

Colegio Intisana

000379

Matemáticas Nivel Medio

Rutas óptimas utilizando teoría de

Grafos

Lascano Valarezo Luis Enrique

000379-0026

Mayo 2019

Número de páginas: 12

## Tabla de contenidos

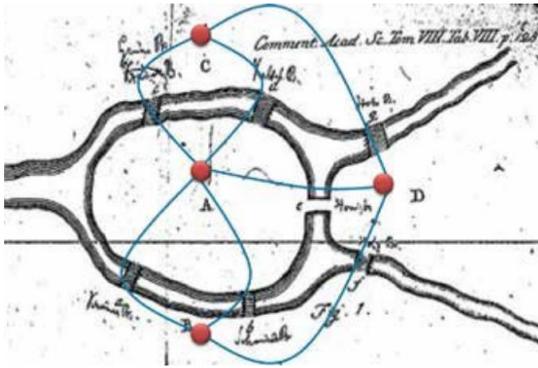
1	Introducción.....	1
2	Objetivos .....	3
3	Conceptos básicos de grafos .....	3
4	El camino Euleriano .....	4
5	Ciclos y caminos hamiltonianos .....	6
5.1	Teorema de Dirac .....	6
5.2	Teorema de Ore.....	6
6	El problema camino más corto y su algoritmo.....	6
7	Implementación del algoritmo.....	10
7.1	Pseudocódigo .....	10
8	Conclusiones.....	12
9	Bibliografía .....	13
10	Anexo .....	14
10.1	Código en JAVA.....	14
10.1.1	Código clase InicialAppCMC.....	14
10.1.2	Código clase Grafo.....	18
10.2	Ejecución .....	24
10.2.1	Primer resultado Grafo 1 .....	29
10.2.2	Primer resultado Grafo 2 .....	32

## 1 Introducción

Cuando recorro mi ciudad junto con mi familia ya sea por paseo o por salir de compras o diversión, he notado que mis padres siempre buscan rutas rápidas y cortas para llegar a nuestro destino. Con el tráfico creciente cada año, hemos ido cambiando nuestras rutas para evitar vías con muchos semáforos o vías que se convierten en áreas ocupadas por alto comercio o por ser salidas de la ciudad.

Pensando en soluciones matemáticas a este problema, he encontrado un estudio vasto en la teoría de grafos en donde se plantean trayectos como el de la ruta de un autobús buscando llegar a todas las paradas necesarias utilizando rutas óptimas o la de un cartero recorriendo la ciudad y visitando los buzones con el menor esfuerzo posible.

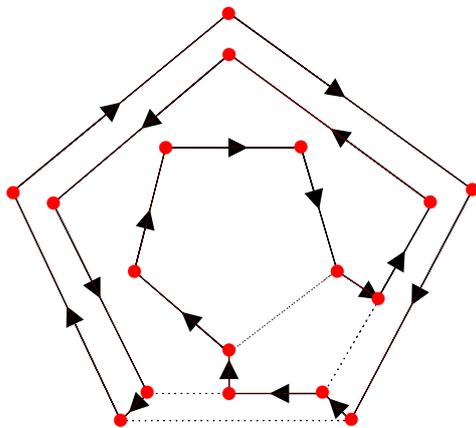
La teoría de grafos se utiliza en muchas aplicaciones desde la modelación de un circuito electrónico, redes neuronales, arquitectura y juegos entre otros y esto se logra en base a una serie de teoremas y aportes de diferentes matemáticos a través del tiempo. Uno de los primeros aportes vino del suizo Leonhard Euler en 1736 quien al visitar la ciudad de Königsberg y ver sus siete puentes que conectaban diversas partes de la ciudad se planteó una ruta turística para cruzar todos los puentes, pero visitando cada uno de ellos una sola vez. (Mathematics | Euler and Hamiltonian Paths, n.d.)



Gráfica 1. Los siete puentes de Königsberg. (Curth, 2015)

El irlandés William Rowan Hamilton (1805-1865) inventó el juego icosiano conocido como el rompecabezas de Hamilton en el cual se debe encontrar un ciclo por las aristas de un dodecaedro. En la teoría de grafos, los caminos hamiltonianos establecen una ruta que visita cada vértice una única vez.

(Gráficos realizados en Microsoft Visio) (Lascano, 2019)



Gráfica 2. Ciclo hamiltoniano dodecaedro dos dimensiones. (Lascano, 2019)

El holandés Edsger Dijkstra (1930-2002) se planteó una ruta óptima para llegar desde Rotterdam a Groningen y en general desde una ciudad a otra y concluyó con un algoritmo conocido como el de camino corto que lo elaboró en veinte minutos sin embargo solamente lo publicó luego de tres años y se convirtió en uno de sus hitos por lo que es conocido. (Yan, 2014)

## 2 Objetivos

- Profundizar en mi comprensión sobre los tipos de grafos, sus aplicaciones y lo que cada una significa.
- Profundizar mi conocimiento en circuitos eulerianos y hamiltonianos mediante la revisión de los teoremas planteados para análisis de grafos.
- Manejar matrices y arreglos para almacenar las propiedades de un grafo y las rutas que se pueden realizar en el mismo.
- Aprender a desarrollar algoritmos y aplicarlos en código para que un programa pueda encontrar el ciclo (camino) a seguir.

Todos estos objetivos serán alcanzados al realizar la codificación de un algoritmo que descubra en un grafo determinado si se puede plantear un camino que visite cada lugar una sola vez y termine en el lugar inicial.

## 3 Conceptos básicos de grafos

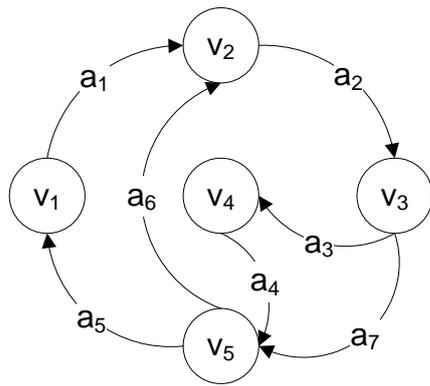
Un grafo es un conjunto de puntos (vértices o nodos) y líneas (caminos, circuitos o arcos) que unen los vértices. Estas líneas pueden ser dirigidas o no dirigidas (conocidas como simétricas). Matemáticamente se lo interpreta como  $G = (V, A)$  donde  $V$  es el conjunto de puntos  $v_i$ ,  $V \neq \emptyset$  y  $A$  es un conjunto de líneas que unen dos puntos de  $V$ ;  $A$  puede ser vacío ( $\emptyset$ ) y se lo puede interpretar ya sea como un conjunto de pares de vértices  $(v_n, v_m)$  o de aristas  $a_p$  que están relacionadas mediante la aplicación  $T$ . (Watkins & Wilson, 1990)

$$V = \{V_1, V_2, V_3, V_4, V_5, V_6, \dots, V_n\}$$

$$A = \{(V_1, V_2), (V_2, V_5), (V_3, V_6), \dots, (v_n, v_m)\} = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, \dots, a_p\}$$

$$T: V \rightarrow A, v_i \rightarrow v_j$$

Por ejemplo, definamos matemáticamente el siguiente grafo:



Gráfica 3. Ejemplo de grafo vértices y aristas. (Lascano, 2019)

$$T: (V_1) = V_1 \rightarrow V_2 = a_1$$

$$T: (V_2) = V_2 \rightarrow V_3 = a_2$$

$$T: (V_3) = \{V_3 \rightarrow V_4, V_3 \rightarrow V_5\} = \{a_3, a_7\}$$

$$T: (V_4) = V_4 \rightarrow V_5 = a_4$$

$$T: (V_5) = \{V_5 \rightarrow V_1, V_5 \rightarrow V_2\} = \{a_5, a_6\}$$

Un vértice tiene grado par cuando tiene caminos pares asociados al mismo. Un vértice es de grado impar cuando tiene caminos impares asociados al mismo. En el ejemplo anterior,  $V_1$ ,  $V_4$ ,  $V_5$  tiene grados pares;  $V_2$ ,  $V_3$  tiene grados impares.

Un grafo puede ponderarse mediante la asociación de un valor o peso a cada arista del gráfico. El peso de un camino en un grafo se obtiene con la suma de los pesos de todas las aristas atravesadas. (Watkins & Wilson, 1990)

#### 4 El camino Euleriano

Como se mencionaba en la introducción, Euler definió las reglas para que un camino o circuito pueda definirse como camino euleriano. Estas reglas plantean:

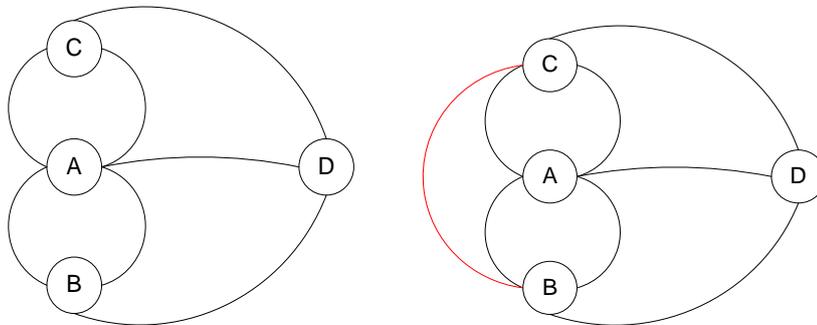
- Un **camino** euleriano es un camino que usa cada arista de un grafo solamente una vez y comienza y termina en diferentes vértices
- Un **circuito** euleriano es un circuito que usa cada arista de un grafo solamente una vez y comienza y termina en el mismo vértice

Para definir si un grafo tiene un camino euleriano se tiene los siguientes teoremas:

- Si los vértices de un grafo son todos de grado par entonces el grafo tiene un circuito euleriano.

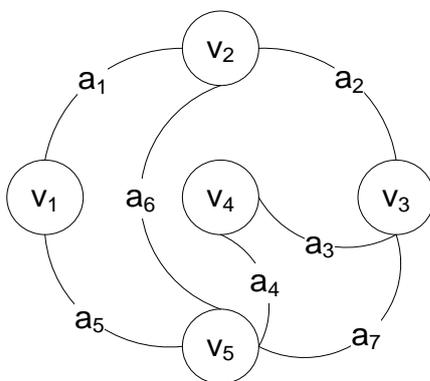
- Si un grafo tiene exactamente dos vértices de grado impar, entonces tiene un camino euleriano (pero no un circuito euleriano) que arranca y termina en los vértices de grado impar. Caso contrario no tiene un camino euleriano

Usando estos teoremas y regresando al problema planteado por Euler en su visita a la ciudad de Königsberg, se puede ver que no tiene un camino Euleriano ya que todos sus vértices tienen grado impar y se necesitaría un puente adicional para que llegue a ser euleriano (ver puente marcado por la línea roja) (Watkins & Wilson, 1990)



Gráfica 4. El grafo de los puentes de Königsberg (izquierda) y el grafo modificado para que sea euleriano (derecha). (Lascano, 2019)

En el ejemplo de la Gráfica 2 se puede constatar que se tiene un camino euleriano (pero no un circuito euleriano):  $V_2, V_1, V_5, V_2, V_3, V_4, V_5, V_3$



Gráfica 5. Grafo con camino euleriano. (Lascano, 2019).

## 5 Ciclos y caminos hamiltonianos

- Un **camino** simple en un grafo  $G$  que pasa por cada vértice una sola vez es llamado camino hamiltoniano.
- Un **circuito** simple en un grafo  $G$  que pasa por cada vértice una sola vez es llamado circuito hamiltoniano

A diferencia de los caminos eulerianos, no existe un criterio simple para determinar un camino o circuito hamiltoniano en un grafo, sin embargo, existen ciertos criterios que sirven como reglas para definir su existencia como, por ejemplo, si hay un vértice de grado uno en un grafo entonces es imposible tener un circuito hamiltoniano. Tenemos teoremas que definen condiciones para la existencia de grafos hamiltonianos: (Shortest Path Algorithms, n.d.) (Watkins & Wilson, 1990)

### 5.1 Teorema de Dirac

Si  $G$  es un grafo simple con  $n$  vértices donde  $n \geq 3$  y el grado de cada vértice es al menos  $n/2$ , entonces  $G$  es un circuito hamiltoniano. (Watkins & Wilson, 1990)

### 5.2 Teorema de Ore

Si  $G$  es un grafo simple con  $n$  vértices donde  $n \geq 3$  y  $\text{grado}(u) + \text{grado}(v) \geq n$  para cada par de vértices no adyacentes  $v$  y  $u$ , entonces  $G$  es un circuito hamiltoniano

Estos teoremas son suficientes, pero no condiciones necesarias para la existencia de un circuito hamiltoniano en un grafo. Existen grafos con circuitos hamiltonianos que no siguen estas condiciones. (Shortest Path Algorithms, n.d.)

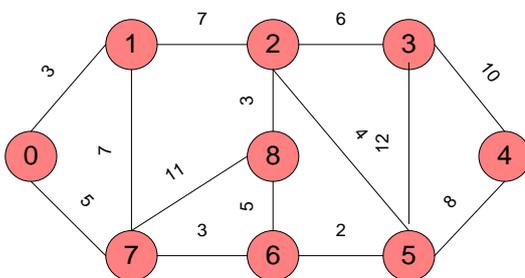
## 6 El problema camino más corto y su algoritmo

El problema del camino más corto consiste en encontrar un camino entre dos vértices en un grafo de tal forma que la suma los pesos de las aristas sea mínimo.

Este problema puede ser resuelto usando algoritmos creados por Richard Bellman, Samuel End y Lester Ford o una variación óptima definida por Dijkstra y es en la cual nos centraremos:

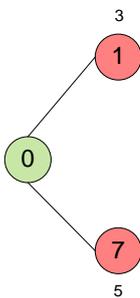
1. Creamos una lista (set) que la llamamos lista del camino corto Setcmc y mantiene un registro de los vértices en el camino corto, es decir las distancias mínimas desde el vértice inicial hasta el final. Al principio esta lista está vacía.
2. Asignamos un valor bien alto de distancia a todos los vértices en el grafo y un valor de cero para el vértice inicial de tal forma que sea considerado primero.
3. Mientras que la lista Setcms no incluya todos los vértices:
  - a. Tomamos un vértice  $u$  que no esté en Setcmc y tiene un valor de distancia mínimo
  - b. Incluimos  $u$  al Setcmc
  - c. Actualizamos los valores de distancia de todos los vértices adyacentes a  $u$  mediante la iteración por estos vértices adyacentes. Para cada vértice adyacente  $v$ , si la suma de la distancia de  $u$  (desde el inicio) y el peso de la arista  $u-v$  es menor que la distancia de  $v$ , entonces se actualiza el valor de la distancia de  $v$ . (Fan, 2012)

Ejemplo:



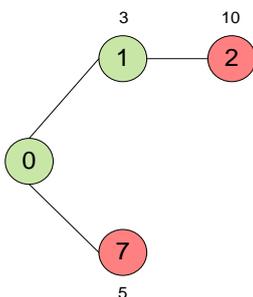
Gráfica 6. Grafo para algoritmo del camino más corto cmc. (Lascano, 2019).

La lista Setcmc está inicialmente vacía y los vértices asignados son: {0, INF, INF, INF, INF, INF, INF, INF} donde INF indica un valor bastante grande. Ahora se toma el vértice con el valor de distancia mínimo, es decir, el vértice 0 y se le incluye en la lista Setcmc y se actualiza los valores de las distancias de sus vértices adyacentes: 1 y 7. Sus valores de distancia son actualizados como 3 y 5. En el siguiente subgrafo se muestra los vértices adyacentes y el vértice ya incluido en Setcmc se muestra en color verde:



Gráfica 7. Primera iteración del algoritmo. (Lascano, 2019).

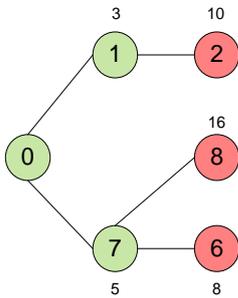
Ahora se toma el vértice con la distancia mínima y que no esté en Setcmc. El vértice 1 es tomado y adicionado a la lista Setcmc la cual pasa a ser: {0,1}. Se actualiza los valores de distancia de los vértices adyacentes a 1. La distancia del vértice 2 pasa a ser:  $3+7 = 10$ .



Gráfica 8. Segunda iteración del algoritmo. (Lascano, 2019).

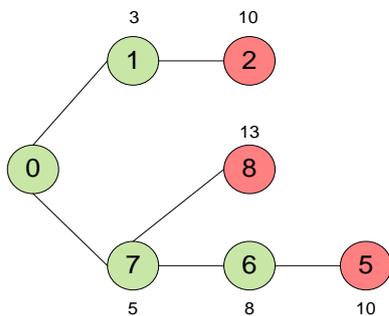
Se toma el vértice con el valor de distancia mínima que no esté en Setcmc. El vértice 7 es tomado y la lista Setcmc es ahora {0,1,7}. Se actualiza la distancia de

los vértices adyacentes a 7. Los valores de distancia de los vértices 6 y 8 pasan a ser 8 y 16.



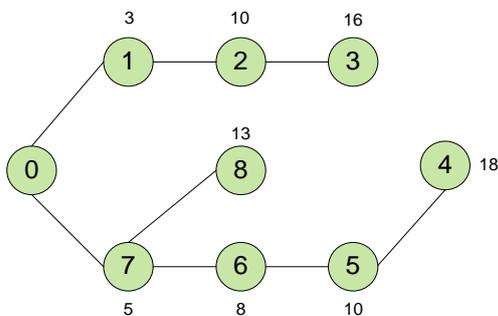
Gráfica 9. Tercera iteración del algoritmo. (Lascano, 2019).

Se toma el vértice con el valor de la distancia mínima y que no esté en Setcmc. El vértice 6 es tomado y Setcmc ahora es {0,1,7,6}. Los valores de distancia de los vértices 5 y 8 son actualizados. Se actualiza el valor del vértice 8 con la distancia 13 pues es menor a la que tenía: 16.



Gráfica 10. Cuarta iteración del algoritmo. (Lascano, 2019).

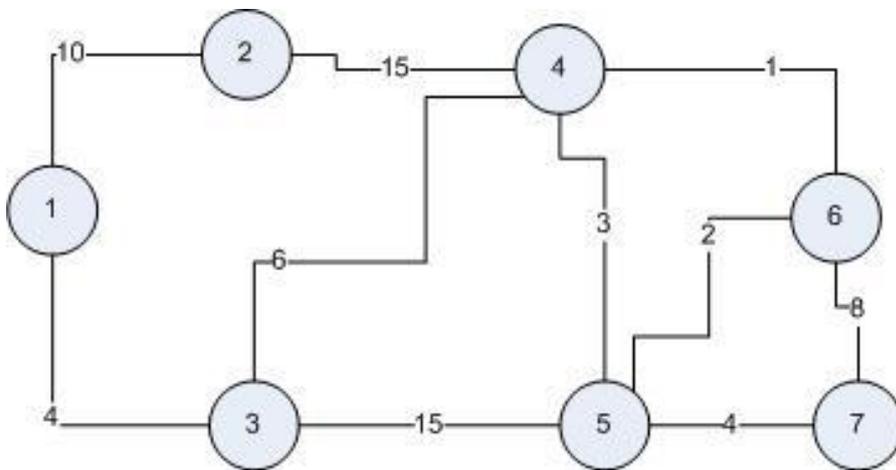
Siguiendo los mismos pasos, terminamos actualizando toda la lista Setcmc.



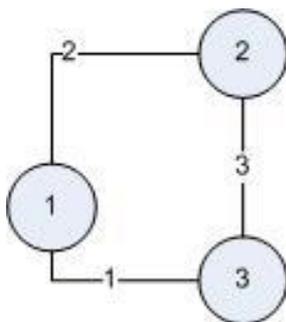
Gráfica 11. Iteraciones completadas. (Lascano, 2019).

## 7 Implementación del algoritmo

La implementación del algoritmo se la realiza mediante la escritura del pseudocódigo correspondiente a la explicación de los gráficos en la 6 El problema camino más corto y su algoritmo. Luego, mediante el uso del Ambiente de Desarrollo Integrado NetBeans se realiza la codificación del programa con su respectiva prueba y error. En el desarrollo se prueba con grafos predeterminados y en la parte final se vincula con una interfaz para que se puedan ingresar los valores del grafo.



Grafo 1. (Lascano, 2019)



Grafo 2. (Lascano, 2019)

Como paso final se realiza la ejecución del programa ingresando los datos de Grafo 1 y Grafo 2 se captura en imagen cada paso realizado.

### 7.1 Pseudocódigo

asigno un valor alto de ponderación a todos los vértices (máximo soportado);

todos los vértices son extremos y ninguno ha sido visitado;

asigno un valor de ponderación igual a cero al vértice inicial;

realizar {

    busco el extremo vértice v con el valor menor de distancia ponderada;

    el vértice v es visitado;

    si v es igual al vértice final: Se encontró el camino más corto.

    caso contrario {

        busco los vértices conectados;

        obtengo las distancias a los vértices conectados;

        si la distancia a cada vértice más el valor de distancia ponderada v es menor que el valor de distancia ponderada del vértice conectado {

            Asigno el valor de distancia ponderada del conectado al de la condición;

            El vértice v ya no es un vértice extremo;

            el vértice anterior en el recorrido al conectado es igual a v;

        }

        caso contrario si el vértice conectado ya fue visitado dicho vértice ya no es un vértice extremo para considerar.

    }

}

hasta que: Se encuentre el camino mas corto;

Se buscan los vértices anteriores conectados partiendo del final, se lo añade a una cadena de caracteres cada vértice anterior,

Se invierte dicha cadena de caracteres y se obtiene los vértices en orden del camino más corto.

Se chequea el valor de distancia ponderada del vértice final y ese es el valor del peso total del camino más corto.

## 8 Conclusiones

La matemática discreta es una rama diferenciada de la materia, puesto que trata con temas muy distintos como lo es la teoría de grafos, el tiempo de ejecución de un algoritmo, lógica booleana, combinatorias, ciertas ramas de la topología, álgebra relacional, entre otros.

La teoría de grafos en específico es una ramificación con diversas aplicaciones como lo es su vínculo con la teoría de juegos, economización, simulación de computadores cuánticos en binarios, entre otros. En la etapa de exploración se evidenció que el tema del desarrollo de algoritmos para solucionar problemas u optimizar recursos es una mina por explotar. Existen problemas tal como el de las reinas en un tablero de ajedrez que representan casos no solventados de la matemática, y este a su vez es aplicable para reducir exponencialmente el número de iteraciones en problemas de combinatorias.

Al realizar la investigación se comprendió que para elaborar programas complejos no basta para nada conocer algún lenguaje de programación, ya que, si no se tiene conocimientos de matemática discreta y algoritmos, lo único que se podrá realizar es un programa que funcione con parámetros preestablecidos, y eso no es útil.

La resolución del problema del camino más corto demuestra la capacidad de la teoría de grafos, ya que este es aplicable para la economización de recursos bajo varios ámbitos como lo es: líneas de producción, movilización, conexión rápida de redes de internet.

Respecto al código desarrollado, se evidencia que no existe una única manera de codificar un algoritmo, puesto que en páginas como lo es la fuente (Shortest Path Algorithms, n.d.) se muestran procesos con complejidad de lenguaje mayor pero que llegan al mismo punto.

Ahora, cuando se salga de paseo en familia, se logrará pasar menos tiempo en el automóvil y más tiempo disfrutando en lugares atarácicos.

## 9 Bibliografía

- Álvarez, M. F., & Parra, J. A. (2013). *TEORIA DE GRAFOS*. Chillán, Diguillín, Chile: Universidad del Bío-Bío.
- Curth, M. D. (Julio de 2015). *Los reyes de la pasarela. Modelos matemáticos en las ciencias (Divulgación científica)*. Obtenido de ResearchGate: [https://www.researchgate.net/figure/Figura-6-Mapa-del-rio-Pregel-en-Koenigsberg-por-Leonhard-Euler-que-muestra-donde-se\\_fig4\\_281096106](https://www.researchgate.net/figure/Figura-6-Mapa-del-rio-Pregel-en-Koenigsberg-por-Leonhard-Euler-que-muestra-donde-se_fig4_281096106)
- Fan, N. (24 de 11 de 2012). *Dijkstra's Algorithm*. Obtenido de Youtube: <https://www.youtube.com/watch?v=gdmfOwyQlcl&t=191s>
- Grbac, T. G., & Domazet, N. (2018). On the Applications of Dijkstra's Shortest Path Algorithm in Software Defined Networks. *Studies in Computational Intelligence*, 737:39-45.
- Lascano, L. E. (15 de 02 de 2019). Quito, Pichincha, Ecuador.  
*Mathematics | Euler and Hamiltonian Paths*. (s.f.). Obtenido de Geeks for Geeks: <https://www.geeksforgeeks.org/mathematics-euler-hamiltonian-paths/>
- Sedgewick, R., & Wayne, K. (2007). *Shortest Paths*. Obtenido de Algorithms and Data Structures Fall 2007.  
*Shortest Path Algorithms*. (s.f.). Obtenido de Hacker Earth: <https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/>
- Teoría de grafos*. (11 de 07 de 2012). Obtenido de UniPamplona: [http://www.unipamplona.edu.co/unipamplona/portallG/home\\_23/recursos/general/11072012/grafos3.pdf](http://www.unipamplona.edu.co/unipamplona/portallG/home_23/recursos/general/11072012/grafos3.pdf)
- Watkins, J. J., & Wilson, R. J. (1990). *GRAPHS-An introductory approach*. New York: John Wiley & Sons, Inc.
- Yan, M. (08 de 01 de 2014). *DIJKSTRA'S ALGORITHM*. Obtenido de Undergraduate seminar Discrete Mathematics: <http://www-math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf>

## 10 Anexo

### 10.1 Código en JAVA

El proyecto tiene de nombre CicloHamiltonianoIAMath.

Existen dos clases, la principal (InicialAppCMC) y la clase que es la plantilla de un grafo (Grafo).

#### 10.1.1 Código clase InicialAppCMC

```
package ciclohamiltonianoiamath;
import javax.swing.JOptionPane;
/**
 *
 * @author luislascano
 */
public class InicialAppCMC {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Double estGrafoPrueba [] []= new Double [6][6];

        ///EJEMPLO DE GRAFO (estGrafoPrueba)
        /*1*/ estGrafoPrueba [0] [1] = 10.;
        /*2*/ estGrafoPrueba [0] [2] = 4.;
        /*3*/ estGrafoPrueba [1] [3] = 15.;
        /*4*/ estGrafoPrueba [2] [3] = 6.;
        /*5*/ estGrafoPrueba [2] [4] = 15.;
        /*6*/ estGrafoPrueba [3] [4] = 3.;
        /*7*/ estGrafoPrueba [3] [5] = 5.;
```

```

/*8*/ estGrafoPrueba [4] [5] = 2.;
////////////////////////////////////
Double estGrafoPrueba1 [] []= new Double [7][7];

/*1*/ estGrafoPrueba1 [0] [1] = 10.;
/*2*/ estGrafoPrueba1 [0] [2] = 4.;
/*3*/ estGrafoPrueba1 [1] [3] = 15.;
/*4*/ estGrafoPrueba1 [2] [3] = 6.;
/*5*/ estGrafoPrueba1 [2] [4] = 15.;
/*6*/ estGrafoPrueba1 [3] [4] = 3.;
/*7*/ estGrafoPrueba1 [3] [5] = 5.;
/*8*/ estGrafoPrueba1 [4] [5] = 2.;
/*9*/ estGrafoPrueba1 [5] [6] = 8.;
/*10*/ estGrafoPrueba1 [4] [6] = 4.;
    // TODO code application logic here

    Grafo gf = new Grafo (estGrafoPrueba1);
String solucion = gf.obtenerCMC(0, 6);
System.out.println(solucion);

////////////////////////////////////
int continuar = JOptionPane.YES_OPTION ;
do{
    try{
        Grafo grafo;
int numeroVertices;
int c = 0;
String datosGrafo="";
int verticeInicio;

```

```

int verticeFinal;

numeroVertices = Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el numero de vertices de su Grafo"));
Double estructuraGrafo[][]=new Double[numeroVertices][numeroVertices];
do{
    int numeroConexionesVerticeActual =Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el numero de conexiones del vertice:
"+(c+1)));
    for(int i =0; i<numeroConexionesVerticeActual;i++){
        int verticeConexion = Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el numeral del vertice con el que el vertice "+(c+1)+"
tiene su conexion "+(i+1))-1;
        Double valorVerticeConexion = Double.parseDouble(JOptionPane.showInputDialog(null, "Ingrese el valor del peso de la conexion o arco entre
los vertices: "+(c+1)+" y "+(verticeConexion+1)));
        estructuraGrafo[c][verticeConexion]= valorVerticeConexion ;
    }
    c++;
}
while (c<numeroVertices);
int repetirCalc ;
do{

    for(int y =0; y<estructuraGrafo[0].length; y++){
        for(int x =0; x<estructuraGrafo[0].length; x++){
            if (estructuraGrafo[y][x] != null){
                datosGrafo += "\n conexion entre vertice "+y+" y "+x+" = "+estructuraGrafo[y][x];
            }
        }
    }
    JOptionPane.showMessageDialog(null, "Los datos de su grafo son: \n "+datosGrafo);
    datosGrafo="";
    verticeInicio =Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el vertice inicial))-1;

```

```

verticeFinal = Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el vertice final))-1;

grafo = new Grafo (estructuraGrafo);
solucion = grafo.obtenerCMC(verticeInicio, verticeFinal);

JOptionPane.showMessageDialog(null, "El camnio mas corto es: \n"+solucion+"\n El peso total del camino es: "+grafo.totalDistanciaRecorrer);
repetirCalc = JOptionPane.showConfirmDialog(null, "Desea realizar nuevo calculo en este grafo?");
}
while(repetirCalc == JOptionPane.YES_OPTION);
continuar = JOptionPane.showConfirmDialog(null, "Desea ingresar un nuevo grafo?");
}
    catch(Exception e){

        JOptionPane.showMessageDialog(null, "Se ha producido un error por informacion erronea");
        System.exit(1);
    }
}
while(continuar == JOptionPane.YES_OPTION);
}
}

```

### 10.1.2 Código clase Grafo

```
package ciclohamiltonianoiamath;
/**
 *
 * @author luisl
 */
public class Grafo {
    int numeroVertices;
    int numeroArcos;
    boolean visitado[];
    boolean verticesExtremo[];
    boolean primeraVez = true;
    Double distanciaPonderada[]; //Importante
    String preSolucionGrafo;
    int numeroVerticeInicio;
    int numeroVerticeFinal;
    Double distanciaVerticeMasBajo;
    int verticeMenorPonderizado;
    Double totalDistanciaReconner;
    Integer verticeAnterior[]; //Importante
    boolean verticesConectados[];

    boolean seguirBuscando = true;
    int verticeMasBajo;
    int central;
    Double estructuraGrafo[][];
    String solucionGrafo = "";
    public Grafo(Double estructuraGrafo[][]) {
        this.estructuraGrafo = estructuraGrafo;
    }
}
```

```

numeroVertices = estructuraGrafo[0].length;
System.out.println(" Numero de vertices " + numeroVertices);          ////CONSTRUCTOR
visitado = new boolean[numeroVertices];
distanciaPonderada = new Double[numeroVertices];
}
////////////////////////////////////
///Funcion para encontrar el camino mas corto//
public String obtenerCMC(int verticeInicial, int verticeFinal) {
    verticesExtremo = new boolean[numeroVertices];
    boolean continuarBusqueda = true;
    grafoSetUpValores(verticeInicial);
    verticeAnterior = new Integer[numeroVertices*(numeroVertices-1)];
    int anterior = 0;
    int anterior2 = 0;

    int count = 0;
    int contador;
    int pos = 0;
    int i;
    do {
        pos = 0;
        verticeMenorPonderizado = obtenerVerticeMenorPonderizado();
        visitado[verticeMenorPonderizado] = true;
        System.out.println("Vertice menor ponderizado: " + verticeMenorPonderizado);
        if (verticeMenorPonderizado == verticeFinal) {
            continuarBusqueda = false;
            System.out.println("Distancia minima hasta al final = " + distanciaPonderada[verticeFinal]);
            totalDistanciaRecorrer = distanciaPonderada[verticeFinal];
        } else {

```

```

getVerticesConectados(verticeMenorPonderizado, numeroVertices, estructuraGrafo);
do {
    if (verticesConectados[pos]) {
        if (distanciaPonderada[pos] >= (distanciaPonderada[verticeMenorPonderizado] + distanciaVertice(verticeMenorPonderizado,
pos))) {

            verticesExtremo[verticeMenorPonderizado] = false;
            distanciaPonderada[pos] = distanciaVertice(verticeMenorPonderizado, pos) + distanciaPonderada[verticeMenorPonderizado];
            verticeAnterior[pos] = verticeMenorPonderizado;
        } else {
            System.out.println("verticeMenorPonderizado: " + verticeMenorPonderizado);
            if (visitado[verticeMenorPonderizado] = true) {
                verticesExtremo[verticeMenorPonderizado] = false;
            }
        }
    }
    pos++;
    count++;
} while (pos < numeroVertices);
}
} while (continuarBusqueda);
int p =verticeFinal;
preSolucionGrafo = (verticeFinal+1) +"";
while( p!= 0){
    preSolucionGrafo += (1+verticeAnterior[p])+"";
    p =verticeAnterior[p];
}
System.out.println("\n El camino a seguir es: ");
for(int l= preSolucionGrafo.length(); l>0;l--){

    if ((l-1) ==0){

```

```

        solucionGrafo += preSolucionGrafo.charAt(l-1);
    }
    else{
        solucionGrafo += preSolucionGrafo.charAt(l-1)+"-";
    }
}
return solucionGrafo;
}
///fin funcion CMC
////////////////////////////////////
//Funcion para encontrar los vertices conectados al vertice central
private boolean[] getVerticesConectados(int verticeCentral, int cantidadVertices, Double[][] estGrafo) {

    boolean verticeVinculado = true;

    verticesConectados = new boolean[cantidadVertices];
    for (int i = 0; i < cantidadVertices; i++) {
        if (i == verticeCentral) {
            System.out.println("\n Comparacion realizada con vertice central \n");
            verticesConectados[i] = false;
        } else {
            verticesConectados[i] = estGrafo[verticeCentral][i] != null;
            if (verticesConectados[i] == false) {
                verticesConectados[i] = estGrafo[i][verticeCentral] != null;
            }
            System.out.println("Comparacion realizada con vertice: " + i);
            System.out.println("Conectado: " + verticesConectados[i]);
        }
    }
}
}

```

```

    verticeVinculado = true;
    System.out.println("");

    return verticesConectados;
}
////Funcion para inicializar las distancias ponderadas a valor muy alto////////////////////////////////////
private void grafoSetUpValores(int verticeInicial) {

    distanciaPonderada[verticeInicial] = 0.0;
    visitado[verticeInicial] = true;
    //verticesExtremo[verticeInicial]= false;

    for (int i = 0; i < estructuraGrafo[0].length; i++) {
        if (i == verticeInicial) {
        } else {
            distanciaPonderada[i] = 99999.0;
            visitado[i] = false;
            verticesExtremo[i] = true;
        }
    }
}

////////////////////////////////////
//// FUNCION PARA OBTENER DISTANCIA DE VERTICE X A Y / ////
private Double distanciaVertice(int verticeCentral, int verticeSecundario) {
    Double distanciaVertice = 0.0;
    if (estructuraGrafo[verticeCentral][verticeSecundario] != null) {
        distanciaVertice = estructuraGrafo[verticeCentral][verticeSecundario];
    } else if (estructuraGrafo[verticeSecundario][verticeCentral] != null) {

```

```

    distanciaVertice = estructuraGrafo[verticeSecundario][verticeCentral];
} else {
    System.out.println("No existe conexion entre :" + verticeCentral + " y " + verticeSecundario);
}
return distanciaVertice;
}
////////////////////////////////////
//////////      FUNCION PARA OBTENER EL EXTREMO MENOR PONDERIZADO      //
private int obtenerVerticeMenorPonderizado() {
    double menorDistancia = 999;
    int vertice = 0;
    int count = 0;
    do {
        if (((distanciaPonderada[count] <= menorDistancia) && verticesExtremo[count])) {
            menorDistancia = distanciaPonderada[count];
            vertice = count;
        }
        count++;
    } while (count < numeroVertices);
    return vertice;
}
}

```

## 10.2 Ejecución

No es necesario volver a enlistar una de las conexiones de un vértice (ya enlistada).

Capturas de pantalla: (Lascano, 2019)

Se ingresa la información del Grafo 1.

Input ×

**?** Ingrese el numero de vertices de su Grafo

Input ×

**?** Ingrese el numero de conexiones del vertice: 1

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 1 tiene su conexion 1

Input ×

**?** Ingrese el valor del peso de la conexion o arco entre los vertices: 1 y 2

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 1 tiene su conexion 2

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vertices: 1 y 3

OK Cancel

Input ×

**?** Ingrese el numero de conexiones del vertice: 2

OK Cancel

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 2 tiene su conexión 1

OK Cancel

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vertices: 2 y 4

OK Cancel

Input ×

**?** Ingrese el numero de conexiones del vertice: 3

OK Cancel

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 3 tiene su conexión 1

OK Cancel

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vértices: 3 y 4

Input ×

**?** Ingrese el numeral del vértice con el que el vértice 3 tiene su conexión 2

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vértices: 3 y 5

Input ×

**?** Ingrese el número de conexiones del vértice: 4

Input ×

**?** Ingrese el numeral del vértice con el que el vértice 4 tiene su conexión 1

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vértices: 4 y 5

Input ×

**?** Ingrese el numeral del vértice con el que el vértice 4 tiene su conexión 2

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vertices: 4 y 6

Input ×

**?** Ingrese el numero de conexiones del vertice: 5

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 5 tiene su conexión 1

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vertices: 5 y 6

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 5 tiene su conexión 2

Input ×

**?** Ingrese el valor del peso de la conexión o arco entre los vertices: 5 y 7

Input ×

**?** Ingrese el numero de conexiones del vertice: 6

Input ×

 Ingrese el numeral del vertice con el que el vertice 6 tiene su conexion 1

Input ×

 Ingrese el valor del peso de la conexion o arco entre los vertices: 6 y 7

Input ×

 Ingrese el numero de conexiones del vertice: 7

Message ×

 Los datos de su grafo son:

conexion entre vertice 0 y 1 = 10.0  
conexion entre vertice 0 y 2 = 4.0  
conexion entre vertice 1 y 3 = 15.0  
conexion entre vertice 2 y 3 = 6.0  
conexion entre vertice 2 y 4 = 15.0  
conexion entre vertice 3 y 4 = 3.0  
conexion entre vertice 3 y 5 = 1.0  
conexion entre vertice 4 y 5 = 2.0  
conexion entre vertice 4 y 6 = 4.0  
conexion entre vertice 5 y 6 = 8.0

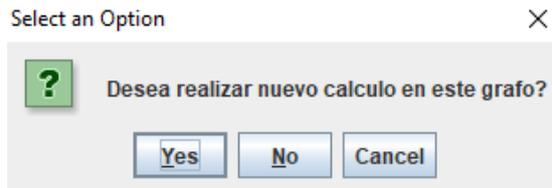
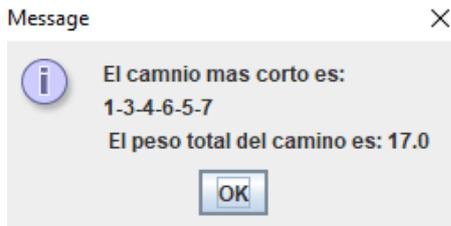
Input ×

 Ingrese el vertice inicial

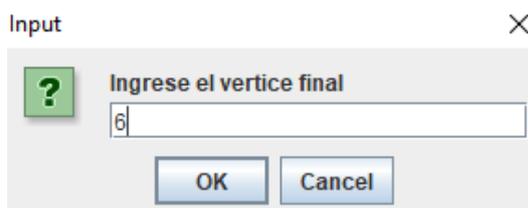
Input ×

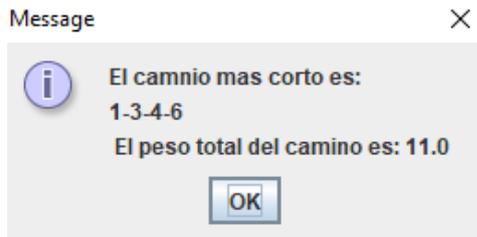
 Ingrese el vertice final

### 10.2.1 Primer resultado Grafo 1

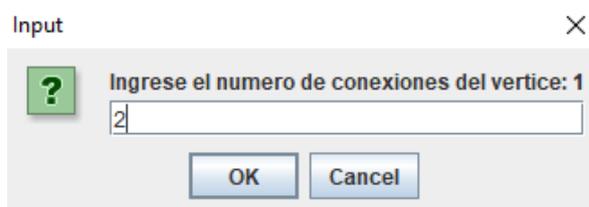
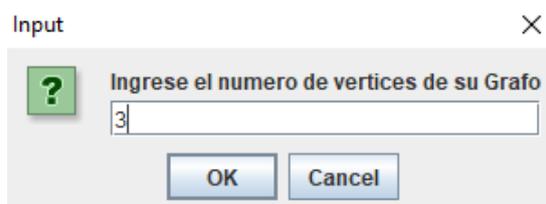
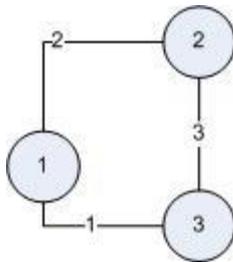
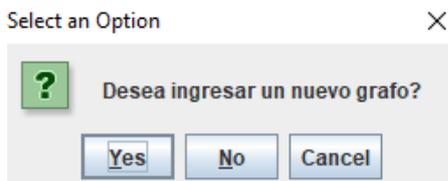
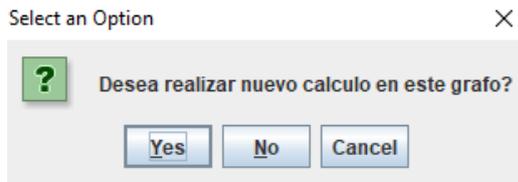


Se selecciona para volver a ingresar un cálculo.





Se seleccionan las opciones para ingresar un nuevo Grafo.



Input ×

**?** Ingrese el numeral del vertice con el que el vertice 1 tiene su conexion 1

Input ×

**?** Ingrese el valor del peso de la conexion o arco entre los vertices: 1 y 2

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 1 tiene su conexion 2

Input ×

**?** Ingrese el valor del peso de la conexion o arco entre los vertices: 1 y 3

Input ×

**?** Ingrese el numero de conexiones del vertice: 2

Input ×

**?** Ingrese el numeral del vertice con el que el vertice 2 tiene su conexion 1

Input ×

**?** Ingrese el valor del peso de la conexion o arco entre los vertices: 2 y 3

Input ×

 Ingrese el numero de conexiones del vertice: 3

Message ×

 Los datos de su grafo son:

conexion entre vertice 0 y 1 = 2.0  
conexion entre vertice 0 y 2 = 1.0  
conexion entre vertice 1 y 2 = 3.0

Input ×

 Ingrese el vertice inicial

Input ×

 Ingrese el vertice final

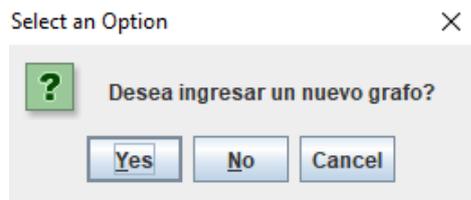
### 10.2.2 Primer resultado Grafo 2

Message ×

 El camino mas corto es:  
1-2  
El peso total del camino es: 2.0

Select an Option ×

 Desea realizar nuevo calculo en este grafo?



Fin ejecución