**2020**

ALGORITMOS DE ENRUTAMIENTO

PRACTICA II

IGNACIO ORTEGA Y ROCÍO RUIZ

ÍNDICE

[ALGORITMO DE DIJKSTRA 2](#_Toc39181421)

[INTRODUCCIÓN 2](#_Toc39181422)

[IMPLEMENTACIÓN 2](#_Toc39181423)

[1. INICIALIZACIÓN 2](#_Toc39181424)

[2. ALGORITMO 2](#_Toc39181425)

[EJEMPLO 4](#_Toc39181426)

[INUNDACIÓN 5](#_Toc39181427)

[INTRODUCCIÓN 5](#_Toc39181428)

[IMPLEMENTACIÓN 5](#_Toc39181429)

[1. INICIALIZACIÓN 5](#_Toc39181430)

[2. ALGORITMO 6](#_Toc39181431)

[EJEMPLO 7](#_Toc39181432)

[VECTOR DISTANCIA 8](#_Toc39181433)

[INTRODUCCIÓN 8](#_Toc39181434)

[IMPLEMENTACIÓN 8](#_Toc39181435)

[1. INICIALIZACIÓN 8](#_Toc39181436)

[2. ALGORITMO 9](#_Toc39181437)

[EJEMPLO 10](#_Toc39181438)

[COMPARATIVA 11](#_Toc39181439)

[CONCLUSION 11](#_Toc39181440)

# ALGORITMO DE DIJKSTRA

## INTRODUCCIÓN

Dijkstra es un algoritmo de enrutamiento que elije la ruta más corta dado un vértice origen, hacia el resto de los vértices de un grafo que tiene pesos en cada arista.

Muchos protocolos de enrutamiento están basados en el algoritmo de Dijkstra.

## IMPLEMENTACIÓN

### 1. INICIALIZACIÓN

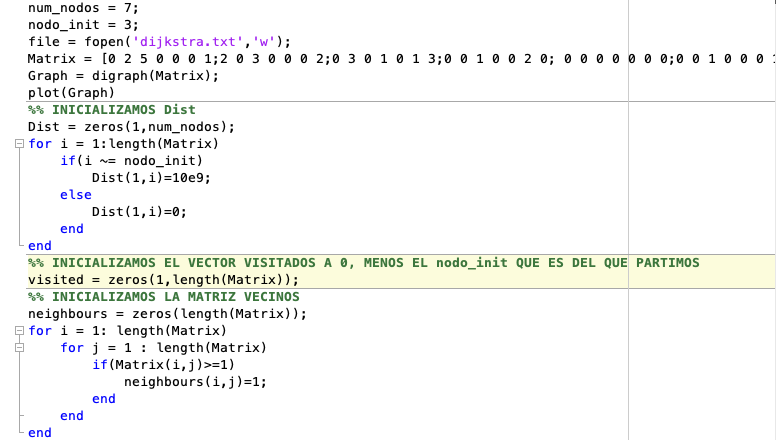
**1.1.** Definimos el numero de nodos, y el nodo inicial desde el que se empezará el algoritmo.

**1.2.** Creamos el archivo por el cual sacaremos los resultados iteración a iteración.

**1.3.** Creamos un array “Dist” de distancias (de tamaño 1 x Nº nodos). Lo rellenamos con un valor infinito relativo(10e9), ya que las distancia son desconocidas al principio, exceptuando la del nodo inicial, que tendrá un valor de 0 ya que, la distancia a sí mismo es 0.

**1.4.** Inicializamos el vector de visitados a 0, menos el nodo inicial.

**1.5.** Inicializamos la matriz de vecinos con ceros. Ponemos valor 1 a los nodos que son vecinos. 



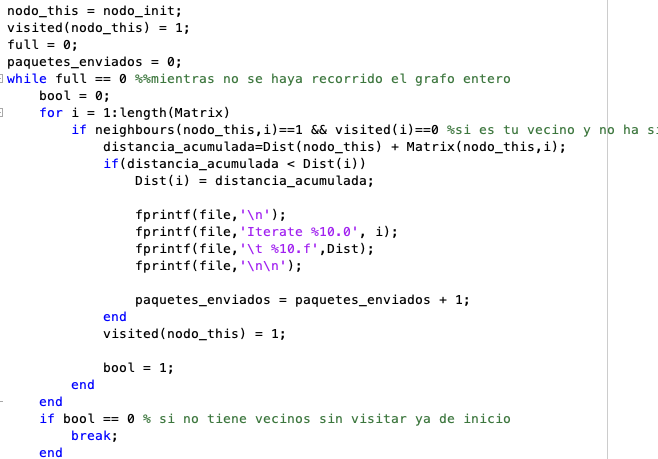
### 2. ALGORITMO

Se recorren todos los nodos adyacentes al nodo actual no visitados, mientras que el grafo no haya sido recorrido entero.

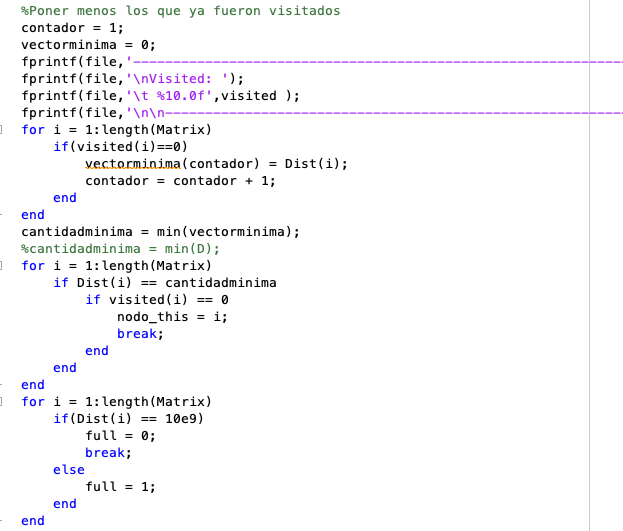
**2.1.** Para el nodo actual, se calcula la distancia acumulada desde dicho nodo hasta sus vecinos. La distancia acumulada desde el nodo actual a sus vecinos es la distancia que actualmente tiene el nodo en el vector Dist + la distancia del nodo actual a sus vecinos definida por el grafo.

**2.2.** Si la distancia acumulada es menor que la distancia en el vector Dist, entonces se actualiza el vector con esta distancia acumulada. Será su nueva distancia mínima. Imprimimos las distancias de la iteración. Aumentamos en número de paquetes enviados.

**2.3.** Se marca el nodo actual como visitado



**2.4**. Se toma como próximo nodo actual el de menor valor en Dist de los nodos no visitados.



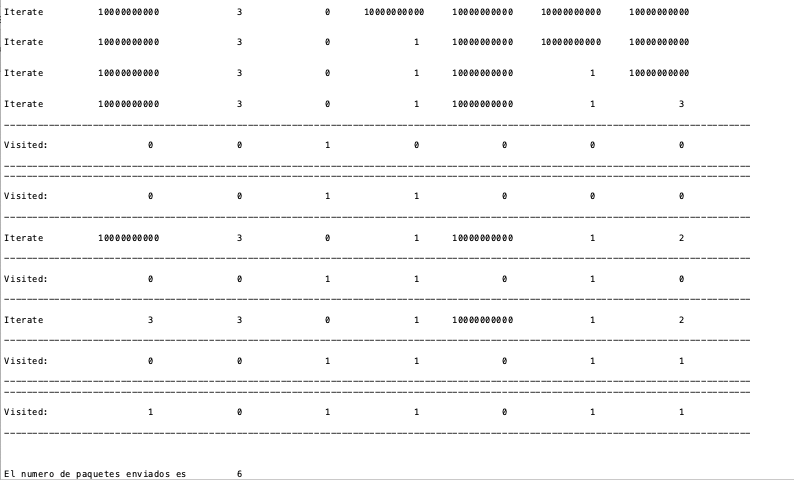
## EJEMPLO

Como demostración hemos elegido el grafo de prueba asignado a esta practica:

Imagen que contiene tabla, hombre, aire, grupo

Descripción generada automáticamente

Como resultado, generamos un fichero .txt que muestra las iteraciones. Y, cuando fija un nodo, muestra el vector de los nodos visitados. Por último nos muestra en numero de paquetes enviados.



# INUNDACIÓN

## INTRODUCCIÓN

Inundación es un algoritmo de enrutamiento que se basa en la difusión de los paquetes a todos los nodos vecino menos al nodo que te lo envía. Por ello encuentra la ruta más corta. El principal inconveniente que plantea este método es el gran número de paquetes que se generan. Hay varias posibilidades para limitarlo. Por ejemplo, cada nodo puede mantener una lista de los paquetes ya transmitidos, y al recibir un duplicado destruirlo y no retransmitirlo. Otra posibilidad, es limitar el tiempo de vida del paquete. Este ultimo es el que hemos implementado en esta practica. El numero máximo de saltos que puede dar antes de encontrar el nodo final es 20. De manera que si un nodo ha caído, y a los 20 saltos no lo encuentran, se dejarán de enviar paquetes.

## IMPLEMENTACIÓN

### 1. INICIALIZACIÓN

**1.1.** Definimos el numero de nodos, el nodo inicial y el nodo final. Inicializamos a 0 el numero de paquetes. Definimos que el nodo actual es el nodo de inicio.

**1.2.** Creamos el archivo por el cual sacaremos los resultados iteración a iteración.

**1.3.** Inicializamos la matriz de vecinos con ceros. Ponemos valor 1 a los nodos que son vecinos.

**1.4.** Inicializamos la matriz *Envíos*, donde se registra la cantidad de paquetes que recibe cada nodo. La primera fila solo tendrá a 1 en nodo inicial (nodo que tiene el paquete). La segunda fila tendrá a 1 los vecinos del nodo inicial, nodos que recibirán el paquete en la primera iteración.

**1.5.** Inicializamos el numero de saltos debido al punto anterior. El numero de filas (de *Envíos*) a 3, ya que anteriormente hemos llenado las dos primas filas. Por tanto, el numero de iteración también lo inicializamos a 3.



### Captura de pantalla de un celular Descripción generada automáticamente

### 

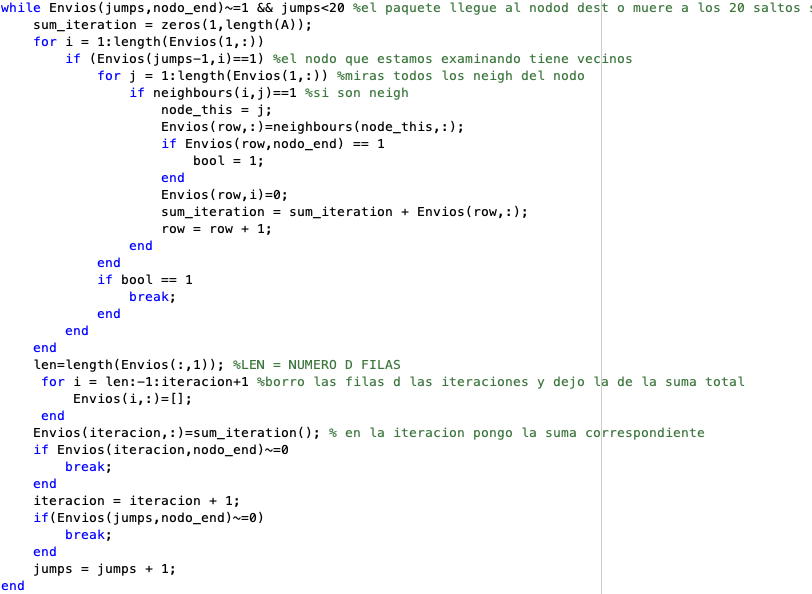
### 2. ALGORITMO

Se recorren nodos adyacentes al nodo actual hasta que se llegue al nodo destino o hasta que se realicen 20 saltos sin encontrar el nodo destino.

**2.1.** En la matriz de *Envíos,* tenemos guardados los vecinos a los que debemos enviar el paquete. Para cada uno de ellos, vamos a mirar sus vecinos, y añadirlos a la matriz de Envíos en una nueva fila. Ponemos al nodo que nos ha mandado el paquete a 0, para no reenviarle el paquete. Sumamos el número de paquetes que va recibiendo cada nodo en cada iteración y lo guardamos en sum\_iteration. Si se ha alcanzado el nodo final, dejamos de mirar más vecinos.

**2.2.** Borramos las filas de cada iteración y las sustituimos por las guardadas en sum\_iteration.

**2.3.** Pasamos a la siguiente iteración, y añadimos un numero de salto.



Para calcular el numero de paquetes enviados, recorremos la matriz de *Envíos* y sumamos todos los elementos.

Una captura de pantalla de un celular

Descripción generada automáticamente

## EJEMPLO

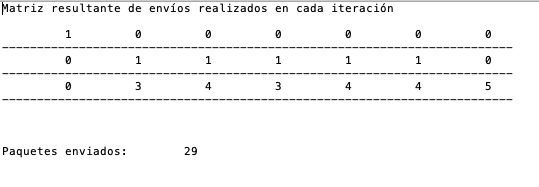
Como demostración hemos elegido un nuevo grafo al asignado como prueba, para que se compruebe, al menos, dos iteraciones.

Imagen que contiene exterior, paraguas, aire, grande

Descripción generada automáticamente

Como resultado, generamos un fichero .txt que muestra los paquetes enviados a cada nodo en cada iteración. Se observa como en el inicio solo tiene el paquete inicial el nodo inicial. En la siguiente iteración, los nodos vecinos, excepto el que lo enviaba tienen un paquete. En la siguiente iteración todos tienen paquetes duplicados ya que son vecinos entre ellos.

Por último nos muestra en numero de paquetes enviados.



Se observa que lo hace en la ruta mínima pero los paquetes duplicados son exagerados.

# VECTOR DISTANCIA

## INTRODUCCIÓN

El enrutamiento de un protocolo basado en vector de distancias requiere que un *Router* informe a sus vecinos de los cambios en la topología o cuando se detecta un cambio en la topología de la red. Este proceso es lento y nos podría llevar al problema denominado cuenta hasta el infinito, en el que un *Router* tarda demasiado en ser notificado de una caída de conexión.

Hemos decidido hacer la implementación en Java para que cada *Router* fuera un objeto con su tabla interna, y la simulación fuera mas real.

## IMPLEMENTACIÓN

### INICIALIZACIÓN

Nuestro programa consta de 3 clases principales: *Graph*, *Router* y *DistanceVector*. Hay otra clase llamada *Simulation* que contiene el *main* del programa y algunas funciones como lectura y escritura de ficheros.

* 1. Leemos el grafo del archivo. De este grafo, sacamos las tablas de internas de cada *Router*.

Cada *Router* tendrá además una tabla interna auxiliar para guardar los cambios hasta que sean definitivos. También contienen una tabla donde almacenan la menor distancia a cada *Router*. (Todas las tablas se irán actualizando).

* 1. Almacenamos en *DistanceVector* todos los *Routers* para poder hacer las respectivas

iteraciones.

* 1. Llamamos a la función que empieza a ejecutar el algoritmo.

***Imagen que contiene negro, sostener, teléfono

Descripción generada automáticamente· Inicialización tablas internas:***

***· Main:***

Captura de pantalla de un celular con letras

Descripción generada automáticamente

### 2. ALGORITMO

Para cada *Router*, se recorren todos los *Routers* vecinos.

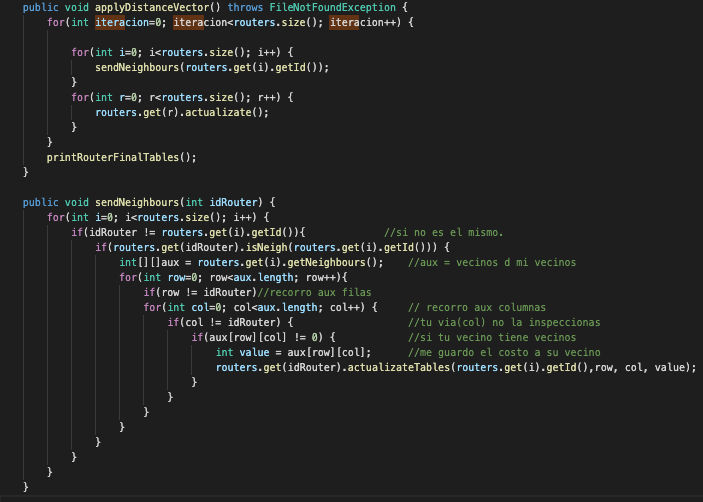
**2.1.** Éstos, te darán sus costes a sus vecinos, y por la vía en la que llegan a él. (Función:

*sendNeighbours()*). El *Router* con estos nuevos datos actualizará su tabla interna auxiliar

**2.2.** Además de actualizar su tabla interna auxiliar con los datos vecinos, esta función(*Router.actualizateTables()*), actualiza el valor minimo para llegar a cada Router. Estos datos mínimos se guardan en la tabla *lessValues.*

**2.3.** Se itera tantas veces como *Routers* haya.

En cada iteración, se actualizan las tablas internas de cada Router con los valores de la tabla interna auxiliar. Esto nos permite explorar cada vez más *Routers*.

 Captura de pantalla de un celular

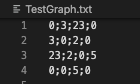
Descripción generada automáticamente

## EJEMPLO

Como ejemplo, hemos escogido un algoritmo de 4 *Routers.* Imagen que contiene texto

Descripción generada automáticamente

*Sacamos otro fichero con las tablas de* *Insertamos un fichero de texto con el grafo:*

Imagen que contiene calculadora, electrónica, teléfono, remoto

Descripción generada automáticamenteImagen que contiene remoto, electrónica, control, negro

Descripción generada automáticamenteCaptura de pantalla de un celular con letras

Descripción generada automáticamente *distancia mínima:*

# COMPARATIVA

• Retardo

El retardo en un algoritmo de enrutamiento es, en términos generales, la cantidad de tiempo que transcurre desde que un nodo manda un paquete hasta que se recibe en el nodo destino. Dependiendo del algoritmo el retardo será por normal general distinto. El algoritmo más rápido en este caso será inundación, ya que al enviar paquetes por todas las vías encontrará la mas corta en el menor tiempo posible. Dijkstra, sin embargo, tendrá que hacer unas cuantas más iteraciones hasta encontrar la ruta más corta. Como algoritmo de más retardo está vector distancia, ya que si se cae una conexión tardaría un tiempo desmesurado en notificarse a todos los *Routers* de este cambio.

• Ancho de banda

El ancho de banda en redes es la cantidad de información que se puede enviar a través de una conexión en un período de tiempo dado.

En el algoritmo de inundación, cada nodo, envía en cada iteración los paquetes por todas las vías posibles menos por la que le ha llegado el paquete. Esto puede llegar a causar la ocupación de un gran ancho de banda. Y como consecuencia, dependiendo de la red, una congestión de tráfico.

En el algoritmo de Dijkstra se mandan menos paquetes en cada iteración, por lo que se ocupará un bajo ancho de banda y se saturará menos la red.

El algoritmo de vector distancia, aunque tenga más iteraciones, manda menos paquetes que inundación también. Ocupa un ancho de banda más pequeño y similar al de Dijkstra.

• Deducciones

Teniendo en cuenta lo mencionado, existen una serie de problemas que pueden surgir al utilizar algunos de estos algoritmos.

Supongamos una red en la que los nodos están conectados consecutivamente, (cada uno con el siguiente y así sucesivamente).

En el caso de que caiga un nodo, el algoritmo de vector distancia, irá recalculando las distancias a todos los nodos y simplemente irá aumentando el peso pero nunca convergerá la distancia, hasta que haya un refresco de tablas internas.

# CONCLUSION

En esta practica hemos aprendido programar y evaluar el rendimiento de diversos algoritmos de enrutamiento. Hemos utilizados diversas implementaciones de la capa 3 del modelo OSI.

Hemos disfrutado del proceso de aprendizaje. La implementación de algoritmos es algo que a ambos nos causa interés. El hecho de ver el uso real de este algoritmo en las redes nos ha motivado a querer hacerlo de la mejor manera que éramos capaces, y enfrentarnos a un lenguaje nuevo para nosotros como es Matlab.

También nos ha ayudado entender de una manera mucho más especifica el funcionamiento de la capa 3 del modelo OSI, y la importancia de elegir el algoritmo de enrutamiento adecuado para cada situación.

Consideramos que esta práctica ha sido fundamental para entender el enrutamiento de la capa 3, y además, hemos aprendido la base de un nuevo lenguaje de programación.