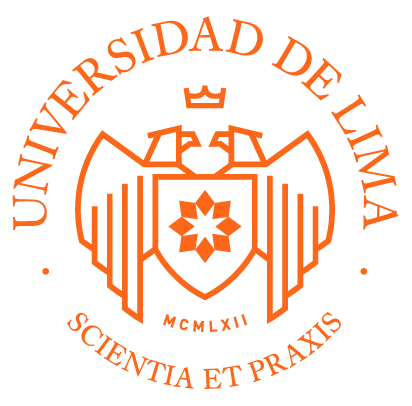
Universidad de Lima

Facultad de Ingeniería y Arquitectura

Carrera de Ingeniería de Sistemas



ALGORITMO DIJKSTRA

Diego Nicolas Ipanaque Ballon

**Código** 20170754

**Asesor**

Carlos Martin Torres Paredes

Lima – Perú

Diciembre de 2020

Contenido

[Resumen 3](#_Toc57384871)

[Introducción 3](#_Toc57384872)

[Estado de arte 3](#_Toc57384873)

[Propuesta de implementación 4](#_Toc57384874)

[Experimentación 8](#_Toc57384876)

[Análisis comparativo 10](#_Toc57384877)

[Conclusiones 11](#_Toc57384878)

[Referencias bibliográficas 11](#_Toc57384879)

[Anexo 11](#_Toc57384880)

**Resumen:**

El presente trabajo fue hecho de manera individual, este tiene el propósito de implementar el algoritmo de Dijkstra en un lenguaje de preferencia a fines de poderlo aplicar en diferentes escenarios.

El algoritmo de Dijkstra esta diseñado para buscar el camino o ruta mas corta entre un punto origen y uno destino. El algoritmo diseña una ruta partiendo de un origen con una meta de recorrer la menor distancia posible, esto se puede demostrar de una forma mas sencilla empleando nodos ya que estos al tener un peso podemos mapear cual es la ruta que tiene menor peso para llegar al nodo destino.

Se ha implementado el lenguaje de programación Java usando Netbeans como entorno de desarrollo. Este algoritmo creado se desarrolla con 3 escenarios.

**Introducción:**

La optimización es algo que todos los ingenieros buscamos implementar en la actualidad, ya que es la forma de dar una solución de forma practica a los problemas que se nos presenten.

En la ingeniería de sistemas, en el área de redes se busca siempre optimizar el servicio que se le brinda a los usuarios por ende siempre buscamos reducir los tiempos de respuesta, reducir el uso del ancho de banda, bajar latencia, etc.

Es por esto que se implemento el algoritmo de Dijkstra para optimizar las distancias de un punto a otro, este algoritmo puede brindar múltiples soluciones a diferentes problemas, pero nosotros en el campo de redes buscamos darle una similitud a los problemas que encontramos como lo suele ser buscar la ruta mas curta de un Router a otro de tal forma que siempre tengamos la ruta más efectiva.

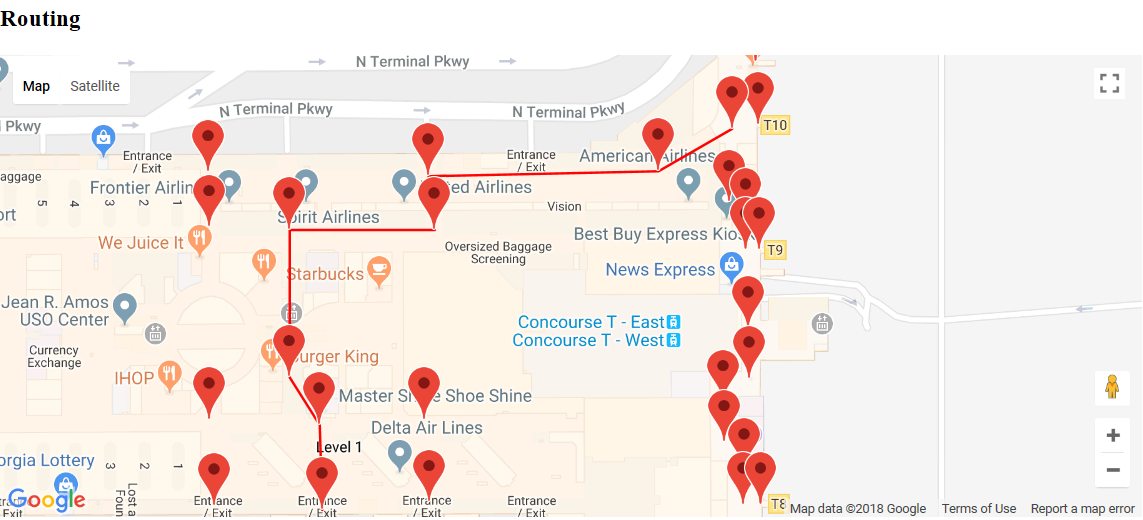
**Estado de arte:**

En un grafico en el cual se presente un grafo con sus pesos se puede definir el camino más rápido o la ruta de coste mínimo en el cual tenemos un nodo de origen (o) a un nodo destino (d) en el cual la suma de las distancias por el cual se va haciendo el recorrido puede compararse para definir la ruta optima.

El algoritmo que se plantea implementar (Dijkstra) tiene una complejidad cuadrática(0(n2)) siendo eficaz para la búsqueda de caminos en diferentes grafos.

Para entender como funciona este algoritmo se planteará lo siguiente:

Un grafo este compuesto de vértices y un peso el cual se denomina como G(v,p). cada vértice tiene una distancia con otro lo cual se denomina L(v). Para poder generar la menor ruta se crea una lista la cual almacena los nodos y sus distancias. Con un inicio 0 para que luego sea comparado con la suma de las rutas que se comparan y la que este mas cerca al menor número después de las múltiples comparaciones de rutas será tomado como ruta optima.

El algoritmo de Dijkstra al ser efectivo para resolver diversos problemas se ha implementado en el transcurso del tiempo en diferentes aplicaciones, el ejemplo mas cercano que podemos encontrar al curso que tenemos por el momento seria la forma de minimizar el recorrido de un archivo por la red. Por otro lado lo encontramos en otras aplicaciones usualmente en servicios que tienden a tener trafico como: Wazer, Google maps, o hasta las rutas de un tren, las cuales buscan que el usuario que emplee la aplicación pueda optar la ruta más rápida para llegar a su destino y esto se debe a la recolección de datos que tiene la aplicación guiándose de un parámetro ingresado por el usuario para realizar la comparación de rutas posibles.

Fuente: Github-Dijkstra and Google Maps.(28 Dec 2019)

**Propuesta de implementación:**

El algoritmo de Dijkstra implementado se compone de 4 funciones principales: setgrafo, dijkstra, dijkstra\_siguiente, ImprimirResultado y mostrargrafo.

**Función setgrafo:**

En esta función se pide el numero de vértices que tendrá el grafo, las aristas, el nodo origen y por último el nodo destino.

System.out.println("Cuantos vertices tiene?");

nodo = ve.nextInt();

System.out.println("cuantas aristas tiene?");

aristas = ve.nextInt();

System.out.println("Ingrese el origen:");

origen = ve.nextInt();

System.out.println("Ingrese el destino");

destino = ve.nextInt();

Luego se hace un mapeado en el grafo para asignar cuando es el peso que tiene el dirigirse de un nodo a otro para luego hacer un mapeado en el grafo.

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

System.out.println("Desde:");

int desde = ve.nextInt();

System.out.println("hacia:");

int hacia= ve.nextInt();

System.out.println("peso:");

int peso= ve.nextInt();

grafo[desde][hacia] = peso;

}

siguiente = new int[nodo + 1];

dis = new double[nodo + 1];

**Función Dijkstra:**

En esta parte de la función se va realizar el mapeado del grafo para saber cual es el nodo que siguiente y capturar la distancia y luego compararla.

Se asigna una distancia a [j] de 10000 al ser un valor grande facilitara la comparación. Y la distancia origen 0 ya que es donde esta empezando la ruta optima.

public void dijkstra() {

for(int j=1; j<dis.length; j++)

{

dis[j]=10000;

}

dis[origen] = 0;

for(int i= 1; i<= nodo;i++){

queue.add(i);

}

dijkstra\_siguiente();

**Función dijkstra\_siguiente:**

En esta parte de la función con el uso de un while se hace un bucle para saber para capturar el valor de la suma de un vértice a su adyacente, aparte este bucle finalizara una vez esta completo o se podría decir que este ya no tenga otro adyacente para seguir acumulando un peso.

public void dijkstra\_siguiente(){

while (!queue.isEmpty()) {

priority.setpriority(queue, dis);

int x = (int) queue.poll();

for (int verticeadyacente = 1; verticeadyacente < grafo[0].length; verticeadyacente ++){

if(grafo[x][verticeadyacente] > 0 ){

if(dis[verticeadyacente] > dis[x] + grafo[x][verticeadyacente]){

dis[verticeadyacente] = dis[x] + grafo[x][verticeadyacente];

siguiente[verticeadyacente] = x;

}

}

}

}

}

**Función ImprimirResultado:**

En esta parte del proyecto se imprime la distancia del nodo origen hacia el nodo destino. Se utiliza un for para recorrer la longitud de la distancia y un while para hacer el ultimo mapeo del grafo e imprimir la distancia optima de la ruta propuesta por el usuario.

public void ImprimirResultado(){

for(int z= 5; z < dis.length; z++){

System.out.println("distancia de " + origen + " hacia " + z + " : " + dis[z]);

}

int dest = destino;

String camino = " " + dest;

while(true){

camino = siguiente[dest] + "------->" + camino;

dest= siguiente[dest];

if(dest == origen){

break;

}

}

System.out.println(camino);

}

**Función mostrargrafo:**

Esta es la última parte del código implementado, ya que sirve para visualizar la ruta que se dio en el grafo propuesto por el usuario. Se utiliza un for para hacer el ultimo recorrido, pero en el grafo general e imprimir nodo por nodo hasta terminar la ruta optima.

public void mostrargrafo(){

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

for (int j=0; j < grafo[0].length; j++){

System.out.println(grafo[i][j] + " ");

}

System.out.println();

}

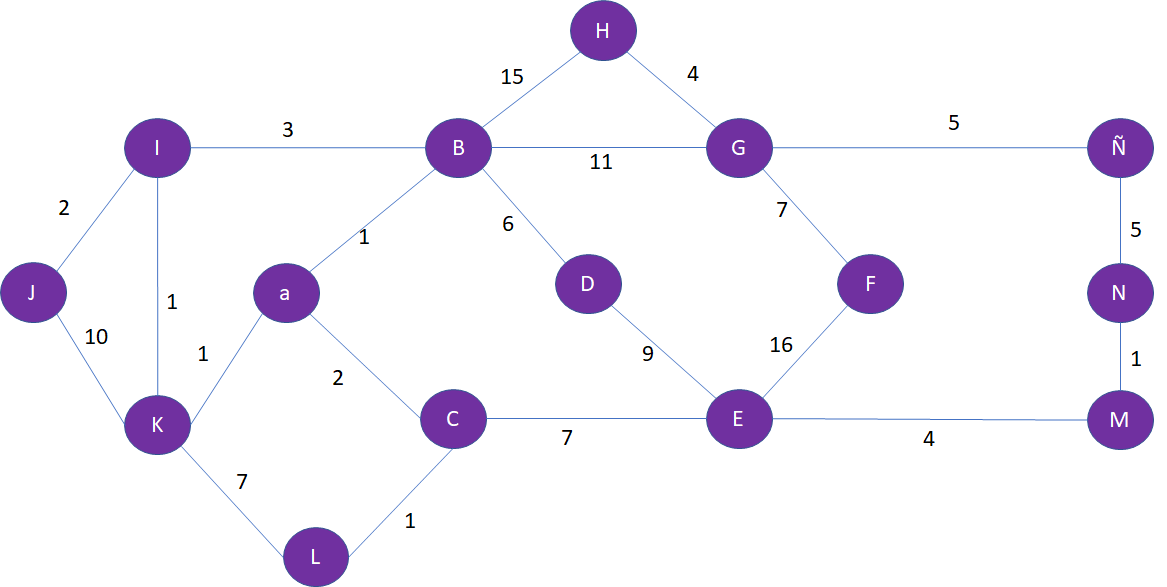
}

}

**Experimentación:**

En el código que implemente el usuario será aquel que llene los datos del escenario que desee, en este caso se utilizo un grafo de 15 vértices y 21 aristas con diferentes pesos el cual busca conseguir la ruta más rápida del nodo origen hacia un nodo destino a elección propia ambos casos del usuario, en este caso del nodo i al nodo m.

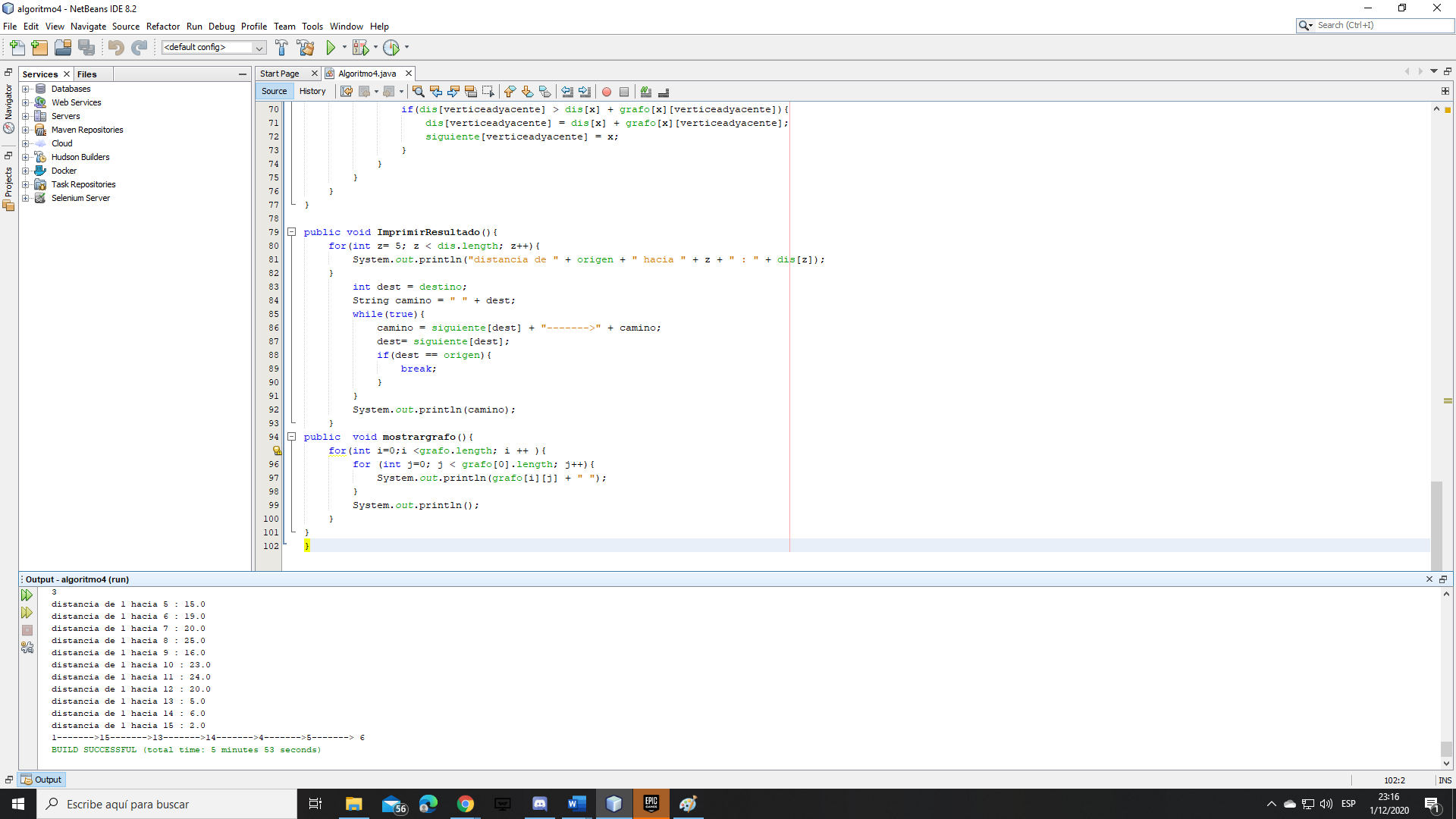
Para que el escenario sea sencillo de entender se presentara este gráfico:



Fuente: Elaboración propia

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Ejemplo 3) |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| **LINK STATEPACKETS** | **A** | **B** | **C** | **D** | **E** | **F** | **G** |
|  | C/2 | H/15 | E/7 | E/9 | F/16 | E/16 | B/11 |
|  | K/1 | G/11 | A/2 | B/6 | D/9 | G/7 | F/7 |
|  | B/1 | D/6 | L/1 |  | C/7 |  | Ñ/5 |
|  |  | A/1 |  |  | M/4 |  | H/4 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **H** | **I** | **J** | **K** | **L** | **M** | **N** | **Ñ** |
| B/15 | B/3 | K/10 | J/10 | K/7 | E/4 | Ñ/5 | G/5 |
| G/4 | J/2 | I/2 | I/1 | C/1 | N/1 | M/1 | N/5 |
|  | K/1 |  | A/1 |  |  |  |  |
|  |  |  |  |  |  |  |  |



Generando un resultado en el cual la ruta mas rapida de i a m es de un peso de 19 como se presenta en el codigo.

Lo que hace en codigo en este caso es capturar los datos ingresados por el usuario:vertices, aristas y peso para luego realizar un mapeado del nodo origen con peso 0 hasta el nodo destino el cual ira comparando el acumulativo de rutas de menor costo hasta obtener la ruta mas eficiente para el problema propuesto.

**Analisis comparativo:**

El codigo implementado esta diseñado para grafos pequeños, en este caso un grafo con de 15 vertices el cual no es complicado llenar los datos que pide el codigo para obtener un resultado.

En resumen se podria decir que para escenarios pequeños el codigo funciona de forma correcta y eficiente ya que nos muestra la ruta que se realiza para poder llegar al destino y nos da el peso u coste para llegar a un destino a este desde el origen que nosotros podamos proponer. Si este codigo se quisiera optimizar se podria implementar la lectura de mas datos de un excel, logrando hacer un mapeo de unos grafos mas grandes. Lo que se diferenciaria del codigo que se propuso con esta idea es que al implementar un excel con los datos de un grafo de muchos vertices haria que los datos sean fijos y solo realicen una lectura de estos, mientras aquí el usuario puede cambiar llenar los datos como el desee. De igual forma podemos decir que depende del escenario o problema que tengamos, si el grafo es grande lo ideal seria utilizar un excel para poder hacer un mapeado de estos datos de forma rapida y eficiente pero si es un grafo pequeño, este codigo puede resultar mas comodo de utilizar y entender.

**Conclusiones:**

En conclusion el algoritmo de dijkstra creado en java cumple con el problema propuesto, y podemos decir que el codigo es moldeable a diferentes problemas y escenarios de la vida real, si se tendria una base de datos con rutas se podria hacer un mapeado grande como los aviones al realizar una escala de un punto origen a otro hasta llegar a su destino o el caso mas habitual el waze que mide la ruta mas corta para que el usuario llegue a su destino, este tipo de algoritmos puede facilitar muchos problemas en la forma que lo implementemos y se ha logrado asociar con lo presentado en clase ya que estos problemas tambien se encuentra en el trafico que hay en una red.

**Referencias bibliograficas:**

* <https://github.com/venkat-abhi/Graph-routing-using-dijkstra-shortest-path-algorithm-and-google-maps/blob/master/README.md>
* Mohammad Enayattabar, tomo 5, N°2 , (Junio 2019). Dijkstra algorithm for shortest path problem under interval-valued Pythagorean fuzzy environment Mohammad Enayattabar, tomo 5, N°2 , (Junio 2019)
* Rafael Rodriguez Puente, Volumen 38 N°1 (enero 2017). Busqueda de caminos minimos haciendo uso de grafos reducidos.
* Gloria Sanchez Torrubia, marzo 2012.Algoritmo de dijkstra.

**Anexo:**

package algoritmo4;

import java.util.Scanner;

import java.util.LinkedList;

import java.util.Queue;

public class Algoritmo4 { //partes del grafo.

Scanner ve = new Scanner(System.in);

Queue<Integer> queue = new LinkedList<>();

double dis[];

int nodo , aristas;

double grafo[][];

int siguiente[];

int origen, destino;

public void setgrafo(){ //se le pide al usurio ingresar los datos del grafo.

System.out.println("Cuantos vertices tiene?");

nodo = ve.nextInt();

System.out.println("cuantas aristas tiene?");

aristas = ve.nextInt();

System.out.println("Ingrese el origen:");

origen = ve.nextInt();

System.out.println("Ingrese el destino");

destino = ve.nextInt();

grafo = new double[nodo + 1] [ nodo + 1];

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

System.out.println("Desde:");

int desde = ve.nextInt();

System.out.println("hacia:");

int hacia= ve.nextInt();

System.out.println("peso:");

int peso= ve.nextInt();

grafo[desde][hacia] = peso;

}

siguiente = new int[nodo + 1];

dis = new double[nodo + 1];

}

public void dijkstra() {

for(int j=1; j<dis.length; j++)

{

dis[j]=10000;

}

dis[origen] = 0;

for(int i= 1; i<= nodo;i++){

queue.add(i);

}

dijkstra\_siguiente();

ImprimirResultado();

}

public void dijkstra\_siguiente(){

while (!queue.isEmpty()) {

priority.setpriority(queue, dis);

int x = (int) queue.poll();

for (int verticeadyacente = 1; verticeadyacente < grafo[0].length; verticeadyacente ++){

if(grafo[x][verticeadyacente] > 0 ){

if(dis[verticeadyacente] > dis[x] + grafo[x][verticeadyacente]){

dis[verticeadyacente] = dis[x] + grafo[x][verticeadyacente];

siguiente[verticeadyacente] = x;

}

}

}

}

}

public void ImprimirResultado(){

for(int z= 5; z < dis.length; z++){

System.out.println("distancia de " + origen + " hacia " + z + " : " + dis[z]);

}

int dest = destino;

String camino = " " + dest;

while(true){

camino = siguiente[dest] + "------->" + camino;

dest= siguiente[dest];

if(dest == origen){

break;

}

}

System.out.println(camino);

}

public void mostrargrafo(){

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

for (int j=0; j < grafo[0].length; j++){

System.out.println(grafo[i][j] + " ");

}

System.out.println();

}

}

}