Algoritmo de Estado de Enlace

Noe Requejo Mendoza 20100927

|  |  |  |
| --- | --- | --- |
| INFORMACIÓN DEL ARTÍCULO  Historial del Artículo  Recibido en Fecha 01-dic-2020  Palabras Clave:  Internet  Enrutamiento dinámico  Protocolo Estado de Enlace  Algoritmo Dijkstra |  | A B S T R A C T  En el siguiente artículo se expondrán los resultados de la experimentación realizada al evaluar el rendimiento del protocolo de Estado de Enlace en una topología de Red, para lo cual se realizó una recopilación de fuentes de información y se implementó el código en lenguaje Java en la plataforma NetBeans. Se implementó una interfaz para facilitar al usuario la elección de la topología preferida y finalmente se analizaron los resultados en base a los conocimientos obtenidos a lo largo del ciclo en el curso de Redes de Computadoras.  The following article will present the results of the experimentation carried out when evaluating the performance of the Link State protocol in a Network topology, for which a compilation of information sources was made and the code in Java language was implemented on the platform NetBeans. An interface was implemented to facilitate the user to choose the preferred topology and finally the results were analyzed based on the knowledge obtained throughout the cycle in the Computer Networks course. |

1. Introducción

Existe un largo camino evolutivo entre la manera de comunicarse de los primeros seres humanos hasta el día de hoy, en la historia cada evento permitió los cambios en la manera de comunicarnos, entre ellos se encuentra el Internet.

El Internet es la red mundial al a que están conectados todos los dispositivos existentes, esta permite la transmisión de información de un punto del globo a otro en cuestión de segundos, pero lo que se verá a continuación responde a la pregunta: ¿Cómo se transmite esta información a través de la red?

En el siguiente artículo se expondrán los tipos de enrutamiento existentes, enfatizando el enrutamiento dinámico con algunas ventajas y desventajas. Para continuar, se hablará del algoritmo de Estado de Enlace y se realizará una implementación en lenguaje Java de los escenarios planteados, el cuál se ejecutará en una interfaz.

1. Marco teórico

Los rúters son dispositivos de red que permiten la interconexión de dispositivos, almacenan distintas direcciones IP (las cuales son únicas e identifican a los dispositivos dentro de la red) y en base a estas realizan la transferencia de datos.

Los rúters requieren, para un funcionamiento básico, tablas de ruteo, las cuales se pueden configurar de dos maneras:

1. Enrutamiento Estático: Este tipo de enrutamiento tiene como objetivo indicarle al rúter que camino tomar para llegar de un punto a otro. Este camino es indicado por el usuario y no cambia a menos que el usuario lo configure nuevamente. Sirve cuando las topologías de red son menores y no se van a tener cambios en ella, como añadir o quitar un rúter.
2. Enrutamiento Dinámico: El enrutamiento dinámico, por otro lado, es una configuración que permite a los rúters dentro de la topología de red intercambiar las tablas de ruteo (encargadas de almacenar todos los dispositivos de la topología cada uno con su dirección IP) de tal manera que se actualizan los caminos posibles desde los distintos puntos de la topología todo el tiempo.

La mayor ventaja que se puede identificar en el enrutamiento dinámico frente al estático es que el primero mencionado se actualiza constantemente y obtiene a la vez distintos caminos posibles entre dos puntos de una topología, esto mismo permite que otorgue mayor escalabilidad a la red y se pueda aplicar a estructuras de red que es necesario cambiar constantemente (agregar o quitar rúters).

Por otro lado, existen también desventajas de utilizar este tipo de enrutamiento, una de las cuales es que para configurar este tipo de enrutamiento y manejar los posibles errores se necesita una persona con conocimientos del área para evitar complicaciones. Por otro lado, el enrutamiento estático no requiere de recursos del rúter ya que cualquier tipo de configuración es realizada directamente por el usuario, mientras que el enrutamiento dinámico si los utiliza. (CPU, memoria, etc.).

El algoritmo de Estado de Enlace es una manera en la que los rúters determinan qué camino es óptimo para llegar de un punto a otro dentro de la topología de red en la que se encuentran, este algoritmo utiliza la lógica del algoritmo de Dijkstra en grafos, por lo que buscará entre sus conexiones la de menor distancia administrativa (determinada por el protocolo aplicado a dicha conexión) y creará el camino de menor peso.

1. Estado del Arte

Para realizar este informe, se utilizó información extraída de las siguientes fuentes:

* 1. Demonstration of Single Link Failure Recovery using Bellman Ford and Dijikstra Algorithm in SDN

En este artículo se realiza una comparación entre los algoritmos Bellman Ford y Dijkstra al momento de recuperar la conexión tras la caída de uno de los nodos dentro de una topología utilizando el menor camino disponible. Para comenzar el investigador presenta las definiciones de ambos algoritmos y presenta el pseudocódigo de funcionamiento de cada uno para lo cual:

El algoritmo Dijkstra encuentra el menor algoritmo sumando todas las distancias que recopile de la topología sí y solo sí esta distancia es considerara un peso menor, el cual se instancia al inicio del programa como un número infinito (de esta manera el primer valor del arreglo diferente de 0 siempre será el menor valor).

El algoritmo Bellman Ford, por otro lado, considera las combinaciones de los pesos desde cada nodo ignorando los pesos de las conexiones individuales.

Se realiza una experimentación con una malla no dirigida de nodos los cuales representan los rúters, se implementa en cada uno un switch que se utilizará para inhabilitar uno de los nodos una vez que la conexión esté establecida.

Conclusión: La conclusión de la experimentación presentada es que el algoritmo de Dijkstra tiene un menor retraso al momento de evaluar la menor ruta si es que existe una interrupción de por medio, consideran que es un algoritmo más resistente a fallas ya que al calcular individualmente los pesos es más sencillo encontrar una ruta alternativa.

* 1. A survey paper of Bellman-ford algorithm and Dijkstra algorithm for finding shortest path in GIS application  
     Escritores: Vaibhavi Patel, Prof.ChitraBaggar

En este Segundo artículo, se presenta la implementación de los algoritmos Dijkstra y Bellman Ford en el área de transporte (informático, computacional), como propósito principal se va a analizar cuál de los algoritmos es mejor para implementarse en esta área.

De la misma manera se da inicio al artículo dando una explicación del funcionamiento de cada uno de los algoritmos con pseudocódigo y explicación textual paso a paso de su implementación. Luego, se comparan los algoritmos en función al tiempo y al espacio considerados, en base a esto se exponen similitudes y diferencias entre ambos algoritmos y finalmente se exponen las conclusiones del artículo.

Conclusión: Se llegó a la conclusión de que el algoritmo de Dijkstra es más veloz que el algoritmo Bellman pero a pesar de este resultado, no se utiliza mucho en el sistema GIS (transporte), el escritor expresa que el algoritmo podría funcionar de manera eficiente incluso con mayores distancias.

* 1. An Improved Dijkstra’s algorithm application to multi-core processors  
     Escritores: Qiong Wu, Guihe Qin, Hongliang Li

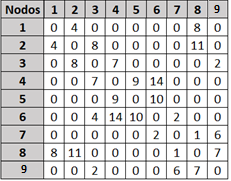
En este artículo lo que se busca es aplicar el algoritmo de menor camino Dijkstra en un entorno con procesadores multicore, de manera que todas las tareas del algoritmo Dijkstra que se puedan paralelizar, se ejecutarán de esa manera.

De la misma manera que los anteriores, se inicia con una breve explicación del algoritmo, los pasos textuales del funcionamiento y la exposición del pseudocódigo, la diferencia es que en este artículo a continuación se ingresa el pseudocódigo del Dijkstra paralelizado. A continuación, se muestran los resultados temporales de la experimentación para comparar el Dijkstra clásico (secuencial) y el paralelizado, se exponen los tiempos en una tabla y finalmente se habla de las herramientas utilizadas para lograr la experimentación realizada.

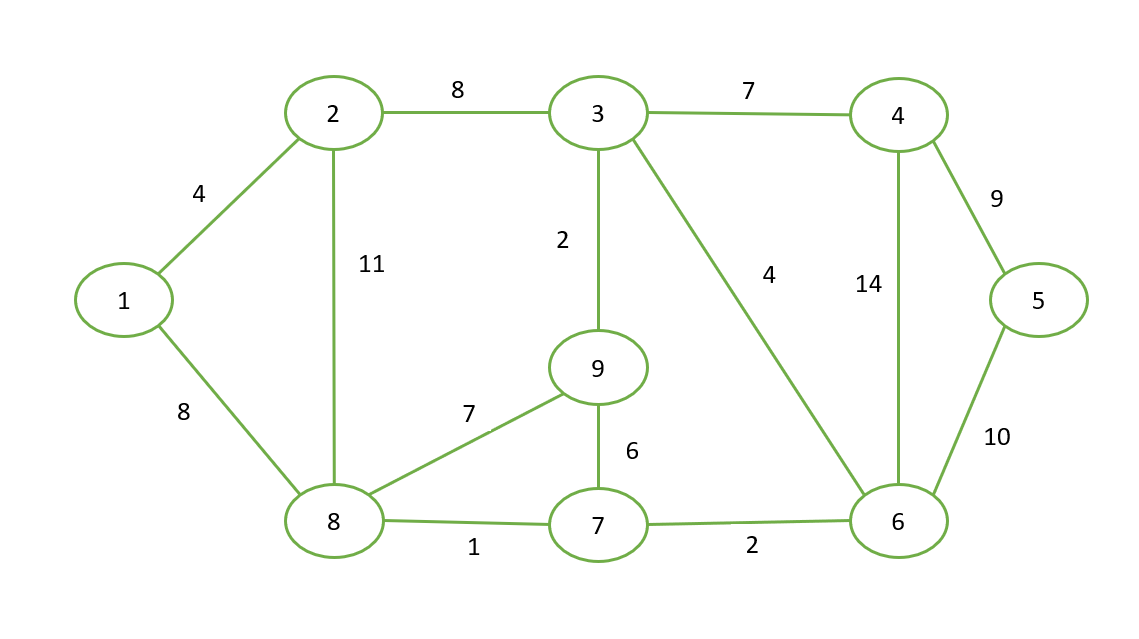
Conclusión: Al final del artículo el lector expone que, en base a la experimentación realizada, se puede llegar a la conclusión de que el cálculo y resolución de tareas no se logra solamente optimizando el algoritmo, sino que también se debe mejorar la arquitectura de hardware. Con el algoritmo planteado, se puede mejorar el rendimiento hasta en un 50% cuando se habla de una gran cantidad de nodos.

1. Implementación
   1. Escenario 1

El escenario 1 representa una topología de 9 rúters conectados entre sí con 14 aristas de peso variable expresada de la siguiente manera en una matriz de adyacencia:

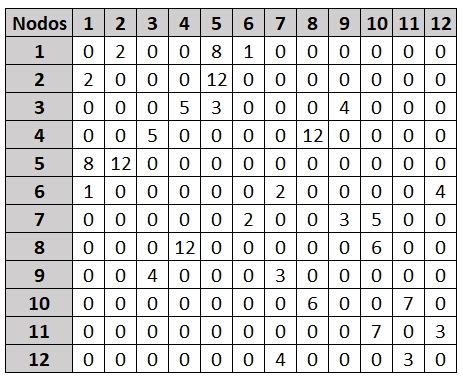


Gráficamente se representa de la siguiente manera:

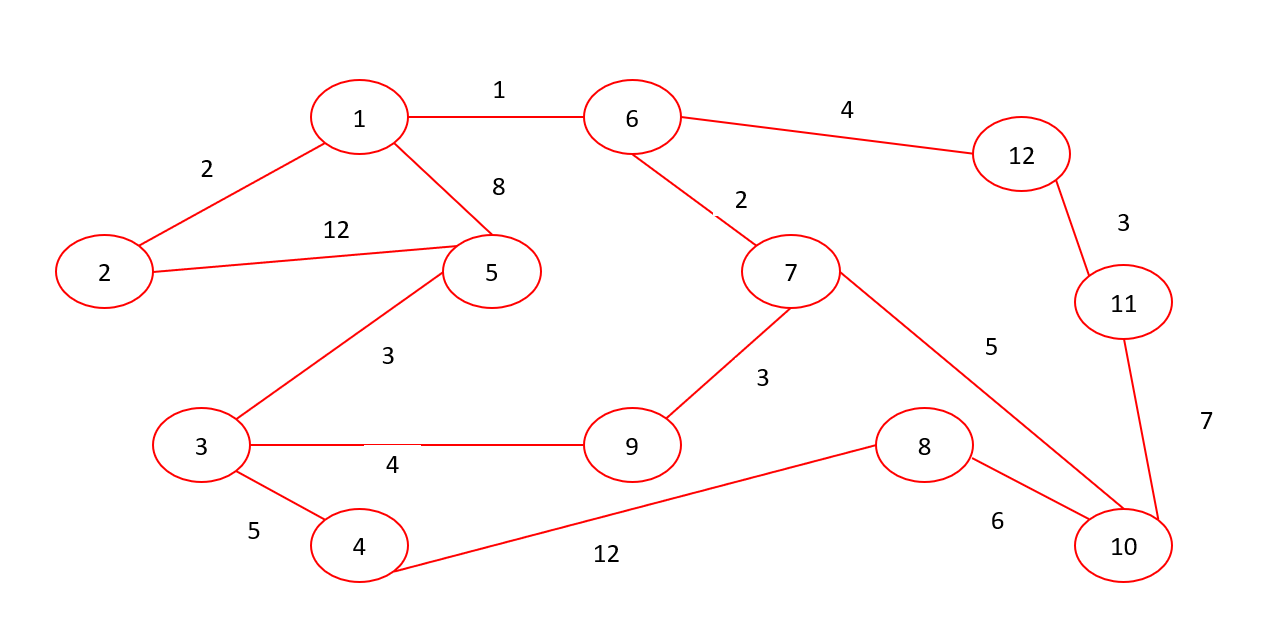


* 1. Escenario 2

El escenario 2 representa una topología de 12 rúters conectados entre sí con 15 aristas de peso variable expresada de la siguiente manera en una matriz de adyacencia:

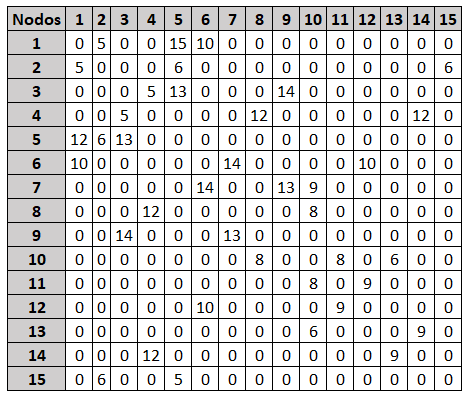


Gráficamente se representa de la siguiente manera:

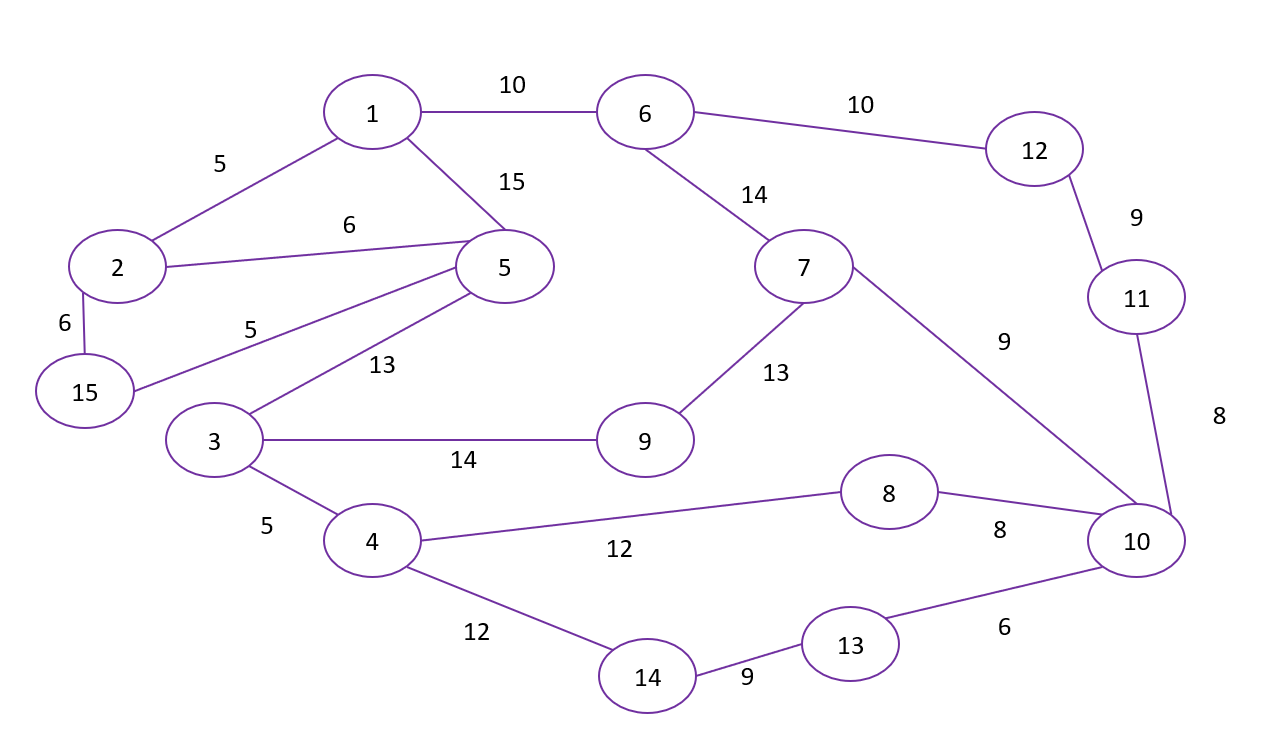


* 1. Escenario 3

El escenario 3 representa una topología de 15 rúters conectados entre sí con 20 aristas de peso variable expresada de la siguiente manera en una matriz de adyacencia:



Gráficamente se representa de la siguiente manera:



1. Experimentación

Para todos los escenarios, nos basaremos en los conocimientos del método Dijkstra y la metodología señalada por el portal GeeksforGeeks (Hasija & Aakash, s.f.). Tomando como base a este portal, se ha modificado el código para poder formular una serie de escenarios que irán creciendo en dificultad.

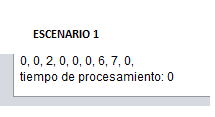
Básicamente, el código lo que hace es crear una matriz de adyacencias de un grafo de dimensión NxN, donde N es el número de vértices, iniciando todos los pesos en 0. Luego, la matriz tendrá conexiones en cada uno de la matriz[x][y] que tengan peso

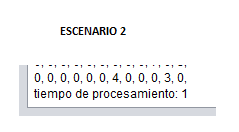
Por último, el algoritmo recorrerá dicha matriz y creará otro vector “distancia\_minima”. La primera interacción será el origen y de ahí , ira recorriendo todos los vértices, y poniendo en el vector “distancia\_minima” la distancia encontrada

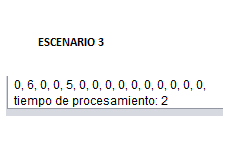
Si una distancia dentro del vector “distancia\_minima” es menor a la nueva distancia encontrada por otro camino a un vértice, entonces, el vector “distancia\_minima se actualizará con la menor distancia mínima encontrada

Al final de saltar por todos los vértices, se imprimirá en pantalla la distancia mínima encontrada desde todos los vértices al origen

Comparando los escenarios, también se obtiene lo siguiente





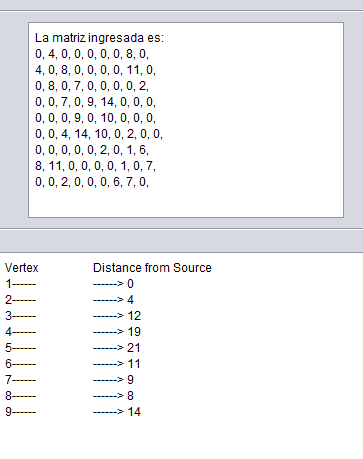


Se puede entonces, verificar que mientras más crezca el número de redes en la red, mayor tiempo de procesamiento será. Sin embargo, estos tiempos son en milisegundos, por lo cual, el tiempo de procesamiento es verdaderamente pequeño, haciendo muy eficiente nuestro programa.

Otra observación obtenida en base a la experimentación, es que, se debe seguir al pie de la letra las instrucciones del archivo txt. Ya que será el txt de donde nuestro programa leerá los datos para llenar la matriz que procesará nuestro programa y que, simulará nuestra red. Comparado con otros programas, la presente investigación cumple con lo solicitado, pero le faltaría poner excepciones de no poner números negativos, datos alfabéticos o nulos en la matriz de adjacencia.

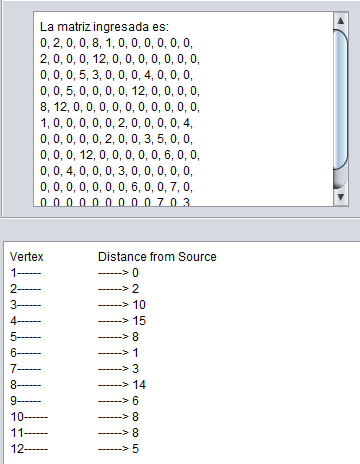
* 1. Experimentación con el Escenario 1

Como se pudo observar, la topología presentada en el escenario 1 tiene conexiones de peso variable, pero con valores de rango bajo. Nuestro algoritmo tendrá que encontrar el camino más corto, de manera sencilla.

Al hacer clic en el botón “Escenario 1” se verá la red que procesaremos, y abajo, las rutas más cortas desde el origen.

* 1. Experimentación con el Escenario 2

En este escenario 2, se tomará en cuenta una topología de nodos conectados con aristas de peso variable y en un rango ligeramente elevado. La red a evaluar será la mostrada en la imagen. Diferentes nodos conectados con sus diferentes pesos. Al hacer clic en “Escenario 2”, el sistema tomará la red mostrada en la imagen, y mostrará los caminos más cortos en la parte de abajo.



* 1. Experimentación con el Escenario 3

Para el escenario 3 se toma en cuenta una topología parecida a la presentada en el escenario 2, con la diferencia de que aquí se presenta un mayor número de nodos y con pesos considerablemente mayores. Al hacer clic en “Escenario 3”, el sistema tomará la topología presentada en la imagen anterior y mostrará el menor camino a cada nodo en la parte inferior del programa.

1. Conclusiones

En base a los datos analizados, podemos analizar que, mediante la investigación actual, el método de “Estado de Enlace” se puede lograr basar exitosamente en los conocimientos de algoritmo Dijkstra

Se concluye también que, dependiendo del método para ingresar los datos de tu matriz (teclado, mouse, archivo, etc) puede aumentar el tiempo para procesar diferentes escenarios

Se concluye también que, a mas nodos en la red, mayor el tiempo de procesamiento. Sin embargo, sigue procesando de manera súper rápida, a ver que las pruebas lanzan tiempos de 0,1 y 2 microsegundos

Por ultimo, se concluye que el método Dijsktra es el mas efectivo comparado con el método de vector distancia, puesto que no debemos guardar toda la información en todos los nodos, sino tan solo un solo vector con las mínimas distancias encontradas, así como el nodo origen y el tamaño de la matriz NxN que creará

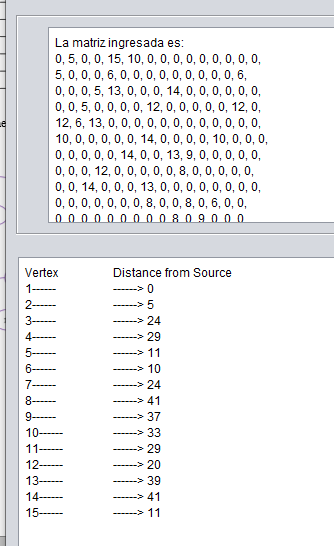
1. Referencias

Hasija, A., & Aakash, H. (s.f.). *www.geeksforgeeks.com*. Recuperado el 10 20, 2020, de https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Patel, V., & Baggar, C. (2014). *A survey paper of Bellman-ford algorithm and Dijkstra algorithm for finding shortest path in GIS application.* Gujarat: International Journal of P2P Network Trends and Technology (IJPTT).

Waleed, S., Faizan, M., Iqbal, M., & Anis, M. I. (2017). *Demonstration of Single Link Failure Recovery using Bellman Ford and Dijil(stra Algorithm in SDN.* Pakistan: IEEE.

Wu, Q., Qin, G., & Li, H. (2015). *An Improved Dijkstra’s algorithm application to multi-core processors.* Changchun: Metal Journal.

1. Anexos

Link Github: <https://github.com/nrequejom/trabajoRedes20100927>

PSEUDOCODIGO

Clase “Método más corto” donde están todos los métodos del algoritmo principal

METODO 1:

En\_que\_vertice\_estoy\_ahora (Vector distancia, vector ValidacionVertice, numero de Vertices):

INICIO

varialble minimaDistancia = infinito

FOR (desde 0 hasta numero de vertices )

SI vectorValidacionVertice = falso y vectorDistancia <= minimadistancia

minimaDistancia = vectorDistancia

minimoIndice = verticeActual

FIN SI

FIN FOR

FIN

Comentario: Este método trabaja, iniciando a todos los vectores distancias en infinito. Mas adelante veremos que, al vértice origen le ponemos como distancia hacia origen = 0, por lo cual, nuestro algoritmo comparará minimaDistancia vs 0 y verá que el vector iniciado como 0 es el origen

METODO 2: METODO DE ALGORITMO DE DIJKSTRA

Dijkstra (grafo NxN, origen, numeroVertices)

INICIO

Declarar VectorValidacion

Declarar VectorMinimaDistancia

FOR(desde 0 hasta el numeroVertices)

vectorMinimaDistancia = infinito

vectorValidación = FALSO

FIN\_FOR

vectorMinimaDistancia (origen) = 0

FOR(desde 0 hasta el numeroVertices- 1)

Declarar verticeActual

Ejecutar “En\_Que\_Vertice\_Esto\_ahora (vectorMinimaDisntancia, vectorValidacion, numeroVertices

vectorValidacion (de verticeActual) = VERDADERO

FIN\_FOR

FOR (desde 0 hasta numeroVertices)

SI(vectorValidacion y grafo existen

Y vectorMinimaDistancia(actual) no es infinito

Y vectorMinimaDistancia (actual) + VectorDistancia(al origen) es menor al vectorDistancia (actuak)

Entonces: VectorMinimaDistancia = vectorMinimaDistancia (actual) + VectorDistancia(al origen)

FIN\_SI

FIN\_FOR

Retorna el vectorMinimaDistancia

FIN

Comentario: Lo que indica este método Dijkstra, es que

Declararemos la distancia desde el origen al origen como 0, para luego

Correr el algoritmo En\_que\_vertice\_estoy\_ahora y que nos encuentre el

Numero de nodo hacia el origen

Luego, recorrerá los vértices que no son el origen para encontrar

Su distancia al origen, y guardarla, pero en caso ya existiese una distancia

al origen guardada anteriormente y que fuera mayor a la encontrada, entonces se descartra ese dato guardado y se lo reemplaza por el actual

El resto de métodos son ordinarios métodos para imprimir el arreglo minimaDistancia desde el origen y , otro metodo para leer los txt que importamos, pero no contribuyen a modificar el grafo ya obtenido

CODIGO DEL PROGRAMA

METODOS DE ALGORITMO DE DIJKSTRA

package Dijkstras;

import java.util.\*;

import java.lang.\*;

import java.io.\*;

public class CaminoMasCorto {

public CaminoMasCorto() {

}

int Metodo\_En\_Que\_Vertice\_Estoy\_Ahora(int dist[], Boolean[] vectorValidaciondeVertice, int verticesEscenarios) {

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index = -1;

for (int v = 0; v < verticesEscenarios; v++) {

if (vectorValidaciondeVertice[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

public String ImprimirVectorCaminoMasCorto(int dist[], int verticesEscenarios) {

String respuesta = "Vertice\tDistancia desde el origen";

//System.out.println("Vertex \t\t Distance from Source");

for (int i = 0; i < verticesEscenarios; i++) {

if (dist[i] == Integer.MAX\_VALUE) {

respuesta = respuesta + "\n" + (i + 1) + "------\t------> Vertice no conecta con origen";

} else {

respuesta = respuesta + "\n" + (i + 1) + "------\t------> " + dist[i];

//System.out.println("\n" i + " \t\t " + dist[i]);

}

}

return respuesta;

}

public int[] Algoritmo\_Dijkstra(int[][] grafo, int origen, int verticesEscenarios) {

int[] VectorDistanciaMinima = new int[verticesEscenarios]; // The output array. dist[i] will hold

Boolean[] VectorValidaciondeVertice = new Boolean[verticesEscenarios];

// Initialize all distances as INFINITE and stpSet[] as false

// Definimos todas las distancias como infinito. Definimos el 3

for (int i = 0; i < verticesEscenarios; i++) {

VectorDistanciaMinima[i] = Integer.MAX\_VALUE;

VectorValidaciondeVertice[i] = false;

}

// Distance of source vertex from itself is always 0

VectorDistanciaMinima[origen] = 0;

for (int count = 0; count < verticesEscenarios - 1; count++) {

int Vertice\_Actual = Metodo\_En\_Que\_Vertice\_Estoy\_Ahora(VectorDistanciaMinima, VectorValidaciondeVertice, verticesEscenarios);

VectorValidaciondeVertice[Vertice\_Actual] = true;

for (int i = 0; i < verticesEscenarios; i++) // Update dist[v] only if is not in sptSet, there is an

{

if (!VectorValidaciondeVertice[i] && grafo[Vertice\_Actual][i] != 0 && VectorDistanciaMinima[Vertice\_Actual] != Integer.MAX\_VALUE && VectorDistanciaMinima[Vertice\_Actual] + grafo[Vertice\_Actual][i] < VectorDistanciaMinima[i]) {

VectorDistanciaMinima[i] = VectorDistanciaMinima[Vertice\_Actual] + grafo[Vertice\_Actual][i];

}

}

}

// print the constructed distance array VER DE BORRAR

//System.out.println(printSolution(dist)); VER DE BORRAR

return VectorDistanciaMinima;

}

}

METODO PARA LEER EL ARCHIVO Y CREAR LA MATRIZ A CORRER POR EL OTRO METODO ANTES EXPLICADO

{

long startTime = System.currentTimeMillis();

String cadena = "";

String cadenaux = "";

String cadenaux2 = "";

String matriz = "La matriz ingresada es: ";

int[][] grafo3 = new int[0][0];

int tamaño = 0;

int origen = 0;

int aux = 0;

int aux2 = 0;

try {//primero: definimos la ubicación del archivo en una variable

//ademas, leemos el archivo, con la clase BufferReader

FileReader ubicacion = new FileReader("C:\\Users\\USER\\Desktop\\2020-2\\Redes\\Trabajo\\prueba3.txt");

BufferedReader lector = new BufferedReader(ubicacion);

while ((cadena = lector.readLine()) != null) {

for (int i = 0; i < cadena.length(); i++) {

cadenaux = cadenaux + String.valueOf(cadena.charAt(i));

}

if (aux == 14) {

tamaño = Integer.parseInt(cadenaux);

//creamos el grafo de tamaño que extraigamos del txt

grafo3 = new int[tamaño][tamaño];

for (int i = 0; i < tamaño; i++) {

for (int j = 0; j < tamaño; j++) {

grafo3[i][j] = 0;

}

}

//----------------------------------------------------

//a partir de la linea 12, comienza a leer los pesos de los caminos

} else if (aux == 15) {

origen = Integer.parseInt(cadenaux);

} else if (aux > 15) {

int aux3 = 0;

for (int k = 0; k < cadena.length(); k++) {

if (cadena.charAt(k) != 46) {

cadenaux2 = cadenaux2 + String.valueOf(cadena.charAt(k));

} else if (cadena.charAt(k) == 46) {

// System.out.print(cadenaux2);

grafo3[aux2][aux3] = Integer.parseInt(cadenaux2);

aux3++;

cadenaux2 = "";

}

}

aux2++;

}

aux++;

cadenaux = "";

}

//System.out.println(aux);

} catch (Exception e) {

e.printStackTrace();

}

for (int i = 0; i < tamaño; i++) {

matriz = matriz + "\n";

for (int j = 0; j < tamaño; j++) {

matriz = matriz + grafo3[i][j] + ", ";

}

}

//int dimensiones = this.getContentPane().getSize().width / 2 - this.Imagen.getSize().width / 2;

// JOptionPane.showMessageDialog(null, dimensiones);

// JOptionPane.showMessageDialog(null, Imagen.getSize().width);

CaminoMasCorto t = new CaminoMasCorto();

this.AreaTextodeRespuesta.setText(t.ImprimirVectorCaminoMasCorto(t.Algoritmo\_Dijkstra(grafo3, origen, tamaño), tamaño));

long endTime = System.currentTimeMillis() - startTime;

matriz = matriz + "\ntiempo de procesamiento: " + endTime;

this.AreaTextodeRespuestaMatriz.setText(matriz);

ImageIcon ImagenAux = new ImageIcon("Escenario3.png");

EtiquetaImagen.setIcon(

new javax.swing.ImageIcon(ImagenAux.getImage().getScaledInstance(this.EtiquetaImagen.getWidth(), this.EtiquetaImagen.getHeight(), Image.SCALE\_SMOOTH)));

}