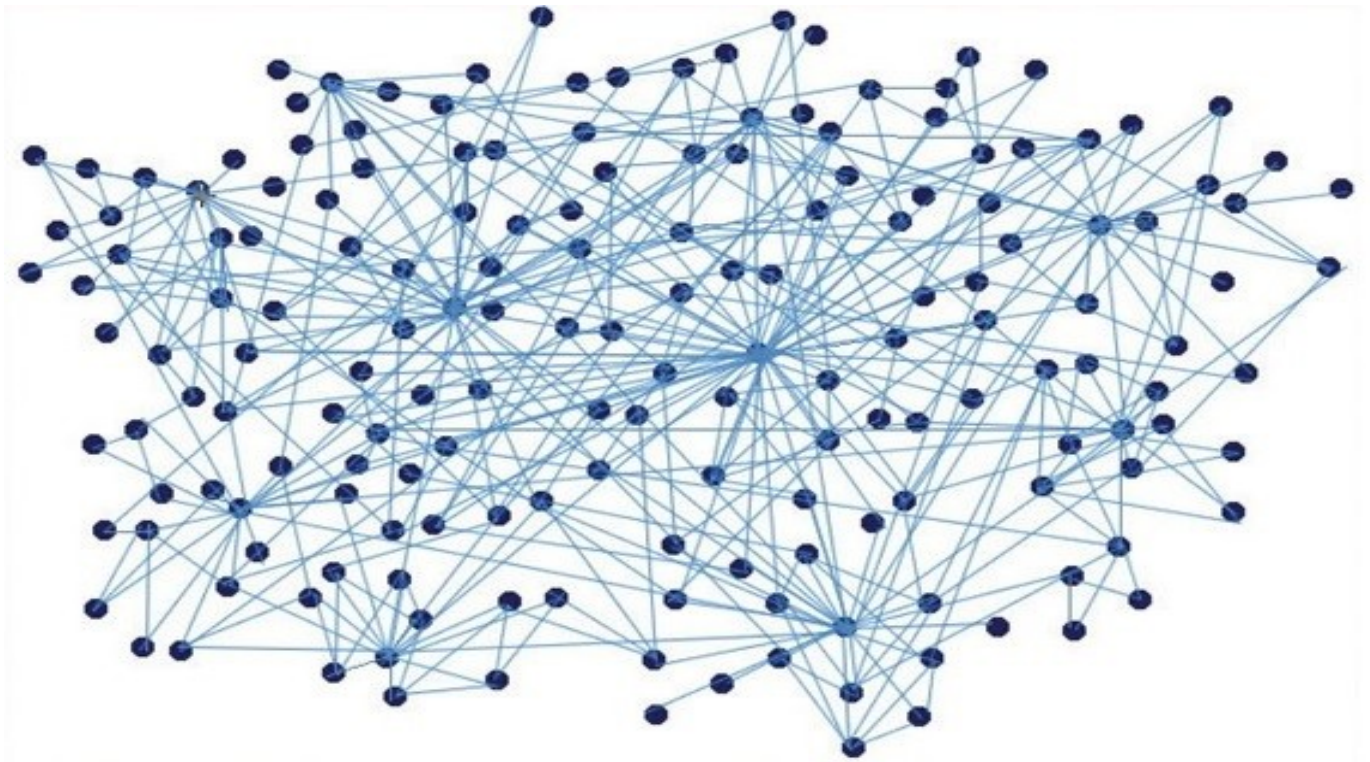


# ALGORITMO DE DIJKSTRA



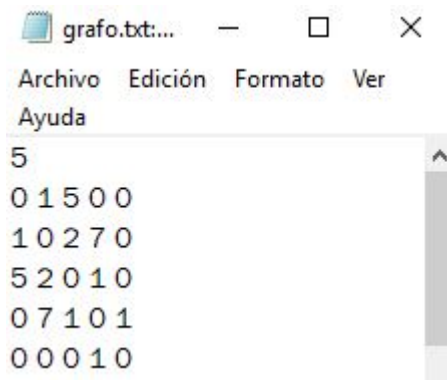
Paula Muñoz Lago

## ÍNDICE

1. ALGORITMO DE DIJKSTRA CON VECTOR DE COSTES.....	2
1.1    CÓDIGO.....	3
1.2    EJECUCIÓN.....	5
1.3    COSTES.....	5
2. ALGORITMO DE DIJKSTRA CON MONTÍCULO DE WILLIAMS.....	6
2.1    MONTÍCULO.....	6
2.2    CÓDIGO.....	7
2.3    EJECUCIÓN.....	9
2.4    COSTES.....	10
3. CONCLUSIÓN .....	10

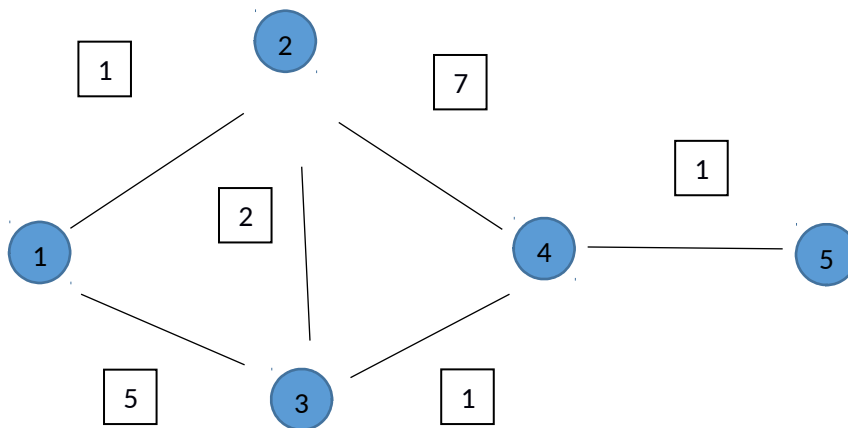
## 1. ALGORITMO DE DIJKSTRA CON VECTOR DE COSTES

Para implementar el algoritmo se ha usado el lenguaje c++. El grafo está representado con una matriz de adyacencia que se cargará a partir de un fichero, en el que tendremos en primer lugar el tamaño y a continuación la matriz, como en el siguiente ejemplo:



```
5
0 1 5 0 0
1 0 2 7 0
5 2 0 1 0
0 7 1 0 1
0 0 0 1 0
```

Que representa el siguiente grafo:



Se ven representados los nodos con círculos y los costes de cada arista con cuadrados.

## 1.1 CÓDIGO

Para obtener los resultados requeridos, inicialmente se pregunta al usuario si quiere probar el algoritmo una sola vez, si este es el caso se carga el archivo “grafo.txt” como se ha explicado anteriormente. En caso contrario, se generarán 200 matrices aleatorias de tamaños de 1 a y se irán guardando los tiempos de ejecución del algoritmo sobre cada matriz en el archivo “costes.txt” para a continuación poder hacer un estudio del coste temporal sobre las gráficas generadas en la aplicación “gnuplot”, para ello obtenemos el tiempo antes de ejecutar el algoritmo y el tiempo al terminarlo y calculamos la diferencia de la siguiente forma:

```
clock_t begin = clock();  
Dijkstra(matriz, x, coste_min, predecesor);  
clock_t end = clock();  
double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
```

El algoritmo está programado en la función Dijkstra(), a la cual le pasamos como parámetro la matriz, el tamaño de esta, y un array en el que se guardarán los costes mínimos de llegar a cada nodo, al igual que un array llamado “predecesor”, en el cual para cada posición se guardará el nodo anterior a éste. Antes de proceder veamos como ejemplo cual sería el resultado en estos arrays tras la ejecución del algoritmo para el grafo de la página anterior.

	1	2	3	4	5
Coste	0	1	3	4	5
Predecesor	-	1	2	3	4

Para ello en primer lugar se establece que el coste de llegar a 0 es 0 y su predecesor es un valor erróneo, ya que no tiene. En un conjunto llamado candidatos, `set<int>` candidatos, se introducirán todos los nodos vecinos del 0, asignando su coste mínimo al coste de llegar desde el nodo 0 hasta éste, además se actualiza su predecesor. Si el nodo no está unido al 0, establecemos que el coste de llegar a éste es infinito, para evitar tener valores erróneos, ya que si no se asigna ningún valor, no funcionarían algunas operaciones más adelante.

```
for (int i = 2; i < n; i++) {  
    if (mat[0][i] != 0) {  
        candidatos.insert(i);  
        coste_min[i] = mat[0][i];  
        predecesor[i] = 0;  
    } else coste_min[i] = std::numeric_limits<int>::max();  
}
```

Entre los candidatos, es decir, los vecinos del nodo inicial, el algoritmo buscará cuál tiene el coste de camino mínimo y a continuación lo es borrado del conjunto “candidatos”.

Tras obtenerlo, actualizará los vectores, es decir, para todos los nodos,  $j = [1..n]$ , a excepción del primero, se comprueba que el coste mínimo de llegar al nodo sea menor que el acceso al mismo a través del elegido. En caso contrario, cambiará su coste, ya que ha encontrado un camino más rápido, y establece que el predecesor es el elegido. Además es necesario insertar el nodo en el conjunto de candidatos, ya que puede darse que desde el conjunto de candidatos iniciales, es decir, vecinos de 0, no pueda llegarse directamente al nodo  $j$ .

```
for (int j = 1; j < n; j++) {  
    if (mat[elegido][j] != 0 &&  
        coste_min[elegido] + mat[elegido][j] < coste_min[j]) {  
        coste_min[j] = coste_min[elegido] + mat[elegido][j];  
        predecesor[j] = elegido;  
        candidatos.insert(j);  
    }  
}
```

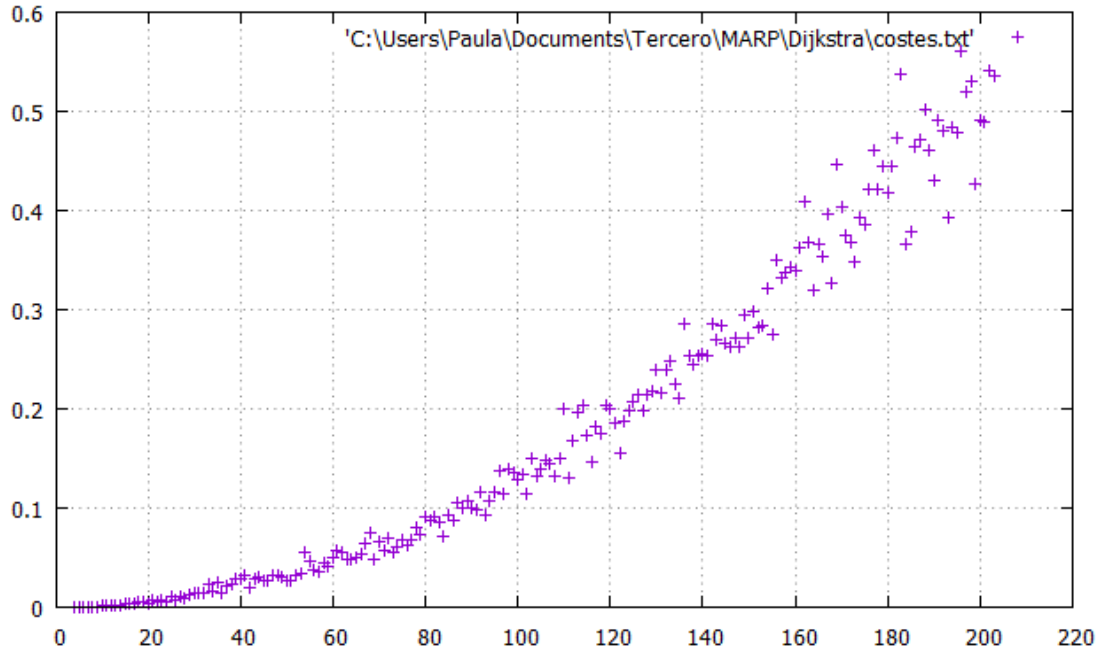
## 1.2 EJECUCIÓN

Tras la ejecución del programa, para el archivo mostrado en la primera página, obtenemos el siguiente resultado:

```
Quiere cargarlo de un archivo o generar 200 matrices aleatorias? (archivo, aleatorias): archivo
----- GRAPH 0-----
Aristas:
1 - 2
1 - 3
2 - 3
2 - 4
3 - 4
4 - 5
Nodo   coste  predecesor
2      1      0
3      3      1
4      4      2
5      5      3
Numero de aristas: 6
Numero de nodos: 5
-----
```

## 1.3 COSTES

Tras la ejecución del algoritmo para 200 matrices de tamaños de 4 a 203 generadas de forma aleatoria, se obtiene, con la aplicación “gnuplot”, la siguiente gráfica:



La complejidad de este algoritmo es cuadrática,  $O(n^2)$ , como puede comprobarse en la gráfica obtenida. Las variaciones entre dos muestras de tamaños similares pueden darse debido a la aleatoriedad de las muestras.

El procedimiento para obtener estos costes ha sido mediante la función que aparece más abajo. Puesto que queremos guardar el coste de cada ejecución junto con el tamaño de la muestra, para ello se usa la siguiente función:

```
void write_costs(double time, int n) {  
    ofstream file;  
    file.open(file_costes, std::ofstream::app);  
    file << n << "\t" << time << endl;  
    file.close();  
}
```

Abrimos el archivo en modo “app”, de “append”, para escribir al final del archivo en vez de sobrescribirlo. Este archivo es que le lee la aplicación “gnuplot” para generar la gráfica.

## 2. ALGORITMO DE DIJKSTRA CON MONTÍCULO DE WILLIAMS

### 2.1 MONTÍCULO

A continuación vemos la implementación del mismo algoritmo pero esta vez con **montículos**, estructura equilibrada cuyos nodos son mayores o iguales a sus hijos, en el caso en el que hagamos un montículo de mínimos.

El tipo del montículo va a ser “par”, que se compone de un elemento y su prioridad, aunque para resolver este algoritmo, se utilizan estos campos como valor (el número del nodo), y como prioridad el coste de llegar a él.

```
class par {  
    int elem;  
    int prioridad;  
  
public:  
    par() { elem = 0, prioridad = 0; }  
    par(int x, int y) { elem = x, prioridad = y; }  
    int getElem() { return elem; }  
    int getPrio() { return prioridad; }  
    void changePrio(int p) { prioridad = p; }  
    void changeElem(int e) { elem = e; }  
};
```

A su vez, el montículo tendrá un array de tipo “par”, un array de posiciones, y un entero que determinará cuál es el último elemento en el array de pares.

```
par *v;  
int *posiciones;  
int ultimo;  
int max_size;  
int size;
```

Las operaciones que implementará serán la inserción, la cual en este tipo de estructuras se realiza por la izquierda, flotar, hundir, intercambiar, eliminar el mínimo (la raíz) etc...

Una operación a destacar es decrecer clave, implementada en la función: `void decrecer_clave(int i)`, siendo `i` el índice del elemento cuya clave se quiere decrecer en una unidad. Comprobará que este índice es menor que el tamaño del montículo y si es así decrece el elemento, es decir, el identificador del nodo y llama a la función flotar, ya que en este punto puede darse que se haya violado la condición de montículo de mínimos, siendo su padre mayor que el nodo en la posición `i`.

Esta operación comprueba que no estemos tratando con el elemento mínimo del montículo, y si es así recorre toda su rama intercambiando el elemento por su padre hasta encontrar un padre menor que él.

```
void Monticulo_Williams_Minimos::flotar(int i) {  
    if (i != 0) {  
        for (; v[padre(i)].getElem() > v[i].getElem(); i = padre(i))  
        {  
            intercambiar(padre(i), i);  
        }  
    }  
}
```

## 2.2 CÓDIGO

Para la obtención de los caminos mínimos a todos los nodos del grafo se usa la función:

```
void Dijkstra(int **mat, int n, int *coste_min, int *predecesor)
```

En la que en primer lugar se establece que el coste mínimo de llegar al primer nodo es 0, y su predecesor es un valor erróneo.



*coste\_minimo, predecesor: de 0 a n-1*

Manteniendo los mismos pasos que en el algoritmo anterior, a continuación el algoritmo recorre la primera fila de la matriz que representa el grafo y rellena el array de costes minimos para todos los nodos adyacentes al primero. Todo nodo que no sea vecino del primero tendrá un coste minimo de acceso a él igual a infinito.

Copia en una lista de tipo “info\_arista”, la primera fila de la matriz, es decir, en ella introduce los “Candidatos”, únicamente aquellos nodos conectados con el nodo 1.

```
typedef struct info_arista{  
    int destino;  
    int valor;  
};
```

Usa el “destino” como identificador del nodo (de 1 a n, ya que la matriz y las listas empiezan en la posición 0, de tal forma que el “nodo 1” se guarda en la posición 0, por tanto el nodo 2 se almacena en la posición 1 etc), y el valor será el coste de llegar hasta ese nodo. Por tanto la operación copiar lista se hace de la siguiente forma:

```
list<info_arista> copia_lista(int **m, int fila, int n) {  
    list<info_arista> l;  
    info_arista ar;  
    for (int i = fila + 1; i < n; i++) {  
        if (m[fila][i] != 0) {  
            ar.destino = i;  
            ar.valor = m[fila][i];  
            l.push_back(ar);  
        }  
    }  
    return l;  
}
```

En esta función el algoritmo recorre la matriz desde la posición indicada por la fila + 1 ya que solo se quiere estudiar conexiones entre nodos que no hayamos procesado anteriormente, por ejemplo, si queremos obtener la lista de adyacentes al dos, no interesa introducir en la lista el nodo 1, en el caso en el que exista una conexión entre ellos, ya que podemos asegurar que ya se ha procesado anteriormente.

Volviendo a la función principal, se inserta en el montículo todos los elementos de esa lista, `M->insert(par(l.front().destino, coste_min[l.front().destino]))` Después, recorrer

todos los posibles elementos del grafo sin contar el último, ya que este ya estará analizado.

Lo haremos estableciendo el “elegido” como el elemento mínimo del montículo, gracias a una operación de la clase implementada, y lo borramos.

```
minimo = M->elem_minimo();  
M->borra_min();  
elegido = minimo.getElem();
```

Copia la lista de este elemento y la recorre, esta vez actualizará los vectores.

Establecerá que si el coste de llegar a su elemento adyacente es menor que el coste anteriormente calculado para llegar a él (ya sea porque tiene más “vecinos”, o su valor sigue siendo infinito), en ese caso se actualizará su nuevo coste y su predecesor será el elegido + 1, ya que puesto que nuestros vectores van de 0 a n-1, al imprimir los resultados ha de adaptarse al usuario.

```
j = l.front().destino;  
coste = coste_min[elegido] + l.front().valor;  
if (coste < coste_min[j]) {  
    coste_min[j] = coste;  
    predecesor[j] = elegido + 1;  
    M->modificar(j, coste);  
}
```

Como se puede ver en la última línea, modifica el coste de este nuevo elemento en el montículo. Esta operación está implementada dentro de la clase

[Monticulo\\_Williams\\_Minimos](#), de tal forma que si el elemento a modificar no existe lo inserta, como podemos ver:

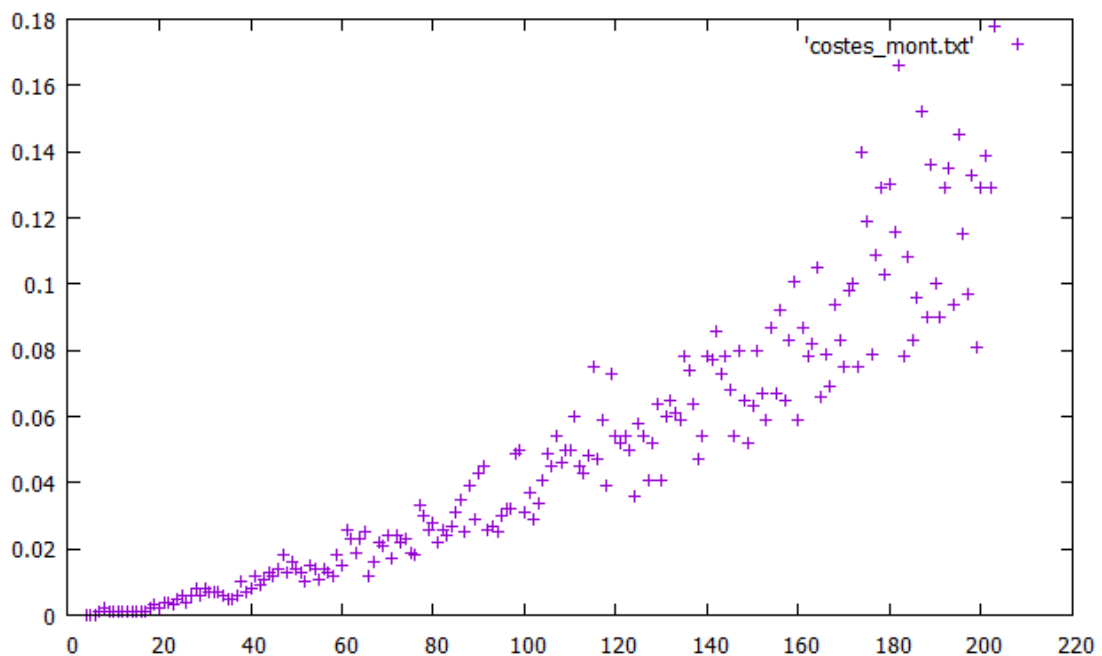
```
void Monticulo_Williams_Minimos::modificar(int pos, int prio) {  
    if (pos > size) {  
        insert(par(pos, prio));  
    }  
    else v[pos].changePrio(prio);  
}
```

## 2.3 EJECUCIÓN

Para probar la aplicación usaremos el archivo que se muestra en la página 1, y obtenemos los siguientes resultados:

```
Quiere cargarlo de un archivo o generar 200 matrices aleatorias? (archivo, aleatorias): archivo
----- GRAPH 0-----
Aristas:
1 - 2
1 - 3
2 - 3
2 - 4
3 - 4
4 - 5
Nodo   coste  predecesor
2      1      1
3      3      2
4      4      3
5      5      4
Numero de aristas: 6
Numero de nodos: 5
-----
```

## 2.4 COSTES



El coste obtenido de esta forma es número de aristas \* logaritmo (número de nodos)

$$O(m \cdot \log(n))$$

Lo cual es bastante más eficiente que el primer caso, hecho con vectores de coste.

### 3. CONCLUSIÓN

El algoritmo de Dijkstra encuentra los caminos mínimos entre el nodo inicial y el resto de nodos del grafo, siendo más eficiente su implementación con montículos binarios, o montículos de Williams.