ALGORITMO DE DIJKSTRA PARA GRAFOS DIRIGIDOS

INTRODUCCIÓN

En este documento se recoge el análisis y las conclusiones obtenidas tras la implementación del algoritmo de Dijkstra con montículo para la obtención de caminos mínimos en un grafo dirigido y valorado. El código C++ de dicha implementación está adjunto al documento y listo para ejecutarse.

ÍNDICE

Autor: Roberto Portillo Torres

1. Descripción de código

El código consta de 4 ficheros fuente pero para comenzar nos centraremos en montDijkstra.cpp y monticuloSesgado.h. En estos dos ficheros se recoge la implementación del algoritmo, en el primero el algoritmo de búsqueda de caminos mínimos y en el segundo el montículo del que hace uso.

El algoritmo de búsqueda está formado por dos partes distintas, primero la fase de inicialización en la cual se toma como raíz el nodo 0 del grafo. Desde este nodo se buscarán los caminos mínimos al resto.

Se inicializan todos los caminos al coste desde la raíz en caso de que tengan conexión directa con esta, y a ∞ en caso contrario:

```
void montDijkstra(const grafo &G, const int N, double costeMin[], int predecesor[]){

    monticuloSesgado M = monticuloSesgado();
    par minimo;
    vector<par> l;
    costeMin[0] = 0;
    predecesor[0] = 0;

    //Inicializa los N costes al máximo valor y los predecesores al nodo origen
    for(int i = 1; i < N; i++){

        costeMin[i] = __DBL_MAX__;
        predecesor[i] = 0;
        M.insertar(par(i, __DBL_MAX__));
    }

    //Actualiza el coste de los adyacentes al primer vertice
    auto it = G[0].cbegin();
    int dest;
    while(it != G[0].cend()){
        dest = it->first;
        costeMin[dest] = it->second;
        M.decrecerClave(dest, costeMin[dest]);
        it++;
    }
```

Figura 1.1: DijkstraMont.cpp, fase de inicialización

Tras esto pasa a realizar la búsqueda voraz de caminos mínimos. Para ello realiza N-2 iteraciones, siendo N el número de nodos del grafo, seleccionando el nodo más cercano a la raíz. Esto lo hace obteniendo el mínimo del montículo cuyo funcionamiento veremos más adelante. Después recorre todos los nodos adyacentes al mínimo seleccionado para actualizar sus costes en caso de que el camino hasta ellos pasando por el nodo seleccionado sea menor que el establecido hasta ahora (figura 1.2).

```
int elegido;
double coste;
int nextToMin;

for(int i = 0; i < N-2; i++){

    // Elige el nodo más cercano a la raiz (que no haya sido escogido antes)
    minimo = M.minimo();
    M.eliminarMinimo();
    elegido = minimo.first;

    // Recorre todos los nodos adyacentes al nodo elegido
    it = G[elegido].cbegin();
    while(it != G[elegido].cend()){

        nextToMin = it->first;
        coste = costeMin[elegido] + it->second;
        // Si el coste hasta los nodos pasando por el elegido es menor del
        // que tienen asignado se actualiza su coste y su predecesor.
        if(coste < costeMin[nextToMin]){

            costeMin[nextToMin] = coste;
            predecesor[nextToMin] = elegido;
            M.decrecerClave(nextToMin, coste);
        }
        it++;
    }
}
```

Figura 1.2: DijkstraMont.cpp, bucle de búsqueda

Hasta ahora hemos pasado por dos bucles de N iteraciones. El segundo de ellos contiene un bucle interior recorriendo los nodos adyacentes a cada nodo, esto tiene un coste de A + N. Por tanto, tenemos un coste de O(A+N) tomando el coste del primer bucle absorbido por el segundo. Falta por contar el coste de obtención del nodo de coste mínimo, operación que se realiza en cada iteración del bucle haciendo uso del montículo. Antes de analizar el coste echemos una ojeada a la implementación.

El montículo está implementado como un montículo zurdo de pares (nodo, valor). El primer valor es el índice del nodo en la lista de adyacencia del grafo y el segundo el coste mínimo en cada momento de la búsqueda.

Este tipo de montículos se caracteriza por la operación unión que conecta dos montículos. Después las operaciones de inserción y eliminación del mínimo harán uso de ella. La unión tiene un coste computacional de O(log N) por lo que las operaciones mencionadas anteriormente también. A continuación el código de cómo se han implementado en C++ (figura 1.3):

```
link unionSesgada(link r1, link r2){

    link aux;
    //Si un monticulo es vacio devolvemos el contrario
    if(r1 == nullptr)
        return r2;
    if(r2 == nullptr)
        return r1;

    //Identificamos la raiz menor, comparando niveles de prioridad (segundo en la tupla)
    if(r1->elem.second <= r2->elem.second){
        //Intercambiamos hizo izquierdo y derecho
        aux = r1->right;
        r1->right = r1->left;
        //Asignamos el nodo padre
        r2->padre = r1;
        //Asignamos el valor de la recursión al hijo izquierdo
        r1->left = unionSesgada(aux, r2);
        //Devolvemos la raiz menor, la otra esta insertada en niveles inferiores
        return r1;
    }
    else{
        //Accion simetrica si r2 es menor
        aux = r2->right;
        r2->right = r2->left;
        r1->padre = r2;
        r2->left = unionSesgada(aux, r1);
        return r2;
    }
```

Figura 1.3: Operación unión (MontículoSesgado.cpp)

La operación de inserción crea un nodo con el elemento a insertar y realiza la unión entre la raíz y él. Para eliminar el mínimo simplemente elimina la raíz y aplica unión sobre sus hijos izquierdo y derecho.

```
// Borrar el elemento minimo: borra la raiz y realiza la union de sus hijos
// COSTE: O(log N)
void eliminarMinimo() {

    link aux;
    if(this->raiz != nullptr){

        aux = this->unionSesgada(this->raiz->left, this->raiz->right);
        delete this->raiz;
        aux->padre = nullptr;
        this->raiz = aux;
    }
}

// Insertar un elemento: realiza la union de la raiz y el elemento a insertar (parametro)
// COSTE: O(log N)
void insertar(par const& e) {

    link nuevo = new node(nullptr, e, nullptr, nullptr);
    this->raiz = this->unionSesgada(this->raiz, nuevo);
    nodos[e.first] = nuevo;
}
```

Figura 1.4: EliminarMinimo e Insertar (MonticuloSesgado.cpp)

Como el algoritmo también requiere de la operación decrecer clave, esta añadió a la implementación. Consiste en acceder a un nodo y darle un nuevo valor. Como este nuevo valor será menor habrá que flotar el nodo mientras sea mayor que su padre, manteniendo así el invariante del montículo. Todo esto se puede hacer en O(log N):

```
// Decrecer la prioridad de un elemento y flotarlo hasta que sea mayor que el padre o raiz
// COSTE: O(log N)
void decrecerClave(int i, double valor){

    this->nodos[i]->elem = par(i, valor);
    flotar(nodos[i]);
}
```

Figura 1.5: DecrecerClave (DijkstraMont.cpp)

El código de flotar se puede leer en el fichero DijkstraMont.cpp. Mientras el nodo sea menor que su padre se intercambia con él. Al intercambiarse se actualizan todos los punteros necesarios para dejar el montículo en un buen estado.

2. Análisis teórico del coste computacional

Llegados a este punto hemos identificado los 5 bloques principales del código:

1 - Bucle de inicialización de coste N
2 - Bucle de búsqueda de caminos mínimos de coste A+N
3 - Búsqueda y eliminación del mínimo en el montículo de coste Log N
4 - Decrecer clave de coste Log N

Como ya hemos mencionado el coste del primer bucle de inicialización lo consideraremos absorbido por el coste del bucle de búsqueda. De este sabemos que se ejecuta A+N veces realizando, en el caso peor, la eliminación del mínimo y decrecer clave en cada iteración. Esto tiene un coste de O(A+N(2 Log N)). Ahora nos podemos deshacer de la constante 2 y estaríamos dentro del mismo orden de coste computacional. Por lo tanto el coste final del algoritmo nos queda dentro de O(A+N(Log N)).

3. Compilación y modos de ejecución

Junto al código hay otros dos ficheros .cpp pensados para facilitar la ejecución a quien quiera probar el código. Para compilar no debería ser necesario instalar ningún paquete adicional puesto que todas las librerías utilizadas forman parte de la STL de C++.

Tras compilar main.cpp, el ejecutable generado aceptará un argumento si se quiere generar un grafo previamente definido en un fichero .txt.

El fichero .txt debe contener una fila con un número que indique el número de nodos del grafo y a continuación cada arista definida en una línea con tres números: el nodo origen, el nodo destino y el valor de la arista. Finalmente el nombre del fichero debe ser grafo[número].txt. Para realizar la búsqueda sobre dicho fichero pasarle el número como argumento.

En caso de no introducir ningún parámetro o de no encontrar el fichero .txt el programa pasará a generar un grafo aleatorio, para lo que solicitará al usuario que indique el número de nodos deseado. El código que genera los grafos, tanto aleatorios como a partir de ficheros de texto está en generadorGrafos.cpp.

Finalmente el programa muestra por pantalla los resultados de la búsqueda dando la opción al usuario de mostrar previamente el grafo en forma de lista de adyacencia.

4. Casos de prueba

Para la depuración del código y la detección y corrección de errores se han utilizado principalmente dos casos. El primero es el grafo de las diapositivas de la asignatura:

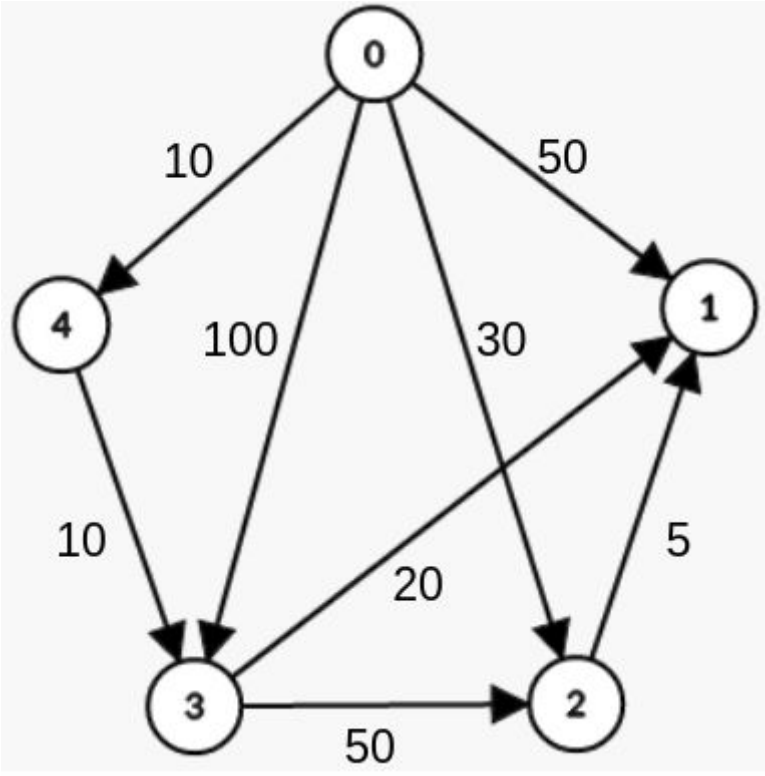

Figura 4.1: Caso de Prueba 1

```
rober@roberubuntu:~/Workspace/MAR1/Practica$ ls
Dijkstra  generadorGrafos.cpp  grafo1.txt  grafo2.txt  main.cpp  montDijkstra.cpp  monticuloSesgado.h
rober@roberubuntu:~/Workspace/MAR1/Practica$ ./Dijkstra 1

- Caminos minimos:

        Nodo     CosteCamino      Predecesor
        ----------------------------------
         0           0              0
        ----------------------------------
         1           35             2
        ----------------------------------
         2           30             0
        ----------------------------------
         3           20             4
        ----------------------------------
         4           10             0
```

Figura 4.2: Resultados de caso de prueba 1

Este caso no es muy complejo pero contribuyó a identificar algunos errores de funcionamiento. Una vez se consiguió una solución se probó un grafo un poco más complejo que ayudó a identificar algunos errores de inicialización. La principal diferencia entre este segundo ejemplo y el primero es que el nodo raíz no es adyacente a todos los demás.

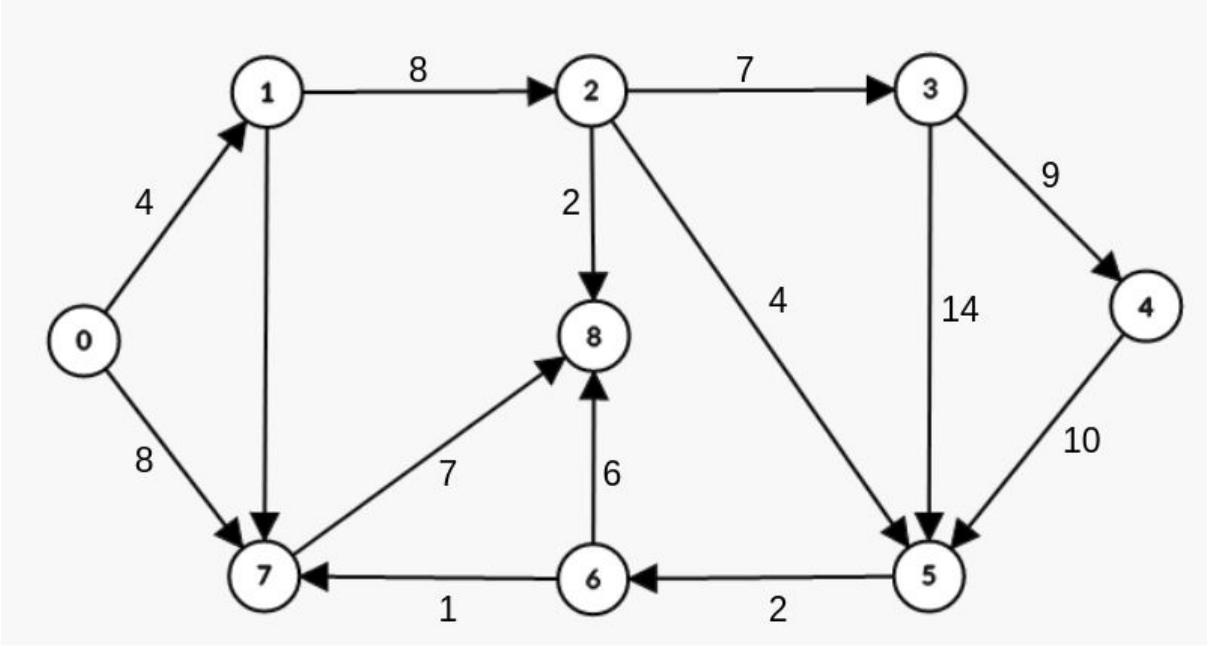

Figura 4.3: Caso de prueba 2

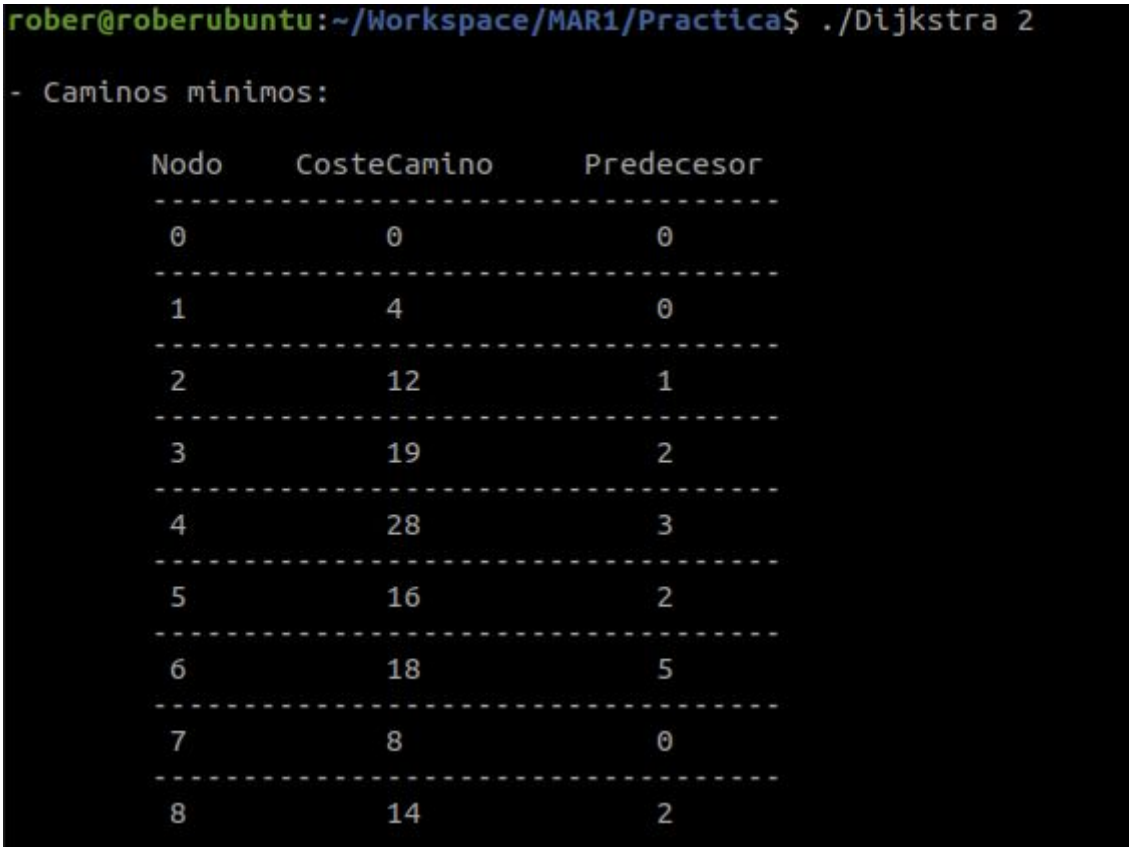

Figura 4.4: Resultados de caso de prueba 2

5. Análisis de tiempos de ejecución

Para medir el tiempo de ejecución se han generado grafos aleatorios con un número de nodos en el rango de 0 a 15000. Como se recomienda en el guión cada medida se hace tres veces y se extrae la media. Además, la máquina no ejecuta nada más que el algoritmo ni está conectada a internet durante la medición.

Como se puede observar en la gráfica (figura 5.1), el coste del algoritmo evoluciona acorde a un crecimiento del orden de O(n log n). Ligeramente mayor debido a que realmente el coste es O((a+n) log n), por lo tanto los resultados concuerdan con lo previsto por la teoría.

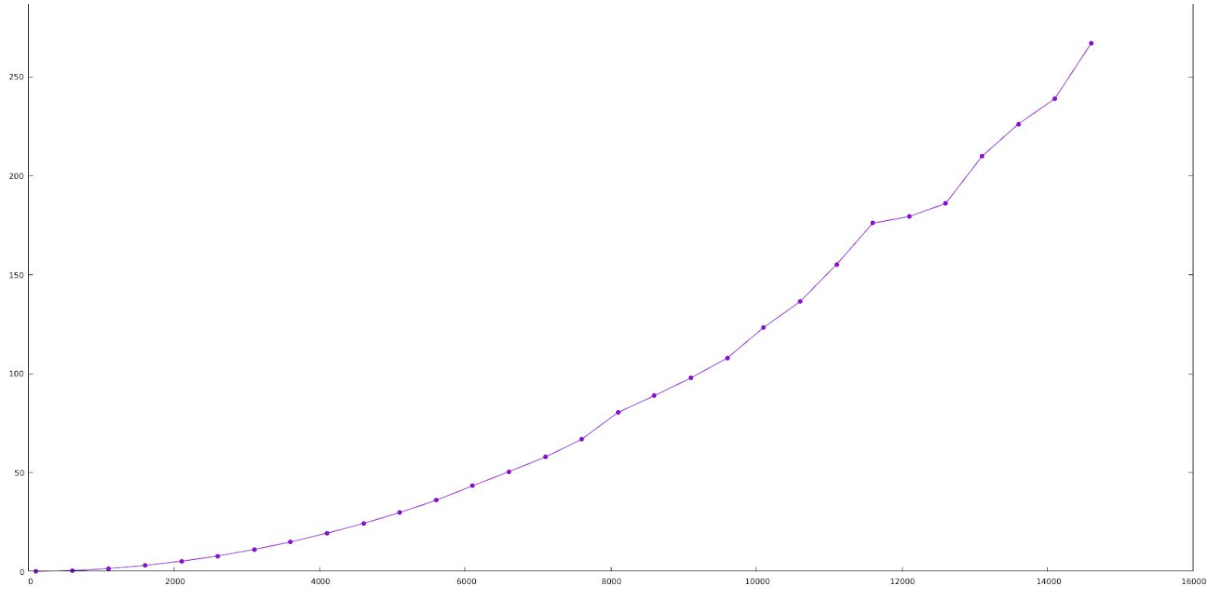

Figura 5.1: Gráfica de Coste de MontDijkstra