiempos tomados por el programa, se han graficado la función: ’$n * \log_{10} n$ ‘



A primera vista, podría parecer que el tiempo de ejecución del algoritmo está acotado por ‘ $n * \log_{10} n$ ‘. Haría falta tomar tiempos para casos de prueba más grandes, pero por falta de memoria en la máquina, se ha optado por ajustar la función ‘ $n * \log_{10} n$ ‘ con una constante (ignorada por el criterio asintótico) para notar la diferencia.



También se ha realizado un ajuste similar con ‘ $n^2 * \log_{10} n$ ‘ para que la herramienta no cambie los valores de los ejes y se puedan apreciar las tres funciones en la gráfica.

Después de todo esto, se observa como los resultados obtenidos coinciden con lo explicado en clase de teoría, y el coste del algoritmo está en ‘ $O\ (n^2 * \log_{10} n)$ ‘ en el caso peor. En realidad el coste es ‘$O((a+n)\log n$’ , pero en el caso peor se cumple que $a = n^2$ porque se generan grafos en los que desde cada vértice hay una arista hasta todos los demás.