Sección de Matemáticas
Universidad de La Laguna

Omaira González Pérez

Problemas de Conectividad en Grafos y Redes

Connectivity Problems in Graphs and Networks

Trabajo Fin de Grado
Grado en Matemáticas
La Laguna, Junio de 2024

Dirigido por
Carlos González Martín

Carlos González Martín
Departamento de Matemáticas,
Estadística e Investigación
Operativa
Universidad de La Laguna
38200 La Laguna, Tenerife

Agradecimientos

Quiero agradecer a mi tutor Carlos por guiarme y ayudarme durante todo el proceso.

A mi familia, en especial a mis padres que me han apoyado en todo momento. Gracias por ser mi mayor respaldo y por estar siempre a mi lado.

A mi abuela, que aunque ya no esté, sé que se sentiría muy orgullosa de mi.

Omaira González Pérez
La Laguna, 13 de junio de 2024

Resumen · Abstract

Resumen

En este trabajo se estudian algunos problemas básicos de conectividad en grafos y redes. Para ello se introducen algunos conceptos imprescindibles para posteriormente comprender los diferentes problemas que se plantean. Se estudiarán algoritmos relacionados con la búsqueda del camino mínimo, algoritmos para encontrar el árbol generador mínimo y problemas clásicos de circuitos en grafos y redes. Todos estos problemas vienen acompañados de ejemplos, los cuales se resuelven utilizando determinados códigos. Finalmente, se estudian otros algoritmos de interés.

Palabras clave: *Teoría de Grafos y Redes – Algoritmo de Dijkstra – Algoritmo de Bellman-Ford – Algoritmo de Kruskal – Algoritmo de Prim – Problema del viajante – Problema del cartero chino ....*

Abstract

In this work, some basic connectivity problems in graphs and networks are studied. For this purpose, some essential concepts are introduced to subsequently understand the different problems that are presented. Algorithms related to the search for the shortest path, algorithms to find the minimum spanning tree, and classical circuit problems in graphs and networks will be studied. All these problems are accompanied by examples, which are solved using specific codes. Finally, other algorithms of interest are studied.

Keywords: *Graph and Network Theory – Dijkstra Algorithm – Bellman-Ford Algorithm – Kruskal Algorithm – Prim Algorithm – Traveling Salesman Problem – Chinese Postman Problem ....*

Contenido

Introducción

En el mundo actual es esencial la conexión entre sistemas diversos y entre las distintas componentes de los mismos. Frecuentemente, los problemas relacionados, de gran importancia práctica, se pueden modelizar sobre grafos y redes, estructuras matemáticas en las que determinadas "posiciones" aparecen convenientemente conectadas.

En este trabajo se estudian algunos problemas básicos de conectividad en grafos y redes que son de gran relevancia en logística, transporte, comunicaciones..., prestando atención a los aspectos algorítmicos y a las aplicaciones.

El objetivo del capítulo 1 es definir los conceptos básicos de la teoría de grafos para poder comprender los problemas que se plantean en los siguientes capítulos. En primer lugar se define lo que es un grafo y una red y se detallan distintos conceptos de interés para el desarrollo posterior de la memoria, ilustrando con ejemplos y gráficas. Se completa el capítulo con diferentes representaciones de grafos.

En el capítulo 2 se introducen dos algoritmos para encontrar el camino mínimo entre dos nodos de un grafo. El primero de ellos es el algoritmo de Dijkstra [6] el cual fue publicado por Edsger Dijkstra en 1959 y resuelve el problema para un grafo sin pesos negativos. Este algoritmo se implementa sobre un ejemplo usando dos códigos realizados en R y en Python mostrando el resultado obtenido al compilarlos. El siguiente método que se aborda en este capítulo es el de Belman-Ford, [11] publicado por Richard Bellman y Lester Ford Jr en 1956 y 1958, el cual si es capaz de encontrar el camino mínimo en un grafo con pesos negativos. Se aplicará a un ejemplo de grafo con pesos negativos usando un código en Python.

En el capítulo 3 se estudian dos algoritmos desarrollados para encontrar árboles generadores óptimos. Uno de estos algoritmos es el de Kruskal, publicado por Joseph Kruskal en 1956. Este algoritmo se aplicará a un ejemplo y se muestra cual es el resultado. Otro de los algoritmos que se estudia en este capítulo es el algoritmo de Prim, el cual fue introducido por Robert Prim en 1957. También se implementa el algoritmo a un ejemplo y se expone el resultado.

En el capítulo 4 se estudian dos problemas clásicos de circuitos en grafos y redes. El primer problema es el Problema del Viajante o también conocido como TSP, es un problema donde un viajante quiere visitar n ciudades partiendo de una de ellas y llegando a la misma ciudad de partida, de forma que se minimice la distancia total recorrida. El recorrido que representa este problema es un circuito hamiltoniano. El otro problema que se estudia es el Problema del Cartero Chino. Se refiere a un cartero que debe entregar el correo para un sistema (conectado) de calles: nuestro cartero desea minimizar la distancia total que tiene que recorrer configurando su ruta de manera adecuada. En este caso, el recorrido que representa es un ciclo euleriano. Para ver mejor como funcionan estos problemas se muestra un ejemplo para el problema del viajante y tres ejemplos para el problema del cartero chino.

En el capítulo 5 se estudian otros algoritmos, cada uno de ellos con un ejemplo.

1

Teoría de Grafos

Hablar de grafos es referirse a un concepto matemático básico en el que una colección de puntos se interrelacionan, total o parcialmente, mediante un conjunto de conexiones. Aunque los antecedentes se remontan a momentos históricos de la antigüedad, el origen efectivo de la teoría de grafos se puede ubicar en el siglo XVIII con el problema de los puentes de Königsberg, el cual consistía en comenzar desde un punto en la ciudad y atravesar cada uno de los siete puentes del río Pregel exactamente una vez, regresando al punto de partida.

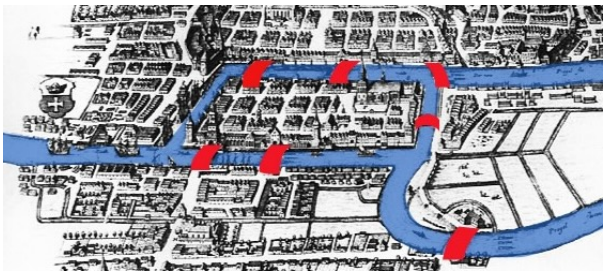


Figura 1.1. Los puentes de Königsberg.[3]

Euler, en 1739, en una publicación aporta una solución generalizada del problema, que puede llevarse a cabo en cualquier lugar en el que ciertos accesos estén restringidos a ciertas conexiones, como el de los puentes de Königsberg.

Investigaciones relevantes como la de Gustav Kirchhoff (1845), con los circuitos de cálculo de voltaje, y Francis Guthrie (1852), con la presentación de la hipótesis de los cuatro colores, le han dado estructura a la teoría de grafos como la entendemos actualmente.[4]

Desde sus orígenes, la teoría de grafos se utilizó para la resolución de juegos matemáticos, para el estudio de circuitos eléctricos y en diversas aplicaciones en una multitud de campos tan diferentes como la economía, física teórica, psi-

cología, física nuclear, lingüística, sociología, zoología, tecnología, antropología, química, biología, etc.

La Teoría de Grafos tiene un poderoso apoyo en los problemas de transporte. Desde un punto de vista elemental, para que sea posible el transporte o la comunicación, son necesarios puntos concretos de emisión o recepción y rutas de comunicación. Estos dos elementos, puntos y rutas, se representan respectivamente por vértices y conexiones. La figura así obtenida es una red de transporte. Además las conexiones pueden estar orientadas según si las rutas de desplazamiento necesitan definirse en un sentido obligatorio o puedan recorrerse en ambos sentidos. [5]

En este capítulo se introducen algunos conceptos básicos sobre la teoría de grafos que serán de gran utilidad para desarrollar contenidos de los siguientes capítulos.

1.1. Grafos y redes

Definición 1: Un grafo es una configuración física o abstracta donde hay lugares o puntos unidos por enlaces.

Estos puntos se denominan nudos, nodos o vértices y a las conexiones se les llaman aristas (si no tienen designada una dirección) o arcos (si tienen asignada una dirección).

Definición 2: Un grafo **dirigido** es un par G=(V,A) formado por un conjunto finito de vértices $V \neq \emptyset$ y un conjunto de arcos $A = \{a = (i, j) : i, j \in V, i \neq j\}$ donde i es el vértice de origen y j el vértice de destino.

Definición 3: Un grafo **no dirigido** es un par G=(V,E) formado por un conjunto finito de vértices $V \neq \emptyset$ y un conjunto de aristas $E = \{e = (i, j) : i, j \in V, i \neq j\}$.

Una arista o arco que conecta un vértice consigo mismo se llama bucle, es decir, $e = (i, i)$ si es una arista y $a = (i, i)$ si es un arco.

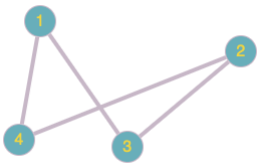


Figura 1.2. Grafo no dirigido.

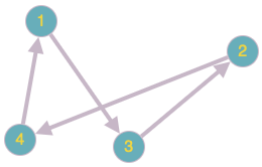


Figura 1.3. Grafo dirigido

Existen grafos mixtos que contienen tanto aristas como arcos.

Figura 1.4. Grafo mixto.

Definición 4: Una red es un grafo que tiene asociadas a los vértices y/o a las aristas o arcos determinadas magnitudes.

Estas magnitudes pueden ser, por ejemplo, distancias si estamos en una red de carreteras o capacidades si estamos hablando del tráfico. En los vértices las magnitudes pueden ser disponibilidades (ofertas, demandas...)

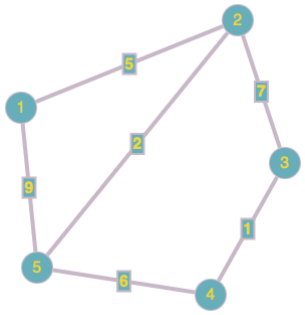


Figura 1.5. Red.

1.2. Conceptos básicos

A partir de este momento se asociarán los conceptos a un grafo dirigido, teniendo en cuenta que cualquier arista se puede reconvertir en dos arcos.

Definición 5: Una cadena es una secuencia de vértices en la que cada vértice está conectado al siguiente mediante un arco. Un camino es una secuencia ordenada de vértices y arcos distintos, donde se conecta un vértice origen i con un vértice destino j.

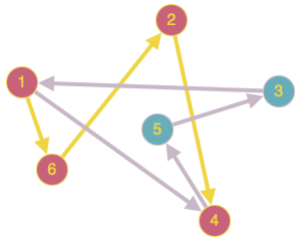


Figura 1.6. Ejemplo de camino entre el vértice 1 y el vértice 4.

Definición 6: Un grafo o una red se llama conexo si existe, al menos, una cadena que conecta cualquier par de vértices.

Definición 7: Un ciclo es un camino donde el vértice de origen y el de destino coinciden.

Definición 8: Un circuito es un ciclo en un grafo dirigido en el cual todos los arcos están orientados de tal manera que se puede recorrer desde el vértice inicial hasta el vértice final siguiendo la dirección de los arcos.

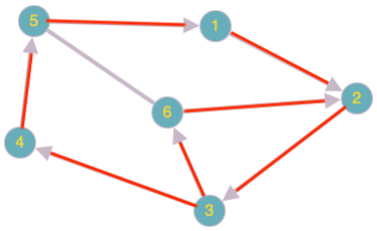


Figura 1.7. Circuito

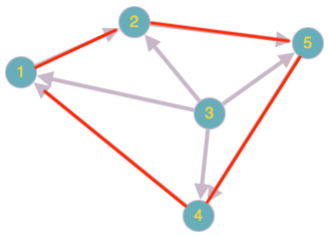


Figura 1.8. Ciclo

Definición 9: Sea $G = (V, A)$. $G' = (V', A')$ se dice subgrafo o subred de G si:

1. $V' \subseteq V$.
2. $A' \subseteq A$.
3. $G' = (V', A')$ es un grafo.

Definición 10: Sea $G = (V, A)$. $G' = (V', A')$ se dice grafo parcial o red parcial de G si:

1. $V' = V$.
2. $A' \subseteq A$.

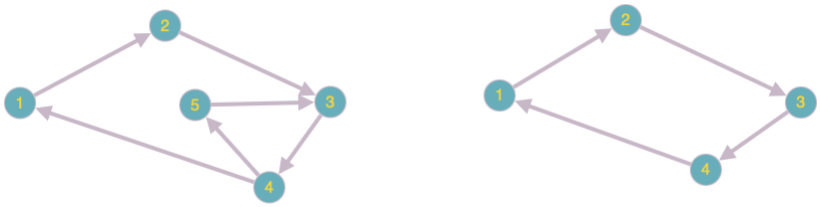


Figura 1.9. Ejemplo de un grafo y un subgrafo.

Los caminos, ciclos, circuitos… son grafos o redes parciales. Cuando un ciclo o un circuito comprende todos los vértices del grafo o red parcial, se denomina ciclo hamiltoniano. Si en un ciclo o en un circuito están todas las aristas o los arcos correspondientes, se llama ciclo euleriano.

Definición 11: Sea $G = (V, A)$ un grafo, se dice que es un multigrafo si dos vértices pueden estar conectados por más de una arista o arco.

Circuito hamiltoniano

Un **circuito** hamiltoniano es un circuito que debe pasar por cada vértice de un grafo una vez y solo una vez.

Un **camino** hamiltoniano es un camino que debe pasar por cada vértice de un grafo una vez y solo una vez.

Este es un ejemplo de un grafo con un circuito hamiltoniano.

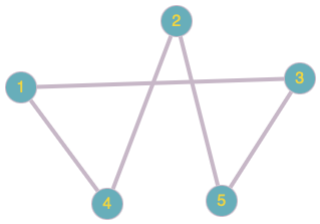


Figura 1.10. Ejemplo de grafo con un circuito hamiltoniano.

Existen grafos donde pueden haber caminos hamiltonianos pero no ciclos, por ejemplo:

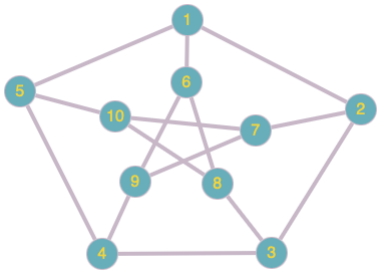


Figura 1.11. Ejemplo de grafo con un camino hamiltoniano.

Ciclo euleriano

Un **ciclo** euleriano es un ciclo en el grafo que utiliza cada arista exactamente una vez.

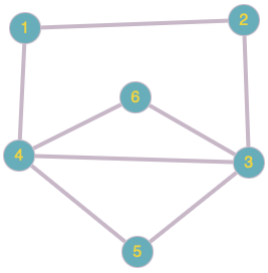


Figura 1.12. Ejemplo de grafo con un ciclo euleriano.

Definición 12: Sea $G = (V, A)$. Una componente conexa es un conjunto $V' \subseteq V$ donde cada par de vértices de V' están conectados por al menos un camino.

Definición 13: Un arco puente es un arco que conecta dos componentes conexas de un grafo.

Definición 14: Un grafo bipartito es un grafo donde sus vértices se pueden dividir en dos conjuntos diferentes y cualquier arco conecta estos dos conjuntos.

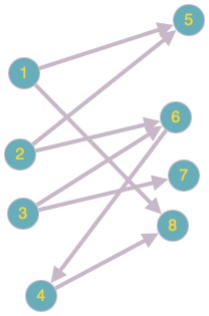


Figura 1.13. Grafo bipartito.

Definición 15: Un árbol es un grafo en el que cualquier par de vértices están conectados por un solo camino.

Existen varias definiciones equivalentes de un árbol, estas son algunas de ellas:

1. Es un grafo sin ciclos y conexo.
2. Es un grafo donde cualquier par de vértices están conectados por una única cadena.
3. Un grafo conexo donde cualquier arco es un arco puente. Si se elimina este arco puente se tienen dos componentes conexas con estructura de árbol.

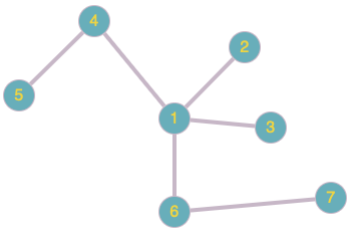


Figura 1.14. Árbol.

Definición 16: Un árbol generador mínimo es un árbol que incluye todos los vértices del grafo original y un subconjunto de arcos tal que la suma de las magnitudes asociadas a los arcos es mínima.

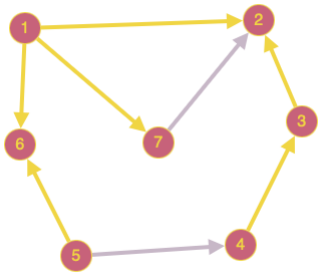


Figura 1.15. Árbol generador mínimo.

Definición 17: Dado un grafo no dirigido el grado de incidencia de un vértice i es el número de aristas que conectan con él. Se denota por $g(i)$.

En un grafo dirigido el grado de incidencia de entrada de un vértice viene dado por el número de arcos que llegan a él. El grado de incidencia de salida de un vértice es el número de arcos que salen de el.

Se llama hoja al vértice v del árbol tal que $g(v) = 1$. Cualquier árbol tiene al menos dos hojas.

1.3. Representaciones de los grafos y redes

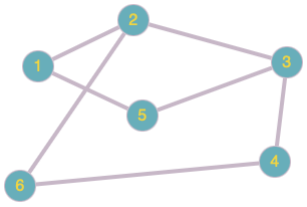
Dependiendo del entorno donde se trabaje, el uso de grafos requiere de diferentes representaciones. En cualquier caso se busca la claridad y el fácil manejo.

Matriz de adyacencia

Sea $V = \{0, 1, .., n\}$ los vértices del grafo $G = (V, E)$. La matriz de adyacencia asociada a G es una matriz A_{nxn} donde las entradas $a_{ij} = 1$ si dos vértices están directamente conectados a través de una arista o arco y $a_{ij} = 0$ en otro caso.

Como esta matriz no almacena el sentido de las aristas, es más útil para representar grafos no dirigidos.

Por ejemplo, la matriz de adyacencia de este grafo



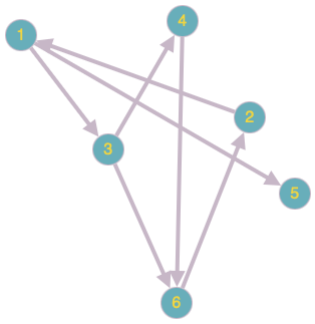
es:

$$A = \begin{bmatrix} 0\,1\,0\,0\,1\,0 \\ 1\,0\,1\,0\,0\,1 \\ 0\,1\,0\,1\,1\,0 \\ 0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0 \end{bmatrix} \tag{1.1}$$

Matriz de incidencia

Sea $V = \{0, 1, .., n\}$ los vértices del grafo $G = (V, A)$. La matriz de incidencia asociada al grafo G es una matriz B_{mxn} donde m es el número de arcos y n el números de vértices. Las entradas $b_{is} = 1$ si el vértice i esta conectado al arco s, $b_{is} = -1$ si el vértice i es el otro extremo del arco s y $b_{is} = 0$ en otro caso.

Por ejemplo, sea el grafo



la matriz de incidencia es:

$$B = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 \end{bmatrix} \tag{1.2}$$

Lista de adyacencia

Una lista de adyacencia de un grafo es una representación de todas las aristas o arcos mediante una lista.

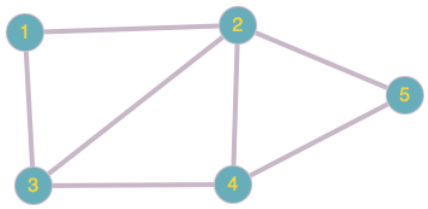


Figura 1.16. Grafo

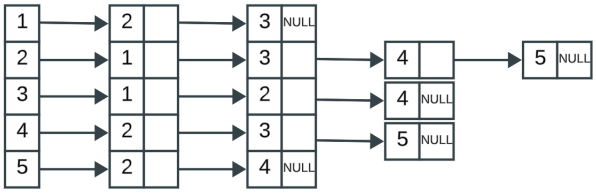


Figura 1.17. Grafo representado a través de la lista de adyacencia

2

Caminos

El estudio de caminos es de gran importancia en grafos y redes ya que contiene, necesariamente, características de eficiencia en la conectividad. En las comunicaciones, en los transportes, en la distribución de bienes de distinta índole..., utilizar rutas con atributos de optimalidad es una exigencia básica.

Entre la gama de problemas que, en este contexto, se pueden plantear, están los de caminos mínimos. Sobre un grafo con magnitudes (tiempos, distancias, costos...) en las conexiones (aristas o arcos), se pretende determinar un camino óptimo entre determinados pares de vértices.

Los algoritmos más populares en cuanto al estudio de la ruta más corta son el de Dijkstra y el de Bellman-Ford. Su función es la misma, encontrar el camino mínimo entre dos vértices, pero tienen dos grandes diferencias: el algoritmo de Dijkstra no permite pesos negativos en los arcos mientras que el de Bellman-Ford si, y la otra diferencia es la complejidad de cada uno, pues el algoritmo de Dijkstra es más eficiente en cuanto a tiempo de ejecución. La elección del algoritmo más apropiado depende de las características específicas del problema y del grafo en cuestión.

Cabe destacar que existen muchos otros algoritmos que calculan la ruta mínima, como por ejemplo el algoritmo de Floyd-Warshall, el algoritmo de Johnson y el algoritmo A*.

En este capítulo se estudiarán los algoritmos de Dijkstra y Bellman-Ford. Para ello, se analizará el pseudocódigo asociado a cada algoritmo y su código en Python, mostrando así un ejemplo para cada uno.[6]

2.1. Algoritmo de Dijkstra

2.1.1. Introdución

El algoritmo de Dijkstra fue creado por Edsger Dijkstra, quien lo publicó por primera vez en 1959.[7]

El algoritmo calcula el camino más corto entre dos vértices de un grafo cuyas aristas tienen asignados pesos. Sin embargo, si las aristas tienen pesos negativos, el algoritmo puede no funcionar correctamente.

Este algoritmo es usado en muchos aspectos de nuestra vida cotidiana, por ejemplo, en los sistemas de navegación GPS para encontrar el camino más corto entre el origen y el destino que se desee, lo que permite encontrar rutas óptimas para conducir, en transporte público o a pie.

2.1.2. Algoritmo

A continuación se presenta el pseudocódigo del algoritmo para calcular la distancia mínima desde el vértice s.[1]

En este pseudocódigo $w(uv)$ va a ser la función que devuelva el peso entre el vértice u y el vértice v mientras que $d(u)$ proporciona la distancia entre el vértice de origen s y el vértice u. También se define A_u como la lista de adyacencia del vértice u.

DIJKSTRA $(G, w, s; d)$

$d(s) = 0, V = G.vertices$

for $v \in V \backslash \{s\}$ **do** $d(v) = \infty$

while $V \neq \emptyset$ **do**

find some $u \in V$ such that $d\{u\}$ is minimal

$V = V \backslash \{u\}$

for $v \in V \cap A_u$ **do** $d(v) = min(d(v), d(u) + w(uv))$ **end**

end

2.1.3. Implementación del algoritmo

A continuación se estudiará un ejemplo mediante el código realizado tanto en R como en Python.

Código en R [8]

El código utilizado calcula las rutas y distancias más cortas desde el vértice 1 a todos los demás vértices en el grafo. Se utiliza la función shortest_paths(), especificando el grafo G, el vértice de origen (1), los vértices de destino ($to = V(G)$ para incluir todos los vértices), y el parámetro de salida para obtener tanto los caminos como las distancias.

El código asigna la variable "S" para almacenar el camino más corto desde el vértice 1 hasta el vértice 10 en el grafo G. A continuación, se asigna la variable "S2" para almacenar la secuencia de aristas del camino más corto desde el vértice 1 hasta el vértice 10.

El grafo G se representa mediante su matriz de adyacencia.

$$M = \begin{bmatrix} 0 & 3 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 17 & 16 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17 & 0 & 8 & 0 & 0 & 0 & 0 & 18 & 0 \\ 0 & 16 & 8 & 0 & 11 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 11 & 0 & 1 & 6 & 5 & 10 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 7 & 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 15 & 0 & 12 & 13 \\ 0 & 0 & 18 & 4 & 10 & 0 & 0 & 12 & 0 & 9 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 13 & 9 & 0 \end{bmatrix}$$

Al compilar el código se genera una imagen del grafo sin el camino mínimo destacado y posteriormente aparecerá el mismo grafo con el camino mínimo en naranja.

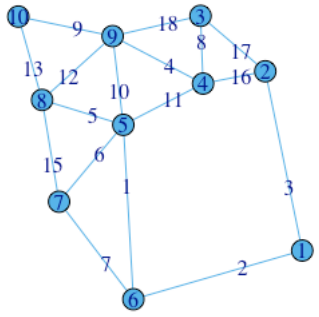


Figura 2.1. Grafo.

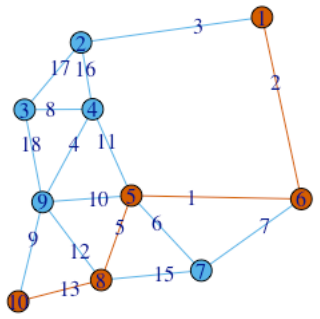


Figura 2.2. Grafo y camino mínimo desde 1 hasta 10 en naranja.

Código en Python[9]

A diferencia del código en R, el código en Python imprime todas las distancias mínimas desde el vértice 1 hasta el resto de vértices del grafo.

La función dijkstra() calcula el camino más corto desde el vértice 1 utilizando el algoritmo de Dijkstra, mientras que la función minDistance() itera sobre todos los vértices en el grafo y compara sus distancias desde el vértice 1, actualizando este valor si encuentra una distancia más corta. Luego devuelve el vértice de destino junto con la distancia mínima.

El resultado al compilar el código es el siguiente:

Tabla 2.1. Distancia desde el origen a los vértices

Vértice	Distancia desde el origen
0	0
1	3
2	20
3	14
4	3
5	2
6	9
7	8
8	13
9	21

En el código anterior, los vértices empiezan en 0. Es decir, el vértice 10 en el código anterior es el vértice 9 en el código actual.

Si se comparan los dos códigos, se puede ver que efectivamente la distancia mínima desde el vértice 1 hasta el 10 es 21.

2.1.4. Complejidad del algoritmo

La complejidad computacional, en el ámbito de la informática y la teoría de la computación, se refiere al estudio y la medición de la cantidad de recursos computacionales (como tiempo, espacio de memoria y otros) que un algoritmo o problema particular requiere para su ejecución. Es un campo esencial para comprender la eficiencia y el rendimiento de algoritmos y desempeña un papel fundamental en la teoría de la computación.

Es importante la complejidad computacional ya que permite comparar la eficiencia de diferentes algoritmos para resolver un mismo problema, identificando aquellos que requieren menos recursos computacionales y, por lo tanto, son más eficientes.

Para medir la complejidad computacional es necesario determinar la complejidad temporal que se refiere al tiempo que un algoritmo o programa necesita para ejecutarse en función del tamaño de la entrada. Se mide en términos del número de operaciones básicas realizadas por el algoritmo y se expresa típicamente en notación "O grande" que describe cómo crece el tiempo de ejecución en relación con el tamaño de la entrada. También es necesario determinar la complejidad espacial que se refiere a la cantidad de memoria o espacio de almacenamiento necesario para ejecutar un algoritmo en función del tamaño de la entrada. Al igual que la complejidad temporal, se mide en términos del crecimiento en relación con el tamaño de la entrada. También se expresa en notación O, pero esta vez se refiere al espacio de memoria requerido.

En definitiva, un algoritmo es más eficiente que otro si es capaz de resolver el mismo problema más rápido y con los recursos mínimos necesarios. [10]

El algoritmo de Dijkstra es un algoritmo eficiente de complejidad $O(n^2)$ donde n es el número de vértices.

2.2. Algoritmo de Bellman-Ford

2.2.1. Introdución

Alfonso Shimbel sugirió este algoritmo en 1955, sin embargo, recibió el nombre de Richard Bellman y Lester Ford Jr., quienes lo publicaron en 1958 y 1956, respectivamente. En 1959, Edward F. Moore publicó una variante del algoritmo, que a menudo se conoce como el algoritmo Bellman-Ford-Moore.[11]

La misión de este algoritmo es buscar el camino más corto en un grafo, al igual que el de Dijkstra con la diferencia de que este algoritmo tiene en cuenta los pesos negativos.

Este algoritmo es muy útil para las redes de comunicaciones ya que encuentra la ruta más corta entre dos routers considerando los retrasos o los costos asociados a las conexiones.

2.2.2. Algoritmo

A continuación se muestra el pseudocódigo asociado al algoritmo de Bellmand-Ford. [1]

Igual que en el de Dijkstra se calculará el camino más corto desde el vértice s hasta el vértice v, $d(u)$ es la distancia desde el vértice origen s hasta el vértice u, $d'(u)$ almacena temporalmente las distancias de $d(u)$ y $w(uv)$ es la función que nos devuelve el peso entre el vértice u y el vértice v.

BELLMANFORD $(G, w, s; d)$

$d(s) = 0, V = G.vertices$

for $v \in V \backslash \{s\}$ **do** $d(v) = \infty$

repeat

for $v \in V$ **do** $d'(v) = d(v)$

for $v \in V$ **do** $d(v) = min(d'(v), min\{d'(u) + w(uv) : uv \in E\})$ **end**

while $d(v) = d'(v)\ \forall v \in V$

2.2.3. Implementación del algoritmo

Se estudiará un ejemplo de un grafo con pesos negativos a través de un código en Python.[12]

Para ver un ejemplo del algoritmo se crea un grafo de la siguiente manera:

```
g = Graph(5)
# (x, y, w) -> arista de `x` a `y` con peso `w`
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)
```

Primero se define el vértice 0 como el vértice de origen y a continuación se calcula la distancia desde el origen a los demás vértices, teniendo como resultado lo siguiente:

Tabla 2.2. Distancia desde el origen a los vértices

Vértice	Distancia desde el origen
0	0
1	-1
2	2
3	-2
4	1

2.2.4. Complejidad del algoritmo

La complejidad del algoritmo es $O(|VE|)$ donde V es la cantidad de vértices y E la de aristas.

El algoritmo de Bellman-Ford es eficiente para grafos pequeños o medianos, pero puede volverse menos eficiente para grafos grandes o densos debido a su complejidad.[13]

3

Árboles

El científico checo Otakar Borůvka desarrolló el primer algoritmo conocido para encontrar un árbol generador mínimo en 1926. Quería resolver el problema de encontrar una red eficiente de electricidad en Moravia. Hoy en día, este algoritmo se conoce como el algoritmo de Borůvka. Otro algoritmo importante fue desarrollado por Vojtěch Jarník en 1930, y puesto en práctica por Robert Clay Prim en 1957. Edsger Wybe Dijkstra lo redescubrió en 1959 y lo llamó el algoritmo de Prim. Otro algoritmo significativo se llama algoritmo de Kruskal, y fue publicado por Joseph Kruskal en 1956.[14]

Dado un grafo no dirigido, el problema del árbol generador mínimo es un problema combinatorio que consiste en encontrar un árbol que pasa por todos los vértices y que la suma de sus aristas es la de menor peso.

Hay múltiples ámbitos en los que se aplican los árboles generadores mínimos. Una de las aplicaciones más clásicas es en la construcción de redes, como pueden ser redes eléctricas, redes telefónicas o redes de carreteras. Para este último caso se podrían considerar diferentes ciudades que se quieren conectar mediante la construcción de una red de carreteras, de forma que se minimice la longitud total de la red.

En este capítulo se abordarán los algoritmos de Prim y Kruskal, que son dos métodos utilizados en la teoría de grafos para encontrar árboles generadores mínimos.

El algoritmo de Prim es el primero y el más sencillo de los algoritmos de la teoría de grafos para encontrar un árbol generador mínimo en un grafo no dirigido. El algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. La solución al problema del árbol generador propuesto por Prim se basa en la idea de ir conectando vértices secuencialmente hasta alcanzarlos a

todos.[15]

El algoritmo de Kruskal también tiene como objetivo encontrar un árbol generador mínimo. Para ello se busca la arista de menor peso y se une los vértices siempre y cuando no se forme un ciclo. Esto se hace hasta que todos los vértices estén comprendidos en el árbol. Para comprobar que el árbol generador mínimo este hecho se revisa que el numero de aristas sea igual al número de vértices menos 1.[16]

3.1. Algoritmo de Kruskal

3.1.1. Introducción

Este algoritmo fue escrito y publicado por Joseph Kruskal en 1956.[16]

Como ya se mencionó anteriormente, el objetivo del algoritmo es construir un árbol (subgrafo sin ciclos) formado por aristas sucesivamente seleccionadas de mínimo peso a partir de un grafo con pesos en las aristas.

Este método permite que los pesos de las aristas sean negativos.

Se utiliza en la vida cotidiana en situaciones en las que se busca minimizar costos o distancias de manera eficiente como por ejemplo en las redes de transporte y cableado.

3.1.2. Algoritmo

Sea $G = (V, A)$ un grafo y $V = \{1, \ldots, n\}$ el conjunto de vértices de ese grafo. Definimos w la función peso. Las aristas de G están ordenadas según su peso, es decir, $A = a_1, ..., a_m$ con $w(a_1) \leq \ldots \leq w(a_m)$.[1]

KRUSKAL$(G, W; T)$

$T = \emptyset$

for $k = 1$ **to** m **do**

if a_k does not form a cycle together with some edges ot T

then append a_k to T **end if**

end

3.1.3. Implementación del algoritmo

Para ver un ejemplo de este algoritmo se utilizará un código realizado en Python.[17]

Primero se define la matriz de la siguiente forma:

```
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)
```

Al implementar este código se tiene como resultado lo siguiente:

Aristas en el árbol generador mínimo construido:

- $2 \rightarrow 3 = 4$
- $0 \rightarrow 3 = 5$
- $0 \rightarrow 1 = 10$

Coste mínimo del árbol generador mínimo: 19

3.1.4. Complejidad del algoritmo

La complejidad del algoritmo de Kruskal es de $O(|E|log|E|)$, donde |E| es el número de aristas en el grafo. Esto significa que el tiempo de ejecución del algoritmo crece logarítmicamente con respecto al número de aristas en el grafo, lo que lo hace muy eficiente para grafos densos o con un gran número de aristas.[1]

3.2. Algoritmo de Prim

3.2.1. Introducción

Robert Prim en 1957 descubrió un algoritmo para la resolución del problema del árbol generador mínimo.[18]

Este algoritmo es muy similar al de Kruskal con la diferencia de que el algoritmo de Prim se basa en seleccionar y agregar vértices al árbol uno a la vez, mientras que Kruskal se centra en seleccionar y agregar aristas al árbol en orden creciente según su peso.

3.2.2. Algoritmo

Sea $G = (V, A)$ un grafo donde el conjunto de aristas se denota por $V = \{1, \ldots, n\}$ y la matriz de adyacencia A_v. La función peso se llamará w.[1]

PRIM$(G, w; T)$

$g(1) = 0, S = \emptyset, T = \emptyset$

for $i = 2$ to n **do** $g(i) = \infty$

while $S \neq V$ **do**

choose $i \in V \backslash S$ such that $g(i)$ is minimal; $S = S \cup \{i\}$

if $i \neq 1$ then $T = T \cup \{e(i)\}$ **end if**

for $j \in A_i \cap (V \backslash S)$ **do**

if $g(j) > w(ij)$ **then** $g(i) = w(ij); e(j) = ij$ **end if**

end

end

3.2.3. Implementación del algoritmo

El lenguaje de programación utilizado en este ejemplo será Python.[19]

Para ello, primero se define el grafo a través de su matriz de adyacencia:

$$\begin{bmatrix} 0 & 2 & 0 & 6 & 0 \\ 2 & 0 & 3 & 8 & 5 \\ 0 & 3 & 0 & 0 & 7 \\ 6 & 8 & 0 & 0 & 9 \\ 0 & 5 & 7 & 9 & 0 \end{bmatrix}$$

Al compilar este código se tiene como resultado el árbol generador mínimo del grafo mencionado anteriormente, esto es:

Arista	Peso
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

3.2.4. Complejidad del algoritmo

Si se utiliza una matriz de adyacencia, la complejidad sería $O(|V|^2)$, donde |V| es la cantidad de vértices, ya que se necesita recorrer todas las celdas de la matriz.[1]

4

Circuitos y ciclos

Muchos problemas de planificación y organización (servicio de recogida de basura, servicio de inspección de instalaciones de gas, servicio de reparto de cartas, servicios de mensajería,...) se pueden tratar como problemas de optimización de recorridos. Además, estos problemas están relacionados con la búsqueda de ciclos eulerianos y circuitos hamiltonianos, que representan recorridos que cubren todas las aristas o vértices de un grafo, respectivamente.

Aunque pueden parecer similares, los caminos eulerianos y hamiltonianos tienen una gran diferencia, los eulerianos pasan por cada arista una y solo una vez, es decir, un ciclo o circuito euleriano es un camino cerrado que recorre cada arista exactamente una vez mientras que los caminos hamiltonianos son una sucesión de aristas adyacentes, que visita todos los vértices del grafo una única vez. Si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano.

Los problemas de optimización combinatoria consisten en el diseño de rutas, o tours, sobre los vértices o los enlaces de un grafo que cumplan una serie de condiciones y que tengan un coste total mínimo.

El objetivo frente a este tipo de problemas es crear algoritmos que puedan resolverlos eficientemente. Estos problemas suelen representarse mediante grafos con numerosos vértices y conexiones entonces el número de posibles soluciones es demasiado grande. Por lo tanto, se requieren algoritmos eficientes que puedan encontrar soluciones óptimas en un tiempo razonable.[20]

En este capítulo se analiza el problema del viajante, el cual posee un circuito hamiltoniano. Se implementa el problema mediante un código en Python en un grafo dado.

También se estudia el problema del cartero chino cuya base son los caminos eulerianos. Se pone en práctica el código en Python y se ejecutará para tres ejemplos.

4.1. Circuito Hamiltoniano

En 1857 Sir William Rowan Hamilton inventó un juego al que llamó *Un viaje alrededor del mundo*. Este juego era un tablero donde había inscrito un dodecaedro regular y en cada vértice del dodecaedro había un número que representaba diferentes ciudades. La idea del juego era encontrar un recorrido circular a lo largo del dodecaedro, visitando cada ciudad exactamente una vez y que el vértice de llegada fuera el mismo que el de salida.

El recorrido que representa este juego se llama circuito o ciclo hamiltoniano. Estos circuitos son muy útiles actualmente para los problemas de rutas y planificación, ya que ayudan a determinar la ruta más eficiente para visitar un conjunto de lugares que pueden ser de interés.[21]

4.1.1. Problema del viajante

Un ejemplo donde encontramos un circuito hamiltoniano es el problema del viajante.

El problema del viajante, o también conocido como TSP, es un problema donde un viajante quiere visitar n ciudades partiendo de una de ellas y llegando a la misma ciudad de partida, de forma que se minimice la distancia total recorrida.

Si bien no se conoce exactamente el origen del problema, en 1832, en una guía para viajantes, se menciona el problema e incluye ejemplos de viajes entre Alemania y Suiza, pero no contiene un tratamiento matemático del mismo.

El TSP fue definido por el matemático irlandés W.R Hamilton y por el matemático británico Thomas Kirkman.

Todo indica que la forma general del TSP fue estudiada por primera vez en Viena y Harvard por diferentes matemáticos durante los años 1930 destacándose Karl Menger quien sobresalió por sus definiciones de los problemas.

Hassler Whitney de la Universidad de Princeton introdujo el nombre “travelling salesman problem” poco después.

A lo largo de los años 1950 a 1960 el problema fue aumentando su popularidad entre los científicos de Europa y Estados Unidos. Una importante aportación fue la de George Dantzing, Delbert Ray Fulkerson y Selmer M. Johnson de la Corporación RAND en Santa Monica, quienes definieron el problema como Programación Lineal en Enteros y desarrollaron para solucionarlo el método de Planos Cortantes.

En los 90, Applegate, Bixby, Chvátal, y Cook construyeron el programa Concorde, el cual es usado en muchos de los registros de soluciones recientes.

En 1991, Gerhard Reinelt publicó el TSPLIB, una colección de instancias de pruebas de dificultad variable, que es usada por muchos grupos de investigación para comparar resultados.

Cook y otros, en 2006, obtuvieron un recorrido óptimo para 85.900 ciudades dado por un problema de diseño de microchip.

Actualmente, TSPLIB es la colección de ejemplos (instancias) más larga conocida.[22]

Formulación

Parámetros

Un viajante quiere visitar n ciudades partiendo de una de ellas y llegando a la misma ciudad de partida, de forma que se minimice la distancia total recorrida.

- $V = \{1, ..., n\}$ es el conjunto de ciudades o localizaciones.
- $c_{ij} = c_a$ es el costo (distancia) de ir de i a j, $i, j \in V, i \neq j$.

$G = (V, A)$, $A = \{(i, j) : i, j \in V, i \neq j\}$ grafo dirigido.

Asumimos que la ciudad de partida es la 1 y cada ciudad se visita solo una vez.

Variables de decisión

$$X_a = \begin{cases} 1 & \text{si se pasa por el arco } a = (i, j) \\ 0 & \text{en otro caso} \end{cases}$$

$\delta^+(i) = \delta^+\{i\} = \{(i, j) \in A : j \neq i\}$

$\delta^-(i) = \delta^-\{i\} = \{(j, i) \in A : j \neq i\}$

Modelo

mín $\sum_{a \in A} c_a x_a$

sujeto a:

$\sum_{a \in \delta^+(i)} x_a = 1 \ \forall i \in V$

$\sum_{a \in \delta^-(i)} x_a = 1 \ \forall i \in V$

$\sum_{a \in A(S)} x_a \leq |S|\text{-}1 \ \forall S \subseteq V$

Implementación del algoritmo

A través de un código realizado en Python se puede ver un ejemplo del Problema del Viajante. [23]

La definición del grafo se realiza mediante su matriz de adyacencia.

$$\begin{bmatrix} 12 & 30 & 33 & 10 & 45 \\ 56 & 22 & 9 & 15 & 18 \\ 29 & 13 & 8 & 5 & 12 \\ 33 & 28 & 16 & 10 & 3 \\ 1 & 4 & 30 & 24 & 20 \end{bmatrix}$$

Se empieza en el vértice 1 y se debe pasar por todos los nodos del grafo.

El resultado de compilar este código es:

Camino mínimo: 1 4 5 2 3 1

Coste mínimo: 55

Complejidad del problema

Cualquier problema de optimización puede ser transformado en un problema de decisión. En el análisis de complejidad se manejan, por lo tanto, los problemas de decisión, que incluyen a los de optimización.

Si se encuentra solución a un problema de decisión, entonces se encuentra también solución a un problema de optimización. Al estudiar los problemas de decisión, se pueden encontrar varias clases: Una es la clase de problemas P que está formada por todos aquellos problemas de decisión para los cuales se tiene un algoritmo de solución que se ejecuta en tiempo polinomial en una máquina determinista. Otra de las clases es la clase de problemas NP que está formado por todos aquellos problemas de decisión para los cuales existe un algoritmo de solución que se ejecuta en tiempo polinomial en una (hipotética) máquina no determinista. Dicho de otro modo, no se ha encontrado un algoritmo determinista que lo resuelva en tiempo polinomial.[24]

En el ámbito de la teoría de complejidad computacional, el TSP pertenece a la clase de problemas NP-completos, estos son un subconjunto de los problemas NP. Los problemas NP-completos son los problemas más difíciles en NP, y tienen la propiedad de que cualquier problema NP puede reducirse a ellos en tiempo polinómico. Por lo tanto, se supone que no hay ningún algoritmo eficiente para la solución del TSP. En otras palabras, el número de posibles soluciones es tan elevado que si pretendemos que el algoritmo encargado de la búsqueda de la solución óptima deba verificar una a una no tendremos tiempo de cálculo para hallarlo: es probable que en el peor de los casos el tiempo de resolución de

cualquier algoritmo para el TSP aumente exponencialmente con el número de ciudades, por lo que incluso en algunos casos de tan sólo cientos de ciudades se tardará bastantes años de CPU para resolverlos de manera exacta.[25]

Variantes del problema del viajante

La primera variación del clásico problema del viajante es el problema del viajante de comercio con cuello de botella (BTSP, por sus siglas en inglés).

Conceptualmente, el BTSP es bastante similar al TSP. Las restricciones son idénticas en el sentido de que cada ubicación debe ser visitada una vez.

Además, el viaje debe terminar en la misma ciudad donde comenzó. La única diferencia de formulación real es que en lugar de minimizar la distancia total del viaje, el objetivo es minimizar la distancia del viaje interurbano más largo. Con este único cambio en la función objetivo, el clásico TSP se convierte en un problema "minimax" donde el objetivo es minimizar la distancia máxima entre dos ciudades. [26]

Otra variante del TSP es el problema del viajante de comercio con ventanas temporales (TSPTW). Consiste en encontrar la ruta más rápida para que un vendedor visite un grupo de clientes dentro de una cantidad de tiempo predeterminada.

Este problema es aplicable en varios sectores como transporte, logística y manufactura, donde cumplir con los plazos es crucial para garantizar un servicio o entrega puntual y aumentar la eficiencia. El TSPTW incluye restricciones de ventanas temporales. Cumplir con estas restricciones es importante para garantizar que los clientes sean atendidos de manera rápida y efectiva.[27]

4.2. Ciclo Euleriano

Los ciclos y caminos eulerianos son uno de los conceptos más influyentes de la teoría de grafos en las matemáticas y la tecnología innovadora. Estos circuitos y caminos, fueron descubiertos por primera vez por Euler en 1736.

Actualmente, los ciclos eulerianos son muy útiles en sistemas de transporte, para planificar rutas eficientes y garantizar la conectividad completa entre todos los puntos.

En ingeniería eléctrica, los ciclos eulerianos pueden ser útiles para diseñar circuitos eléctricos eficientes y garantizar que la corriente fluya de manera óptima.[28]

4.2.1. Problema del cartero chino

El siguiente problema, se refiere a un cartero que debe entregar el correo para un sistema (conectado) de calles: nuestro cartero desea minimizar la distancia total que tiene que recorrer configurando su ruta de manera adecuada. Este problema es conocido en la actualidad generalmente como el problema del cartero chino.

El problema fue estudiado originalmente por el matemático chino Kwan Mei-Ko en 1960, cuyo documento fue traducido al inglés en 1962. El nombre original "problema del cartero chino" fue acuñado en su honor. [29]

Formulación

Parámetros[30]

- $V = \{1, ..., n\}$ es el conjunto de nodos.
- E conjunto de arcos.

Variables de decisión

x_{ij} : número de veces que veces que el arco $(i.j)$ es recorrido en el vehículo que parte del nodo i y termina en el nodo j.

$c_{i,j}$: La longitud para recorrer el arco (i, j) comenzando desde el nodo i y terminando en el nodo j.

Modelo

mín $\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$

sujeto a:

$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = 0 \ ; \ i = 1, 2, ..., n \ \forall i \in V$

$x_{ij} + x_{ji} \geq 1 \ \forall (i, j) \in E$

$x_{ij}, x_{ji} \geq 0$

Implementación del algoritmo

Igual que los algoritmos anteriores, se utilizará un código realizado en Python.[31]

Se definen tres grafos diferentes para ver tres ejemplos.

```
# Grafo 1:
g1 = Graph(6)
g1.add_edge(0, 1, 1)
```

```
g1.add_edge(0, 2, 2)
g1.add_edge(1, 3, 3)
g1.add_edge(1, 5, 7)
g1.add_edge(2, 3, 5)
g1.add_edge(2, 5, 4)
g1.add_edge(3, 4, 6)
g1.add_edge(4, 5, 8)
graphs.append(g1)

# Grafo 2
g2 = Graph(5)
g2.add_edge(0, 1, 1)
g2.add_edge(0, 2, 4)
g2.add_edge(0, 3, 4)
g2.add_edge(1, 2, 2)
g2.add_edge(1, 3, 2)
g2.add_edge(1, 4, 3)
g2.add_edge(2, 3, 5)
g2.add_edge(3, 4, 6)
graphs.append(g2)

# Grafo 3
g3 = Graph(5)
g3.add_edge(0, 1, 1)
g3.add_edge(0, 2, 3)
g3.add_edge(1, 2, 5)
g3.add_edge(1, 3, 2)
g3.add_edge(1, 4, 4)
g3.add_edge(2, 3, 9)
g3.add_edge(3, 4, 4)
graphs.append(g3)
```

Este código devuelve las aristas eulerizadas de cada grafo, el camino desde cada nodo y la distancia mínima.

El término eulerizar se refiere al proceso de agregar aristas adicionales a un grafo para convertirlo en un grafo euleriano, lo que facilita la búsqueda del ciclo de menor distancia.

```
Problema del cartero chino:

Grafo 1:

   Aristas Eulerizadas: [[1, 3], [2, 5]]

   Camino partiendo de 0: [0, 2, 5, 4, 3, 1, 5, 2, 3, 1, 0]
   Camino partiendo de 1: [1, 3, 4, 5, 2, 5, 1, 3, 2, 0, 1]
   Camino partiendo de 2: [2, 5, 4, 3, 1, 5, 2, 3, 1, 0, 2]
   Camino partiendo de 3: [3, 1, 5, 2, 5, 4, 3, 2, 0, 1, 3]
   Camino partiendo de 4: [4, 5, 2, 5, 1, 3, 2, 0, 1, 3, 4]
   Camino partiendo de 5: [5, 2, 5, 4, 3, 1, 3, 2, 0, 1, 5]

   Distancia: 43

------------------------------

Grafo 2:

   Aristas Eulerizadas: [[0, 2]]

   Camino partiendo de 0: [0, 2, 3, 4, 1, 3, 0, 2, 1, 0]
   Camino partiendo de 1: [1, 4, 3, 2, 0, 3, 1, 2, 0, 1]
   Camino partiendo de 2: [2, 0, 3, 4, 1, 3, 2, 1, 0, 2]
   Camino partiendo de 3: [3, 4, 1, 3, 2, 0, 2, 1, 0, 3]
   Camino partiendo de 4: [4, 3, 2, 0, 3, 1, 2, 0, 1, 4]

   Distancia: 31

------------------------------

Grafo 3:

   Aristas Eulerizadas: [[2, 3]]

   Camino partiendo de 0: [0, 2, 3, 4, 1, 3, 2, 1, 0]
   Camino partiendo de 1: [1, 4, 3, 2, 3, 1, 2, 0, 1]
   Camino partiendo de 2: [2, 3, 4, 1, 3, 2, 1, 0, 2]
   Camino partiendo de 3: [3, 2, 3, 4, 1, 2, 0, 1, 3]
   Camino partiendo de 4: [4, 3, 2, 3, 1, 2, 0, 1, 4]

   Distancia: 37

------------------------------
```

Complejidad del problema

El CPP se divide básicamente en tres categorías: dirigido, no dirigido y mixto, dependiendo de las direcciones de las aristas en un grafo. El CPP dirigido y no dirigido pueden resolverse mediante algoritmos polinomiales. Por lo tanto, pertenecen a la clase P. En el CPP mixto, algunas de las rutas son dirigidas y otras son no dirigidas, y este problema pertenece a la clase de problemas NP-hard.[30]

Variantes del problema del cartero chino

El problema del cartero chino en un grafo dirigido (Problema del Cartero Chino Dirigido, DCPP por sus siglas en inglés) es un caso especial del problema CPP, que se enfoca en encontrar el recorrido de menor costo en un grafo dirigido, donde cada arco debe ser recorrido al menos una vez.

Si un multigrafo tiene un recorrido euleriano, entonces este recorrido es una solución del DCPP. El algoritmo para construir el recorrido euleriano tiene una complejidad temporal. El recorrido euleriano existe en un multigrafo dirigido si el multigrafo está fuertemente conectado y el número de arcos que salen de cada vértice es igual al número de arcos que entran en él.

Un multigrafo que cumple las condiciones para la existencia del recorrido euleriano se llama multigrafo euleriano. Si el multigrafo original no es euleriano, entonces para la solución del DCPP algunos arcos deben ser atravesados más de una vez. En otras palabras, el multigrafo debe ser complementado con copias de algunos de los arcos para convertirse en multigrafo euleriano, de manera que el costo de las copias adicionales de los arcos sea mínimo.[32]

5

Otros algoritmos

En este capítulo se estudian algunos algoritmos que tienen aplicaciones prácticas significativas en campos como la informática, la ingeniería y la logística.

Se analizan algoritmos relevantes como el algoritmo de Floyd-Warshall, utilizado para encontrar el camino más corto entre todos los pares de vértices en un grafo, y el algoritmo de Borůvka, que es otra metodología para encontrar el árbol generador mínimo.

También se estudia un algoritmo de búsqueda. El algoritmo de Búsqueda en Profundidad (DFS) que es fundamental para la exploración de grafos y tiene múltiples aplicaciones en la resolución de problemas de conectividad.

5.1. Algoritmo de Floyd-Warshall

5.1.1. Introducción

El algoritmo de Floyd-Warshall fue publicado por Robert Floyd en 1962.[33]

Este algoritmo se utiliza para encontrar los caminos más cortos entre todos los pares de vértices en un grafo. Este algoritmo es altamente eficiente y puede manejar grafos con pesos en las aristas tanto positivos como negativos, lo que lo convierte en una herramienta versátil para resolver una amplia gama de problemas de redes y conectividad.[34]

5.1.2. Algoritmo

A continuación se muestra el pseudocódigo del algoritmo donde se calcula la distancia mínima entre todos los pares de vértices del grafo G.[1]

En este pseudocódigo $G = (V, A)$ un grafo y $V = \{1, \ldots, n\}$ el conjunto de vértices de ese grafo. Se define $w(ij)$ como la matriz de adyacencia que re-

presenta los pesos de las aristas del grafo G y $d(i,j)$ es la matriz que almacena las distancias más cortas entre todos los pares de vértices en el grafo. Se define $w(ij) = \infty$ si no existe arista entre los vértices ij

FLOYD $(G, w; d)$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    if $i \neq j$ **then** $d(i,j) = w(ij)$ **else** $d(i,j) = 0$ **end**
  end
end
for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $d(i,j)$=$min(d(i,j), d(i,k) + d(k,j))$
    end
  end
end

5.1.3. Implementación del algoritmo

Para este ejemplo se utiliza el lenguaje de programación Python.[35]

Se necesita un grafo que viene dado de la siguiente forma:

$$\text{floyd\_warshall}(4, [[1,3,-2],[2,1,4],[2,3,3],[3,4,2],[4,2,-1]])$$

Primero se define el número de vértices, en este caso, 4. Luego se indica el vértice inicial, vértice final y distancia.

Al compilar el código, este es el resultado:

par	dist	camino
1 → 2	-1	1 → 3 → 4 → 2
1 → 3	-2	1 → 3
1 → 4	0	1 → 3 → 4
2 → 1	4	2 → 1
2 → 3	2	2 → 1 → 3
2 → 4	4	2 → 1 → 3 → 4
3 → 1	5	3 → 4 → 2 → 1
3 → 2	1	3 → 4 → 2
3 → 4	2	3 → 4
4 → 1	3	4 → 2 → 1
4 → 2	-1	4 → 2
4 → 3	1	4 → 2 → 1 → 3

Se muestra los pares de vértices a los que se le calcula la distancia mínima y el respectivo camino más corto entre los vértices.

5.1.4. Complejidad del algoritmo

Este algoritmo tiene una complejidad de $O(|V|^3)$, donde $|V|$ es el número de vértices en el grafo.[1]

5.2. Algoritmo de Borůvka

5.2.1. Introducción

Este algoritmo fue publicado por primera vez en 1926 por Otakar Borůvka como un método para construir una red eléctrica eficiente para Moravia. Este algoritmo es frecuentemente denominado el algoritmo de Sollin, especialmente en la literatura de computación paralela ya que fue redescubierto por Georges Sollin en 1965.[36]

La misión de este algoritmo es encontrar el árbol generador mínimo en un grafo.

5.2.2. Algoritmo

Ahora se presenta el pseudocódigo asociado al algoritmo de Borůvka.[1]

Sea $G = (V, E)$ un grafo donde $V = \{1, \ldots, n\}$ es el conjunto de vértices de ese grafo. Se define w como la función de peso para la cual dos aristas distintas siempre tienen pesos distintos.

BORUVKA $(G, w; T)$

for $i = 1$ **to** n do $V_i = i$ **end**

$T = \emptyset$; $M = \{V_1, \ldots, V_n\}$;

while $|T| < n - 1$ **do**

for $U \in M$ **do**

find an edge $e = uv$ with $u \in U$, $v \notin U$ and $w(e) < w(e')$ for all edges $e' = u'v'$ with $u' \in U$, $v' \notin U$;

find the component U' containing v;

$T = T \cup \{e\}$;

end

for $U \in M$ **do** $MERGE(U, U')$ **end**

end

Donde MERGE es una unión disjunta.

5.2.3. Implementación del algoritmo

Para el siguiente ejemplo se debe definir un grafo, este viene dado de la siguiente forma:

```
g = Graph(9)
g.add_edge(0, 1, 4)
g.add_edge(0, 6, 7)
g.add_edge(1, 6, 11)
g.add_edge(1, 7, 20)
g.add_edge(1, 2, 9)
g.add_edge(2, 3, 6)
g.add_edge(2, 4, 2)
g.add_edge(3, 4, 10)
g.add_edge(3, 5, 5)
g.add_edge(4, 5, 15)
g.add_edge(4, 7, 1)
g.add_edge(4, 8, 5)
g.add_edge(5, 8, 12)
g.add_edge(6, 7, 1)
```

```
g.add_edge(7, 8, 3)
```

A compilar el código, se obtiene el siguiente resultado:

```
---------Creando el MST------------
{0: 1, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8}
arista_añadida [0 - 1]
peso_añadido: 4

{0: 1, 1: 1, 2: 4, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8}
arista_añadida [2 - 4]
peso_añadido: 2

{0: 1, 1: 1, 2: 4, 3: 5, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8}
arista_añadida [3 - 5]
peso_añadido: 5

{0: 1, 1: 1, 2: 4, 3: 5, 4: 4, 5: 5, 6: 6, 7: 4, 8: 8}
arista_añadida [4 - 7]
peso_añadido: 1

{0: 1, 1: 1, 2: 4, 3: 5, 4: 4, 5: 5, 6: 4, 7: 4, 8: 8}
arista_añadida [6 - 7]
peso_añadido: 1

{0: 1, 1: 1, 2: 4, 3: 5, 4: 4, 5: 5, 6: 4, 7: 4, 8: 4}
arista_añadida [7 - 8]
peso_añadido: 3

{0: 4, 1: 4, 2: 4, 3: 5, 4: 4, 5: 5, 6: 4, 7: 4, 8: 4}
arista_añadida [0 - 6]
peso_añadido: 7

{0: 4, 1: 4, 2: 4, 3: 4, 4: 4, 5: 4, 6: 4, 7: 4, 8: 4}
arista_añadida [2 - 3]
peso_añadido: 6

El peso total del árbol generador mínimo es: 29
```

Este código crea el árbol generador mínimo y muestra cual es el peso total de este.[37]

5.2.4. Complejidad del algoritmo

El algoritmo de Borůvka tiene una complejidad del $O(|E| \log |V|)$ donde $|E|$ es el número de aristas del grafo G y $|V|$ el número de vértices.[1]

5.3. Algoritmo de búsqueda en profundidad.

5.3.1. Introducción

La búsqueda en profundidad (DFS) es un algoritmo para el desplazamiento o la búsqueda de árboles o estructuras de datos de un grafo. Fue descrito por primera vez por Charles Pierre Trémaux, un matemático y cartógrafo francés, en su trabajo sobre la solución de laberintos en la década de 1830.[38]

5.3.2. Algoritmo

Sea $G = (V, E)$ un grafo y s un vértice de G.

DFS$(G, s; nr, p)$

for $v \in V$ **do** $nr(v) = 0$; $p(v) = 0$ **end**

for $e \in E$ **do** $u(e) = false$ **end**

$i = 1$; $v = s$; $nr(s) = 1$;

repeat

while there exists $w \in A_v$ with $u(vw) = false$ **do**

choose some $w \in A_v$ with $u(vw) = false$; $u(vw) = true$;

if $nr(w) = 0$ **then** $p(w) = v$; $i = i + 1$; $nr(w) = i$; $v = w$ **end**

end

$v = p(v)$

until $v = s$ **and** $u(sw) = true$ for all $w \in A_s$

En este pseudocódigo nr etiqueta los vértices según el orden en el que son alcanzados y $p(w)$ es el vértice desde el cual se accedió a w.[1]

5.3.3. Implementación del algoritmo

A continuación se muestra un ejemplo donde se hace un recorrido en profundidad de un grafo.[39]

Sea el siguiente grafo:

```
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
```

```
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 1)
```

En este caso g es el grafo y $g.addEdge(i, j)$ crea las aristas entre los vértices i y j.

Si se compila el código, el resultado es el siguiente:

A continuación se muestra el Recorrido en Profundidad (comenzando desde el vértice 2):

$$2 \rightarrow 0 \rightarrow 1 \rightarrow 3$$

Esto indica que el recorrido en profundidad comenzando desde el vértice 2 visitó los vértices en el orden 2, 0, 1 y 3.

5.3.4. Complejidad del algoritmo

La complejidad de este algoritmo es $O(|V| + |E|)$, donde $|V|$ es el número de vértices y $|E|$ es el número de aristas del grafo.[39]

Conclusiones

El objetivo principal de este trabajo ha sido estudiar algunos problemas de conectividad en grafos y redes. Se han explicado algunos conceptos esenciales para poder comprender los algoritmos y problemas que se analizan a lo largo del trabajo.

Dos de los algoritmos más importantes para la búsqueda del camino mínimo en un grafo son el algoritmo de Dijkstra y el de Bellman-Ford que, tras estudiarlos, se llega a la conclusión de que aunque el resultado sea el mismo, el de Bellman-Ford es más útil para grafos con pesos negativos pues el de Dijkstra no funciona en este tipo de grafos.

En cuanto a la búsqueda del árbol generador mínimo destacan el algoritmo de Kruskal y el de Prim, estos algoritmos comparten objetivo pero su enfoque, complejidad y requisitos del grafo son diferentes, lo que los hace más adecuados para diferentes tipos de problemas.

También se estudian dos problemas clásicos de circuitos en grafos y redes. El problema del viajante el cual busca un recorrido mínimo entre n ciudades pasando una vez por cada una de ellas. Este problema es muy útil en algunos aspectos de la vida cotidiana y tras estudiarlo se llega a la conclusión que a pesar de su complejidad sigue siendo un problema de investigación activa. El otro problema que se analiza es el problema del cartero chino que se refiere a un cartero que debe entregar el correo, el cartero desea minimizar la distancia total que tiene que recorrer configurando su ruta de manera adecuada. Igual que el problema del viajante, el problema del cartero chino es bastante complejo pero a la vez muy útil.

Finalmente, se han estudiado varios algoritmos adicionales que son de gran interés en el contexto de los problemas de conectividad en grafos y redes.

Bibliografía

[1] JUNGNICKEL, Dieter. *LaTeX: Graphs,Networks and Algorithms* 4nd edition.

[2] *Graph Online*. Disponible en: https://graphonline.ru/es/.

[3] *Los puentes de Königsberg*. Disponible en: http://prometeo.matem.unam.mx/recursos/Licenciatura/IPM_UAM_CUAJIMALPA//scorm_player/2062/content/index.html.

[4] *Teoría de Grafos*. Disponible en: https://www.grapheverywhere.com/teoria-de-grafos/.

[5] *Fundamentos y aplicaciones de la teoría de grafos. Grafos eulerianos y hamiltonianos. Diagramas en árbol. (2ª Parte)*. Disponible en: https://core.ac.uk/download/pdf/235864433.pdf.

[6] *Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization*. Disponible en: https://iopscience.iop.org/article/10.1088/1757-899X/917/1/012077.

[7] *Algoritmo de Dijkstra*. Disponible en: https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra.

[8] *Dijkstra Algorithm using R*. Disponible en: https://rpubs.com/yogesh665/DJA.

[9] *How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm*. Disponible en: https://colab.research.google.com/drive/1iS0jy3BBo-TuRNnPLEB-UK6VxWkFauJP?usp=sharing.

[10] *Complejidad computacional*. Disponible en: https://jeffrychaves.com/diccionario/complejidad-computacional/

[11] *Bellman-Ford Algorithm*. Disponible en: https://shanmukhchowdary147.medium.com/bellman-ford-algorithm-21ac42a8cf3d

[12] *Bellman-Ford-Algorithm-Python*. Disponible en: https://colab.research.google.com/drive/1sZD5bs-L4pQDOyp2dQnIUCEfoAWEvlKv?usp=sharing

[13] *Algoritmo de Bellman-Ford*. Disponible en: https://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/bellman-ford

[14] *Minimum spanning tree*. Disponible en: https://simple.wikipedia.org/wiki/Minimum_spanning_tree

[15] *El algoritmo de Prim*. Disponible en: http://alejandrosanchezyali.blogspot.com/2020/11/al-algoritmo-de-prim.html

[16] *Algoritmo de Kruskal*. Disponible en: https://jcrd0730.wixsite.com/estr/single-post/2016/05/25/algoritmo-de-kruskal-1

[17] *Kruskal's Minimum Spanning Tree (MST) Algorithm*. Disponible en: https://colab.research.google.com/drive/1haOC7VDuKL-86j-xYjEZCE2gYvXLr3m7?usp=sharing

[18] *Algoritmo de Prim*. Disponible en: https://es.wikipedia.org/wiki/Algoritmo_de_Prim#:~:text=El%20algoritmo%20fue%20diseñado%20en,DJP%20o%20algoritmo%20de%20Jarnik.

[19] *Prim's Algorithm for Minimum Spanning Tree (MST)*. Disponible en: https://colab.research.google.com/drive/10aQQ54RBNA8mPoqgjwj9zy5pXUNcbIy8?usp=sharing

[20] *La combinatoria poliédrica y los problemas de las rutas de vehículos*. Disponible en: https://www.researchgate.net/publication/28311535_La_combinatoria_poliedrica_y_los_problemas_de_las_rutas_de_vehiculos

[21] *William Hamilton: el vándalo matemático que inventó un juego de mesa*. Disponible en: https://www.bbvaopenmind.com/ciencia/matematicas/william-hamilton-vandalo-matematico-invento-juego-mesa/

[22] *Problema del viajante*. Disponible en: https://es.wikipedia.org/wiki/Problema_del_viajante#:~:text=El%20TSP%20tiene%20diversas%20aplicaciones,como%20la%20secuenciación%20de%20ADN.

[23] *Travelling Salesman Problem (Greedy Approach)*. Disponible en: https://colab.research.google.com/drive/1N4kNOG1Xm4q5MqaYtXXxiJIsURecAuxV?usp=sharing

[24] *Algorítmica y Lenguajes de Programación*. Disponible en: http://di002.edv.uniovi.es/~dani/asignaturas/transparencias-leccion19.PDF

[25] *Problema del viajante de comercio*. Disponible en: http://blog.electricbricks.com/2010/06/travelling-salesman/

[26] *Variants of the Traveling Salesman Problem*. Disponible en: https://www.researchgate.net/publication/333165892_Variants_of_the_Traveling_Salesman_Problem

[27] *What is Traveling Salesman Problem with Time Windows (TSPTW)?*. Disponible en: https://www.upperinc.com/glossary/route-optimization/traveling-salesman-problem-with-time-windows-tsptw/

[28] *Eulerian Cycles: Why Are They So Unique, and Are They Significant to Us in the 21st Century?*. Disponible en: https://towardsdatascience.com/eulerian-cycles-why-are-they-so-unique-and-are-they-any-significant-to-us-in-the-21st-century-3ca489af585c

[29] *Chinese postman problem*. Disponible en: https://en.wikipedia.org/wiki/Chinese_postman_problem

[30] *An Overview of Chinese Postman Problem*. Disponible en: https://www.researchgate.net/publication/337026526_An_Overview_of_Chinese_Postman_Problem

[31] *chinese-postman-problem*. Disponible en: https://colab.research.google.com/drive/1eR0BhSH9EFTu1XzYExk856svKB14pUW8?usp=sharing

[32] *The Variants of Chinese Postman Problems and Way of Solving through Transformation into Vehicle Routing Problems*. Disponible en: https://www.researchgate.net/publication/326298123_The_Variants_of_Chinese_Postman_Problems_and_Way_of_Solving_through_Transformation_into_Vehicle_Routing_Problems

[33] *Floyd–Warshall algorithm*. Disponible en: https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm

[34] *Floyd Warshall Algorithm*. Disponible en: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

[35] *floyd-warshall-python* . Disponible en: https://colab.research.google.com/drive/1kTgejkwHNU68wmQ5BH1J7psZASKOUATU?usp=sharing

[36] *Borůvka's algorithm*. Disponible en: https://en.wikipedia.org/wiki/Borůvka%27s_algorithm

[37] *Boruvka-s-Algorithm*. Disponible en: https://colab.research.google.com/drive/14PMWyKn8_eoJTsVXQaskB4GZ5XMGnY2q?usp=sharing

[38] *DFS*. Disponible en: https://advanceintelligence.wordpress.com/2014/10/09/heuristica/

[39] *Depth First Search or DFS for a Graph*. Disponible en: https://colab.research.google.com/drive/1mQU0kmU52FD1QmZdT31qDD9M80tNVCTP?usp=sharing

Connectivity Problems in Graphs and Networks

Omaira González Pérez
Facultad de Ciencias • Sección de Matemáticas
Universidad de La Laguna
alu0101100982@ull.edu.es

Abstract

In this work, some basic connectivity problems in graphs and networks are studied. For this purpose, some essential concepts are introduced to subsequently understand the different problems that are presented. Algorithms related to the search for the shortest path, algorithms to find the minimum spanning tree, and classical circuit problems in graphs and networks will be studied. All these problems are accompanied by examples, which are solved using specific codes. Finally, other algorithms of interest are studied.

1. Introduction

A graph is a physical or abstract configuration where there are places or points connected by links. These points are called nodes or vertices, and the connections are referred to as edges (if they do not have a designated direction) or arcs (if they have an assigned direction).

Figure 1: Mixed graph.

Some basic concepts about graphs and networks

- A minimum path is the shortest route between two specific vertices in a graph where the edges have associated weights.
- A minimum spanning tree is a tree that includes all the vertices of the original graph and a subset of edges such that the sum of the magnitudes associated with the edges is minimized.
- A Hamiltonian circuit is a circuit that must pass through each vertex of a graph exactly once.
- An Eulerian cycle is a cycle in the graph that uses each edge exactly once.

2. Algorithms

1. There are several algorithms that search for the minimum path in a graph, but the most prominent ones are Dijkstra's algorithm and Bellman-Ford algorithm.

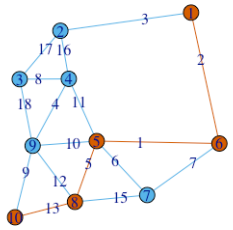


Figure 2: Example of finding the shortest path using the Dijkstra's algorithm in R.

2. When it comes to searching for the minimum spanning tree, the most prominent algorithms are Prim's algorithm and Kruskal's algorithm. Minimum spanning trees are fundamental in graph theory and have numerous practical applications in various fields.

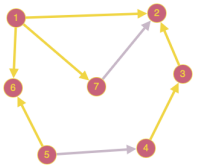


Figure 3: Minimum spanning tree.

3. An example where we find a Hamiltonian circuit is the Traveling Salesman Problem (TSP). The Traveling Salesman Problem, also known as TSP, is a problem where a salesman wants to visit n cities starting from one of them and ending at the same starting city, in such a way that the total distance traveled is minimized.

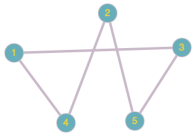


Figure 4: Hamiltonian circuit.

4. The Chinese Postman Problem refers to a postman who must deliver mail for a (connected) system of streets: our postman wants to minimize the total distance he has to travel by arranging his route properly. The Chinese Postman Problem can be solved by finding an Eulerian cycle in a graph.

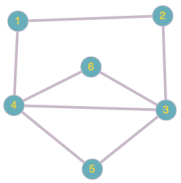


Figure 5: Eulerian cycle.

References

[1] JUNGNICKEL, Dieter. LaTeX: Graphs,Networks and Algorithms 4nd edition.

[2] Teoría de Grafos. Disponible en: https://www.grapheverywhere.com/teoria-de-grafos/.

[3] Fundamentos y aplicaciones de la teoría de grafos. Grafos eulerianos y hamiltonianos. Diagramas en árbol. (2ª Parte). Disponible en: https://core.ac.uk/download/pdf/235864433.pdf.

[4] La combinatoria poliédrica y los problemas de las rutas de vehículos. Disponible en: https://www.researchgate.net/publication/28311535_La_combinatoria_poliedrica_y_los_problemas_de_las_rutas_de_vehiculos.