Trabajo Fin de Máster

Máster en Ingeniería Industrial

Aplicación de la teoría de grafos a problemas de costo

Autor: Eduardo Moyano García

Tutor: Manuel Ordóñez Sánchez

Dep. De Matemática Aplicada II
Escuela Técnica Superior de Ingeniería

Sevilla, 2024

Trabajo Fin de Máster
Máster en Ingeniería Industrial

Aplicación de la teoría de grafos a problemas de costo

Autor: Eduardo Moyano García

Tutor: Manuel Ordóñez Sánchez

Departamento de Matemática Aplicada II

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado: Aplicación de la teoría de grafos a problemas de costo

Autor: Eduardo Moyano García

Tutor: Manuel Ordóñez Sánchez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

En primer lugar, quisiera agradecer a mi familia todo el apoyo que me han mostrado tanto en los buenos como en los malos momentos. Siempre dándome ánimos y haciéndome ver el lado bueno de las cosas cuando yo no lo veía. Siempre contentos con mi esfuerzo y no dándole tanta importancia a los resultados.

Además, quiero acordarme de quiénes han sido fundamentales en esta etapa en la escuela: mis amigos. Principalmente a Jesús, Miguel, Tomás y Gabriel con los que he compartido tantísimas horas de clases, prácticas y biblioteca, pero sin olvidarme de los momentos de descanso y de deporte que con ellos he disfrutado. Mis años en la escuela no hubieran significado tanto para mí si ellos no hubiesen estado siempre ahí.

Por último, agradecer también a todos los profesores y personal de la escuela por su dedicación, esfuerzo y profesionalidad, que siempre han mostrado conmigo y que tendré permanentemente presente en mi etapa profesional.

Eduardo Moyano García

Sevilla, 2024

Resumen

En este trabajo se ha realizado un resumen de los conceptos fundamentales de la teoría de grafos, esenciales para abordar problemas de costo. También se han detallado otros puntos clave como los siguientes:

- Árbol y arborescencia de unión de valor óptimo: en la teoría de grafos un árbol es un grafo conexo y acíclico, es decir, que no tiene ciclos y todos los vértices están conectados. Por otro lado, una arborescencia de unión de valor óptimo es un subconjunto de aristas de un grafo que conecta todos los vértices con el menor peso total posible. La arborescencia de unión de valor óptimo busca minimizar el peso total de las aristas para conectar todos los vértices.
- Camino de menor valor: camino entre dos nodos en un grafo que tiene la menor suma de pesos en las aristas que lo componen. Es decir, es el camino más corto o con el menor costo entre dos nodos específicos en un grafo ponderado.
- Redes de flujo: grafos dirigidos donde cada arista tiene una capacidad máxima de flujo y se busca encontrar la manera más eficiente de enviar un flujo desde un nodo fuente a un nodo destino, respetando las capacidades de las aristas y maximizando el flujo total.
- Administración de proyectos: Técnicas para planificar, organizar, dirigir y controlar proyectos representando las actividades como nodos y arcos en un grafo para facilitar la toma de decisiones y una gestión eficiente que logre alcanzar los objetivos del proyecto en tiempo y forma.

Además se ha explicado cómo aprovechar la herramienta MatLab para aplicar la teoría de grafos en la resolución de problemas, incluyendo comandos para encontrar árboles de unión y caminos de valor mínimo.

En resumen, este trabajo destaca la utilidad de los grafos en la resolución de problemas cotidianos y resalta la importancia de MatLab para su implementación.

Abstract

In this work, a summary of the fundamental concepts of graph theory, essential to address cost problems, has been made. Other key points have also been detailed, such as the following:

- Tree and optimal value union tree: in graph theory a tree is a connected and acyclic graph, that is, it has no cycles and all vertices are connected. On the other hand, an optimal value join tree is a subset of edges of a graph that connects all vertices with the lowest possible total weight. The optimal value join tree seeks to minimize the total weight of edges to connect all vertices.

- Path of least value: path between two nodes in a graph that has the smallest sum of weights on the edges that compose it. That is, it is the shortest path or path with the least cost between two specific nodes in a weighted graph.

- Flow networks: directed graphs where each edge has a maximum flow capacity and the aim is to find the most efficient way to send a flow from a source node to a destination node, respecting the capacities of the edges and maximizing the total flow.

- Project administration: Techniques to plan, organize, direct and control projects, representing activities as nodes and arcs in a graph to facilitate decision making and efficient management that achieves project objectives in a timely manner.

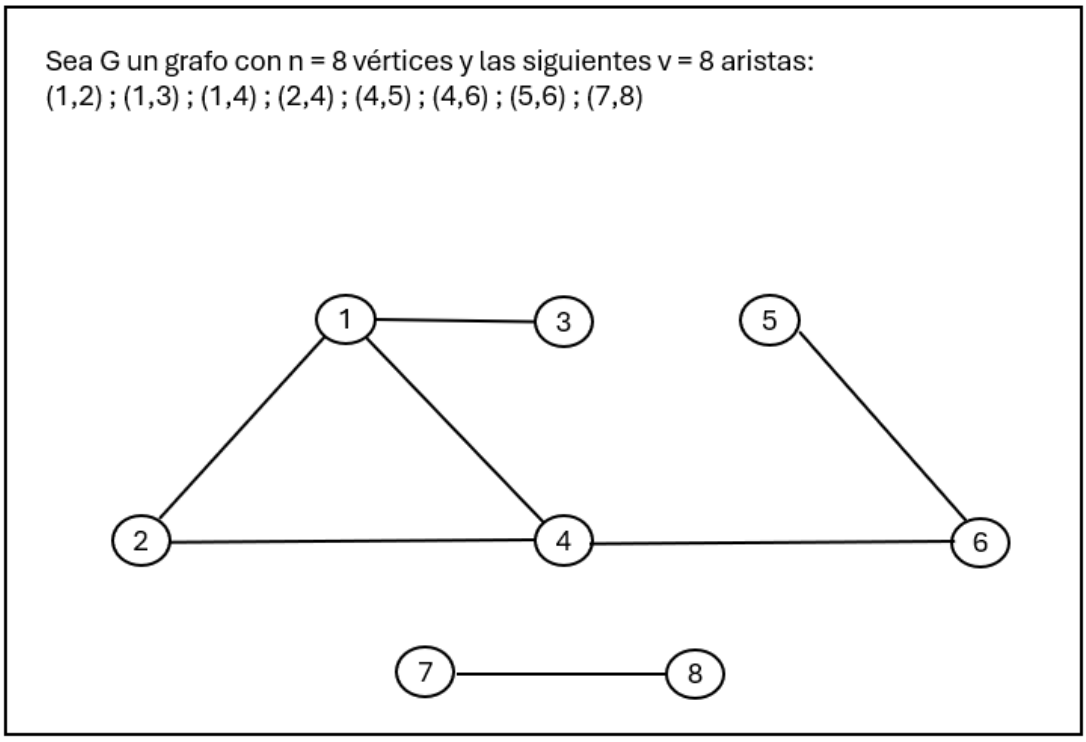
In addition, it has been explained how to take advantage of the MatLab tool to apply graph theory in solving problems, including commands to find union trees and minimum value paths.

In summary, this work highlights the usefulness of graphs in solving everyday problems and highlights the importance of MatLab for its implementation.

Índice

1. INTRODUCCIÓN A LA TEORÍA DE GRAFOS.

Un Grafo es un par G = (V, A), donde V es un conjunto de vértices y A es un conjunto de aristas. Número de vértices: n, número de aristas: m.



Tipos de grafos:

1. Grafo no orientado o no dirigido: Son aquellos en los que el orden de los pares en las aristas no es importante.
    - Una cadena entre los vértices x e y es una sucesión de vértices y aristas que van desde x a y (o desde y hasta x).
    - Un ciclo es una cadena donde coinciden los vértices de salida y llegada.
    - En grafos no orientados, no tiene sentido definir grados de incidencia de entrada o salida.

2. Grafo orientado o dirigido: Son aquellos grafos en los que sí importa el

orden

de los pares en las aristas. En estos grafos, las aristas recibirán el nombre de arcos, y se denotarán por x (vértice inicial) →y (vértice final).

- Un arco (x, x), es decir, un arco donde los vértices inicial y final coinciden, se llama bucle.

3. Adyacencia:
   - Dos arcos se llaman adyacentes si tienen un vértice en común.
   - Dos vértices se dicen adyacentes si hay un arco que los conecta.
   - Un vértice se llama aislado si no es adyacente a ningún otro vértice.
4. Incidencia:
   - Un arco cuyo vértice inicial o final es x se dice incidente en x.
   - El grado de incidencia de un vértice x, denotado por g(x), es el número de arcos incidentes en él.
   - El grado de incidencia de salida (entrada) de un vértice x, denotado por g + (x) (g - (x)), es el número de arcos que tienen a x como vértice inicial (final).
5. Sucesor y antecesor:
   - Si (x, y) es un arco x→y, y se llama sucesor de x y x se llama antecesor o predecesor de y.
   - Γ(x) y Γ - (x) denotan los sucesores y predecesores, respectivamente, del vértice x.
6. Caminos:
   - Un camino de x 1 a x k es una sucesión de vértices y arcos desde x 1 hasta x k: x 1, e 1, x 2, e 2, …, e k-1, x k, donde e i = (x i, x i+1).
   - La longitud de un camino es el número de arcos que lo forman.
7. Alcanzabilidad:

- Un vértice y es alcanzable desde x si hay un camino de x a y. Si lo hay, se llamará descendiente de x. DS(x) denota el conjunto de descendientes de x.
- Así mismo, x se llamará ascendente de y. AS(x) denota el conjunto de ascendientes de x.

8. Circuitos:
    - Un circuito es un camino donde coinciden el vértice inicial y final.
    - Al igual que para los caminos, la longitud del circuito es el número de arcos que lo componen.

Grafo orientado	Grafo no orientado
Camino	Cadena
Circuito	Ciclo
Grado de incidencia de entrada/salida	No tiene sentido el grado de incidencia

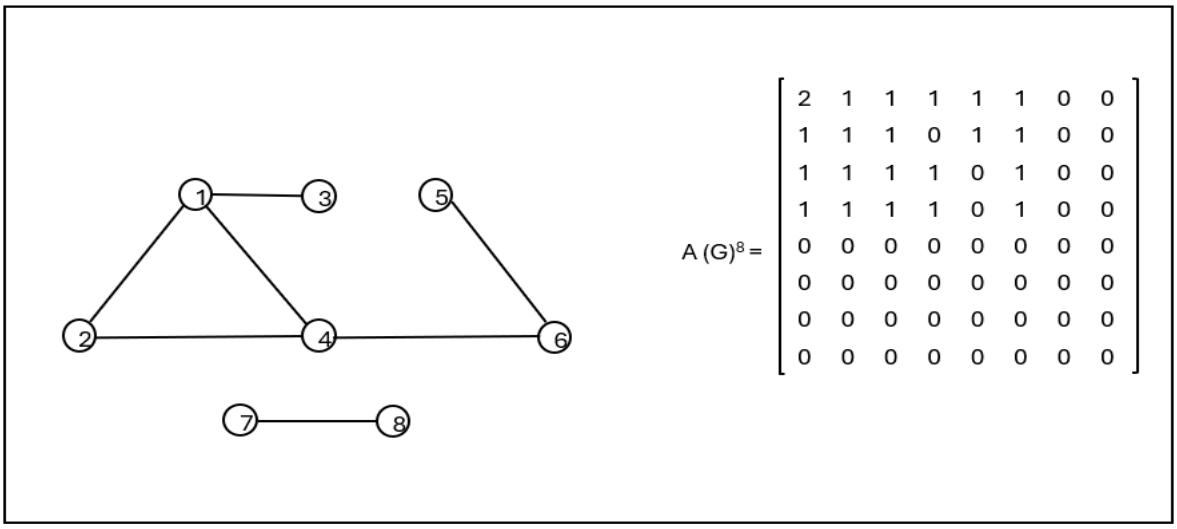
Matriz de adyacencia:

La matriz de adyacencia del grafo G, denotada por A(G), es una matriz nxn definida por:

$$a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in A. \\ 0, & \text{if } (i,j) \notin A. \end{cases}$$

- Si G es no orientado, la matriz es simétrica y g(i) es la suma de la fila i o de la columna o de la columna i.
- Si G es orientado, g + (i) es la suma de la fila i y g - (i) es la suma de la columna i. Γ(i) está formado por los vértices asociados a las columnas con 1 en la fila y Γ - (i) está formado por los vértices asociados a las filas con un 1 en la columna i.

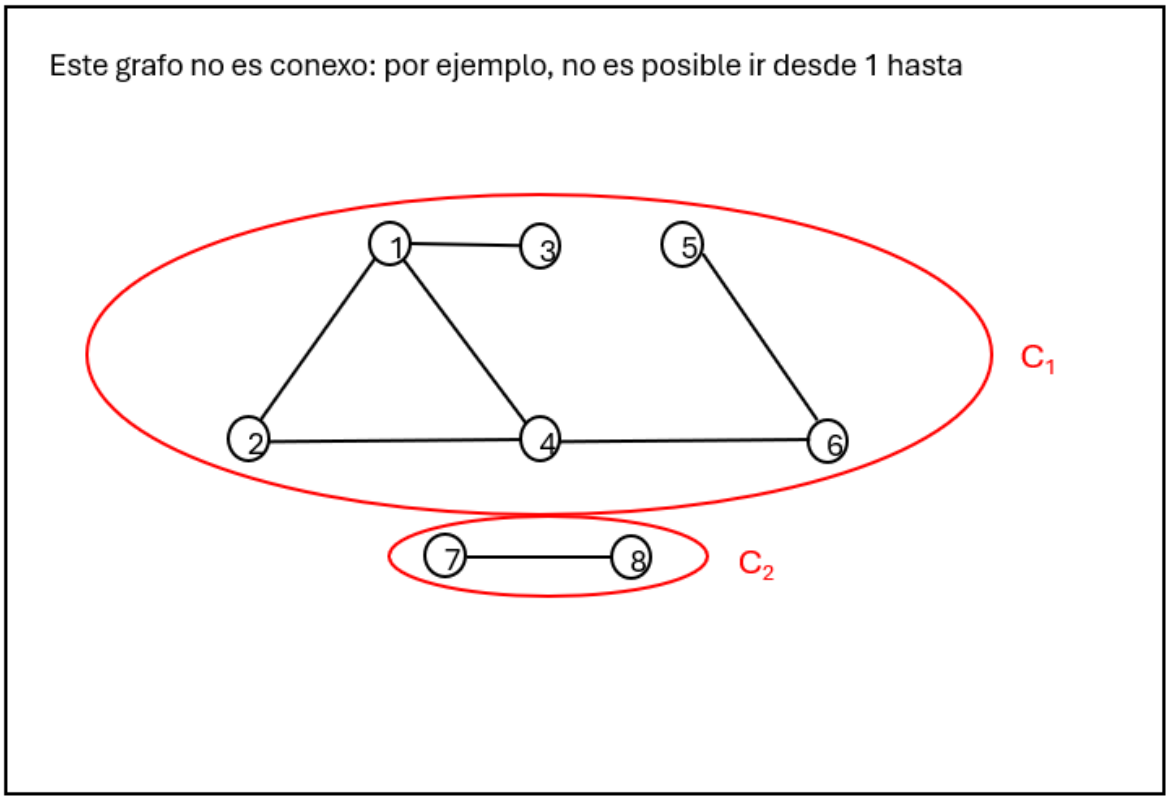
Teorema: Si G es un grafo simple y A(G) P denota la potencia p-ésima de la matriz de adyacencia de G, el valor (a ij P) es el número de caminos que hay de longitud P desde i hasta j.



Conexión para grafos no orientados:

Si un grafo no orientado no es conexo, podemos definir las componentes conexas: C ⊆ V es una componente conexa del grafo si:

- Es posible unir todo par de vértices en C usando una cadena
- Si se añade un vértice más a C, la propiedad anterior ya no se cumpliría



- Para saber si un grafo no orientado es o no convexo, se puede utilizar el siguiente algoritmo:
  - 1º Hacer C=A(G) y k=1
  - 2º Si C tiene una fila con todos sus elementos no nulos, el grafo es conexo.

    En otro caso, ir al paso 3.
  - 3º Hacer k=k+1:
    - 1.- Si k < n, hacer C = C+A(G) k y volver al paso 2.
    - 2.- Si k = n, el grafo no es conexo.

Conexión para grafos orientados:

Se distinguen dos tipos de conexión:

- Conexión débil: un grafo orientado es débilmente conexo si su grafo no orientado asociado no es conexo.
- Conexión (fuerte): un grafo orientado es fuertemente conexo (o conexo) si para todo par de vértices i, j existe un camino de i a j y un camino de j a i.

Para saber si un grafo orientado es o no convexo, se puede utilizar el siguiente algoritmo:

A(G) es la matriz de adyacencia del grafo orientado G.

1º Comienzo con un vértice i, y hacer Ds(i) = {i}, As(i) = {i}.

2º Sea k ∈ Ds(i):

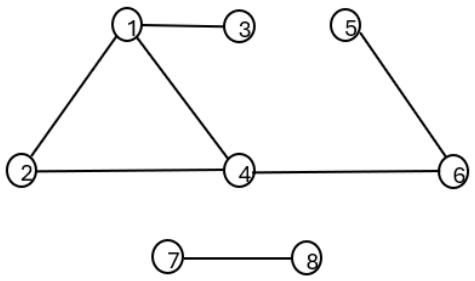
- Añadir a Ds(i) todos los vértices de j de manera que a ij = 1 en A(G).
- Iterar para todos los vértices de Ds(i).

Sea k ∈ As(i):

- Añadir a As(i) todos los vértices j de manera que a ij = 1 en A(G).
- Iterar para todos los vértices de As(i).

3º La componente (fuertemente) conexa en la que está el vértice i viene dada por Ds(i) ⋂ As(i).

4º Eliminar de A(G) las filas y columnas que se corresponden a los vértices de la componente conexa. Volver al paso 1º.



$A(G)^8 =$

$$\begin{bmatrix} 2 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- Paso 1: tomo el vértice 1. Ds(1) = {1}, As(1) = {1}.
- Iterando el paso dos, obtengo Ds(1) = {1,2,3,4,5,6}, As(1) = {1,2,3,4}
- La componente conexa viene dada por Ds(1)∩ As(1) = {1,2,3,4}.

Redes:

- Una red orientada R = (V, A, p) está formada por un grafo orientado G = (V, A) y un vector p que a cada arco e le asigna un

peso p e.

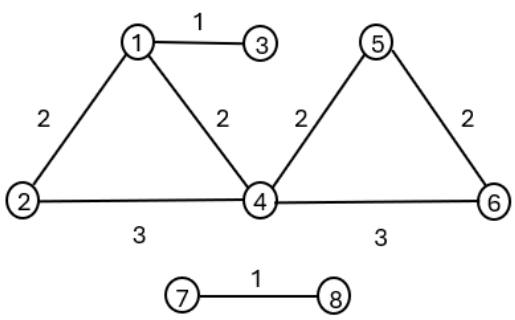
- Una red no orientada R = (V, A, p) está formada por un grafo no orientado G = (V, A) y un vector p que a cada arista e le asigna un peso p e.

En ambos casos, los pesos indican un valor que puede ser una longitud, un tiempo, ...

- Una red no orientada R = (V, A, p) es conexa si G = (V, A) es un grafo no orientado conexo.
- Una red orientada R = (V, A, p) es (débilmente) conexa si G = (V, A) es un grafo orientado (débilmente) conexo.

La matriz de adyacencia con pesos de una red R = (V, A, p), denotada por A(R), es una matriz nxn definida por:

$$a_{ij} = \begin{cases} p_e, & \text{si } e = (i,j) \in A. \\ 0, & \text{si } e = (i,j) \notin A. \end{cases}$$



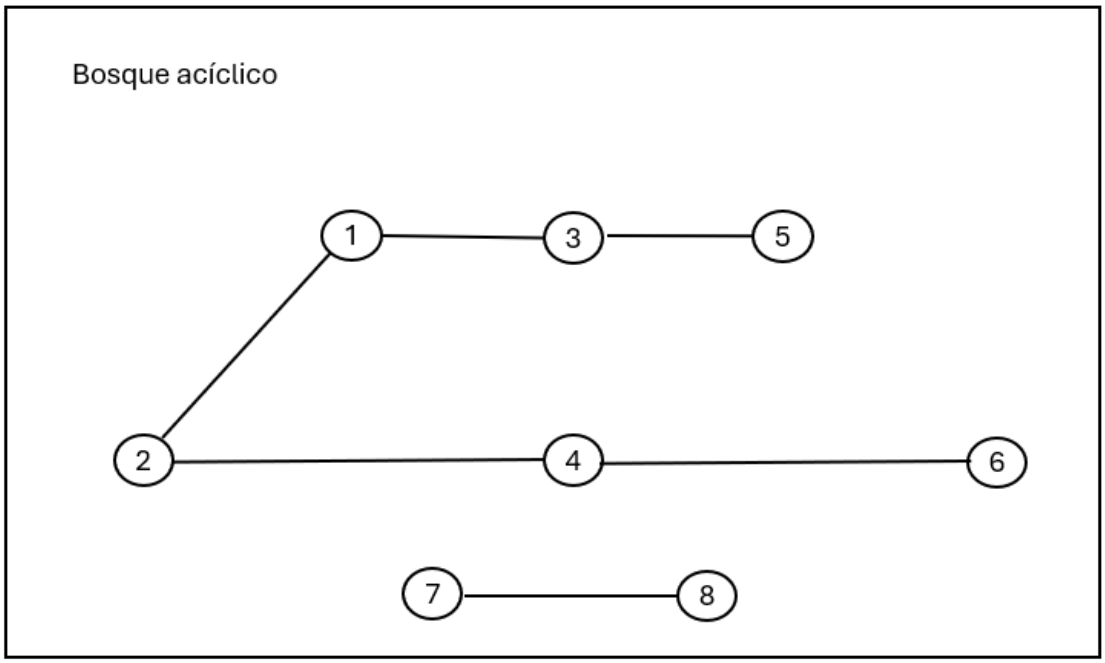
$A(G)^8 =$

$$\begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 2 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

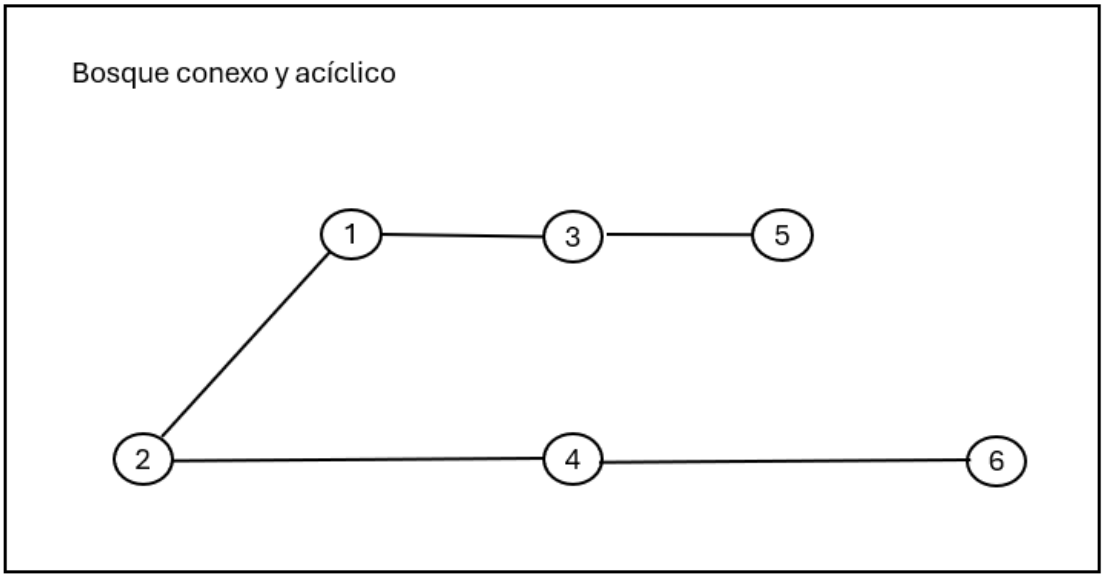
2. ÁRBOL Y ARBORESCENCIA DE UNIÓN DE VALOR ÓPTIMO

Árboles y bosques:

- Un bosque es un grafo G = (V, A) no orientado y acíclico



- Un árbol es un grafo G = (V, A) no orientado, conexo y acíclico

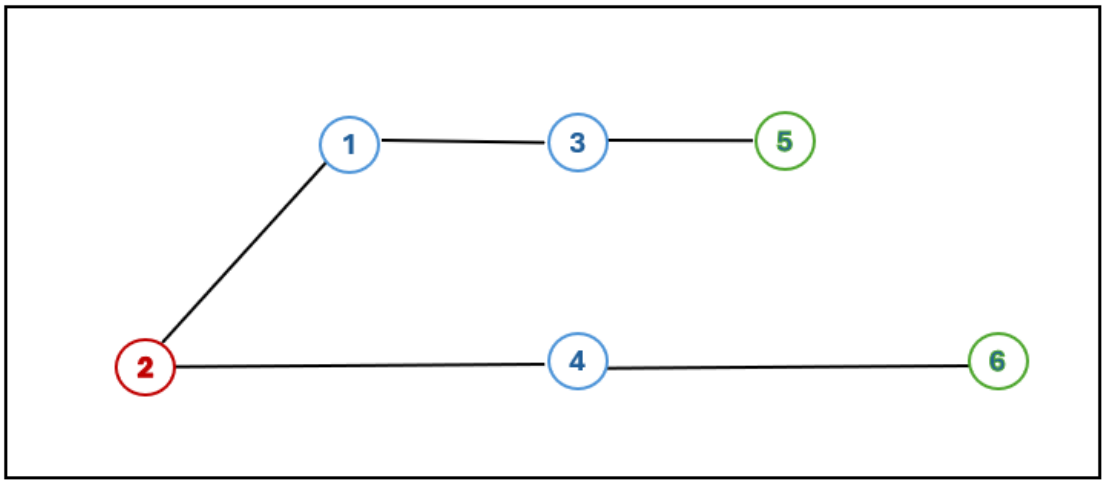


- Un bosque está formado por varios árboles
- La raíz de un árbol es uno de sus vértices

Raíz, nudo, hojas:

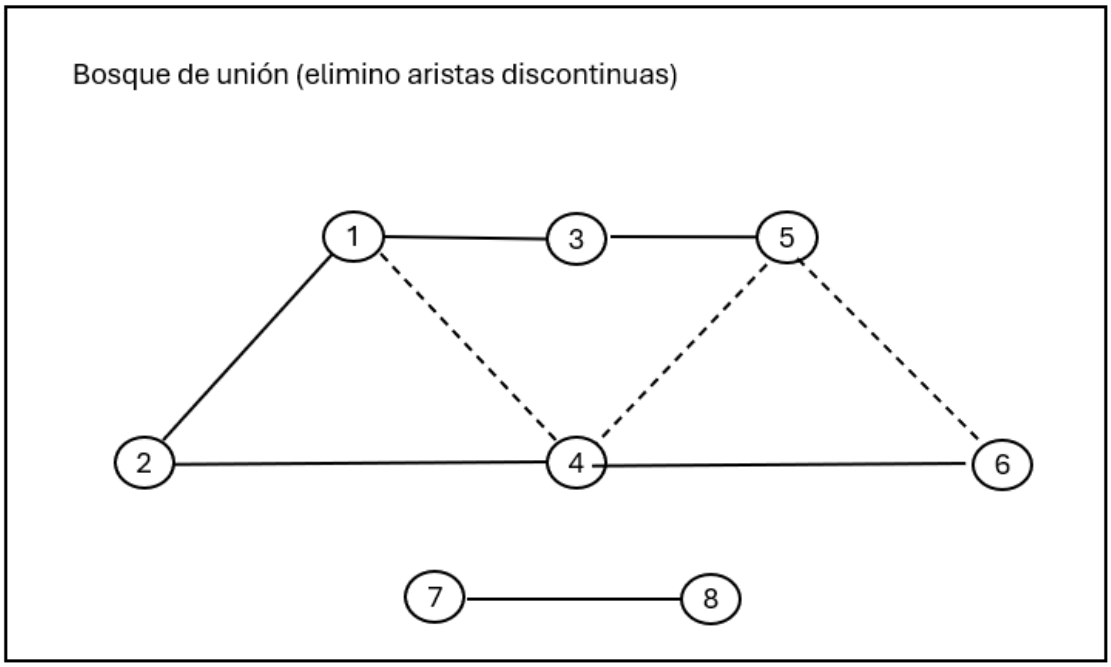
- Raíz: vértice “inicial”
- Nudos: vértices “intermedios” que cumplen g(v) > 1
- Hojas: vértices “finales” que cumplen g(v) = 1

- Nivel de un vértice: longitud mínima de una cadena desde la raíz al vértice

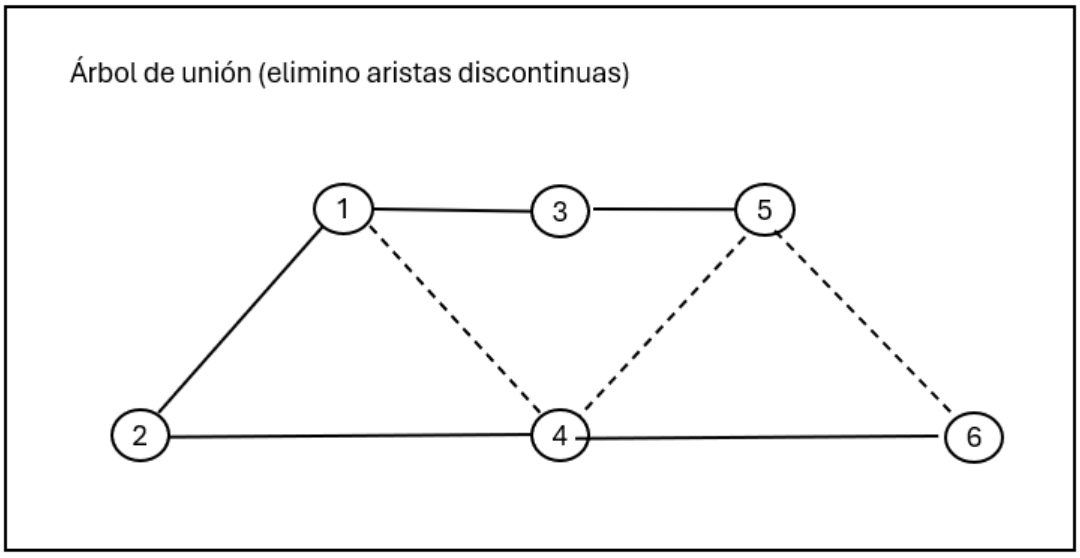


Árbol de unión:

- Dado un grafo G = (V, A) no orientado, un bosque de unión es un bosque $G' = (V, A')$, donde $A' \subseteq A$.
- El bosque de unión siempre existe (por ejemplo, con A = Ø).

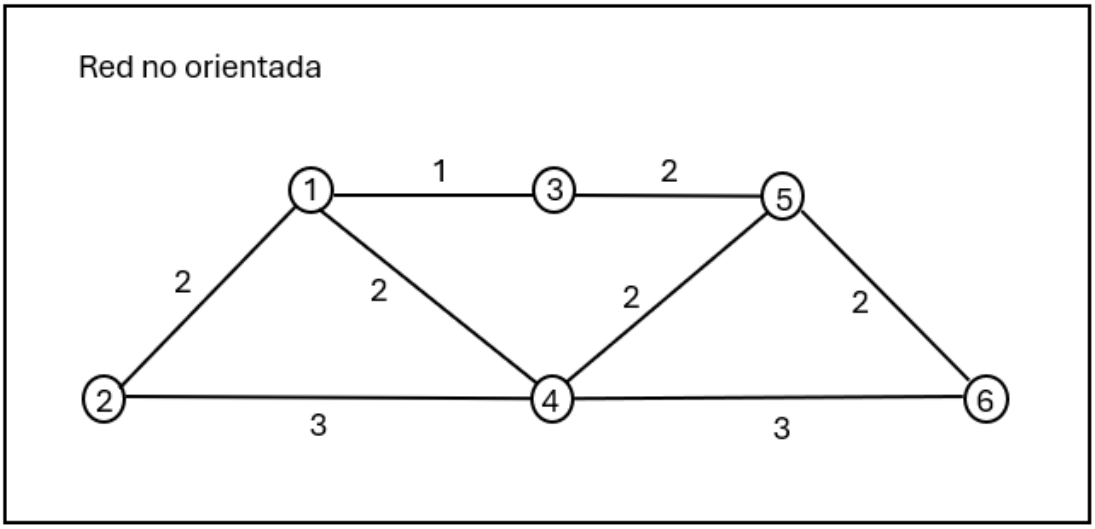


- Dado un grafo G = (V, A) no orientado, un árbol de unión es un árbol $G' = (V, A')$, donde $A' \subseteq A$.
- El árbol de unión existe siempre que el grafo G sea conexo.



Ejemplo de un problema:

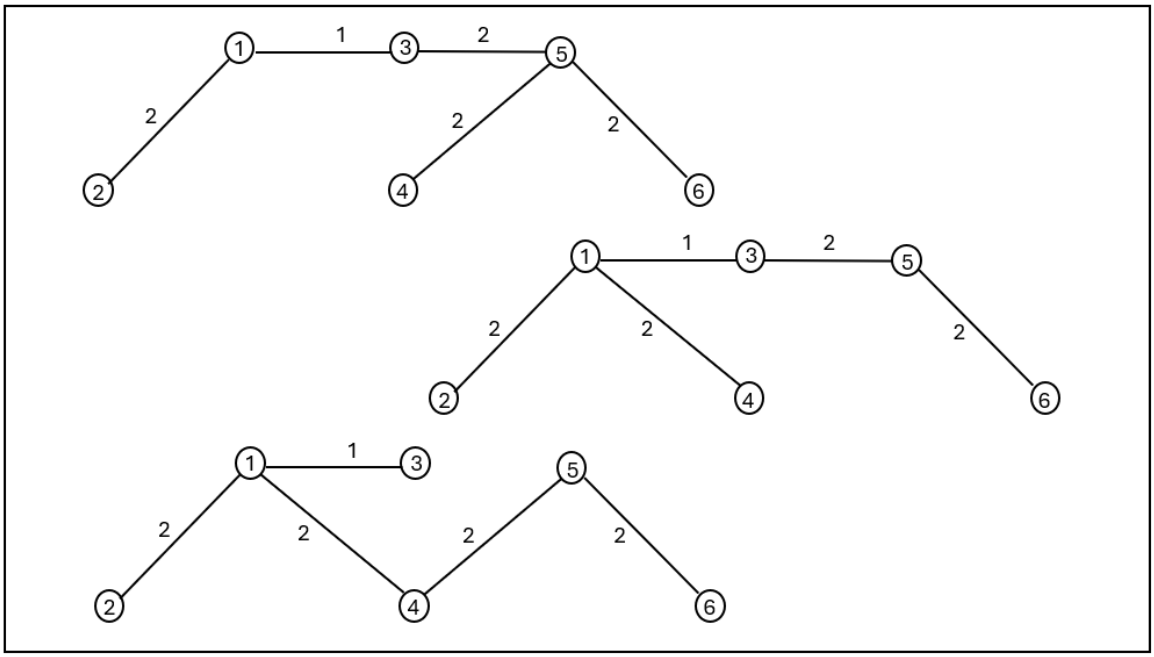
- Descripción:
  - Partimos de una red no orientada: R = (V, A, p).
  - Buscamos una red no orientada R´ = (V, A´, p) de manera que G´ = (V, A´) es un árbol de unión del grafo no orientado G = (V, A), y de manera que se minimiza $\sum_{e \in A'} p_e$.



- Algoritmos:
  - Algoritmo de Kruskal: añade a las aristas de menor valor que no forman ciclos.
    - Partimos de R = (A, V, p) donde G = (V, A) es un grafo no orientado.
    - 1º Defino A´ = Ø, A = A.
    - 2º Sea e^* una arista de manera que $p_{e^*} = \min \{p_e : e \in A\}$.

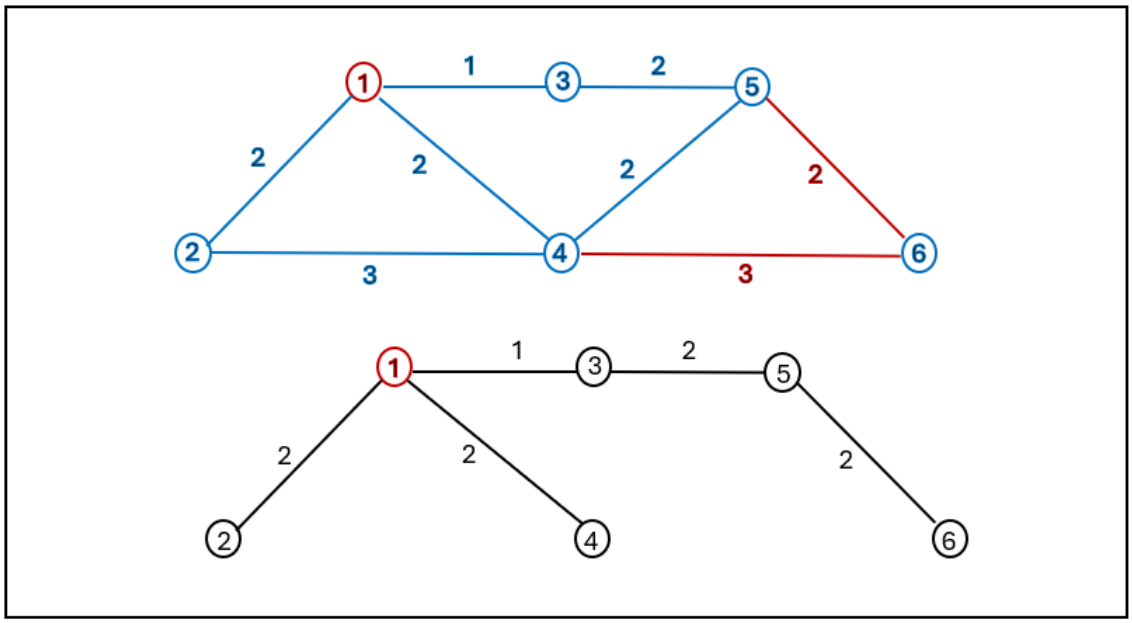
        - Si e^* forma ciclo con las aristas en A´, elimino e* de A.
        - Si e^* no forma ciclo con las aristas de A´, añado e* a A´ y lo elimino de A.
    - 3º Iterar el paso 2º hasta que A = Ø.

- Para el problema ejemplo, mediante este algoritmo se obtienen 3 AUVM distintos:



- Algoritmo de Prim: parte de un vértice dado y va añadiendo vértices y aristas que minimicen el valor y no formen ciclos.
    - Partimos de R = (A, V, p) donde G = (V, A) es un grafo no orientado y seleccionamos $v \in V$.
    - 1º Defino V´ = {v} y A´ = Ø.
    - 2º Sea $e^* = (x^*, y^*) \in A$ de manera que:
        - $p_e^* = \min \{p_e \mid e = (x, y) \text{ con } x \in V', y \notin V'\}$
    - 3º Añado y* a V´ y e* a A´.
    - 4º Volver al paso 2º hasta que V = V´.

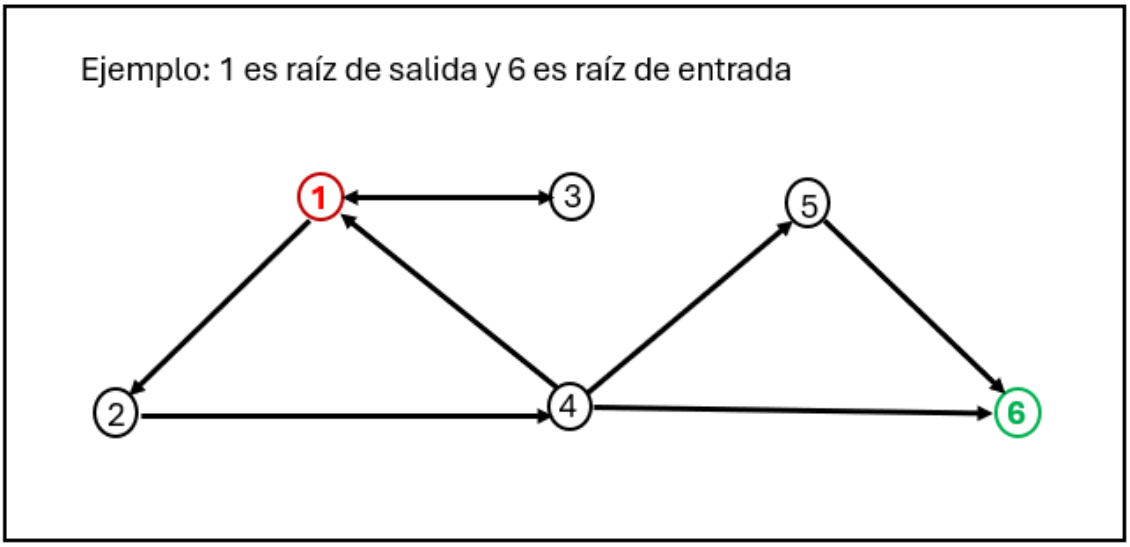
- Para el problema ejemplo, mediante este algoritmo se obtiene el siguiente AUVM:



Raíz de un grafo orientado:

Sea G = (V, A) un grafo orientado

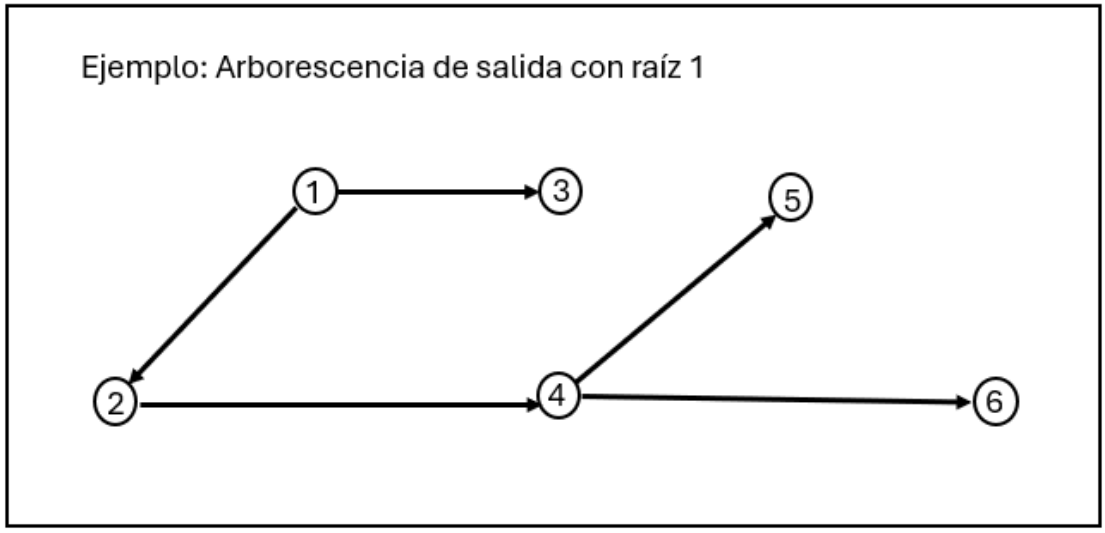
- Un vértice v se dice raíz de salida de G si todo vértice es alcanzable desde v.
- Un vértice v se dice raíz de entrada de G si v es alcanzable desde todo otro vértice.



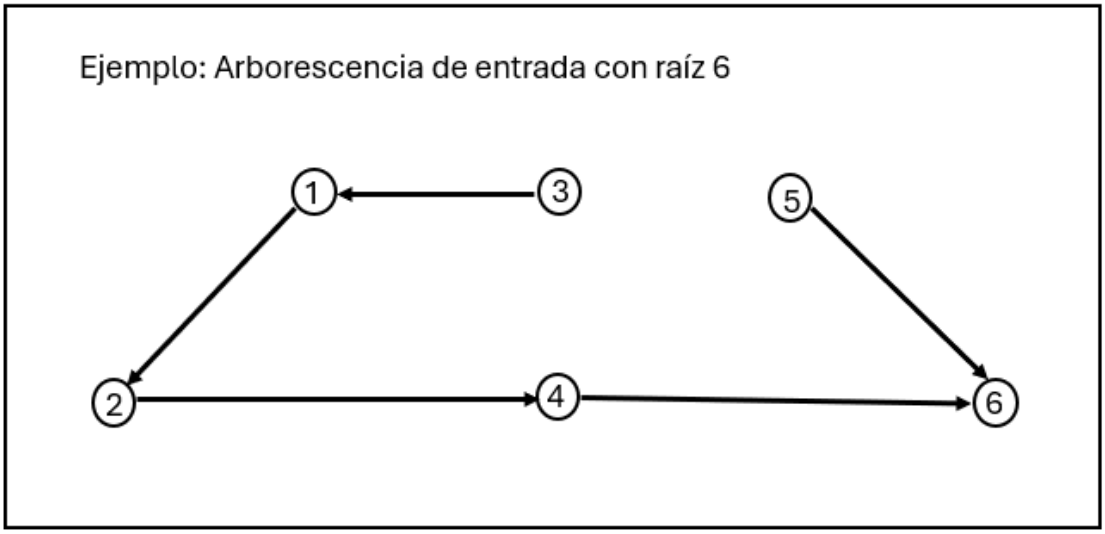
Arborescencias:

Sea G = (V, A) un grafo orientado

- G es una arborescencia de salida si G es conexo, acíclico y tiene una raíz de salida.

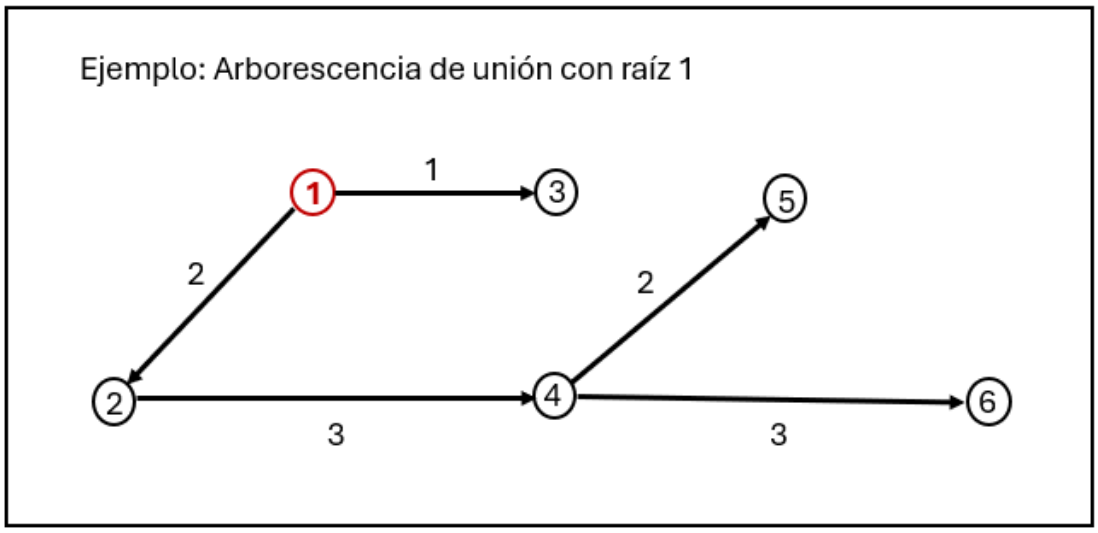


- G es una arborescencia de entrada si G es conexo, acíclico y tiene una raíz de entrada.



Arborescencia de Unión:

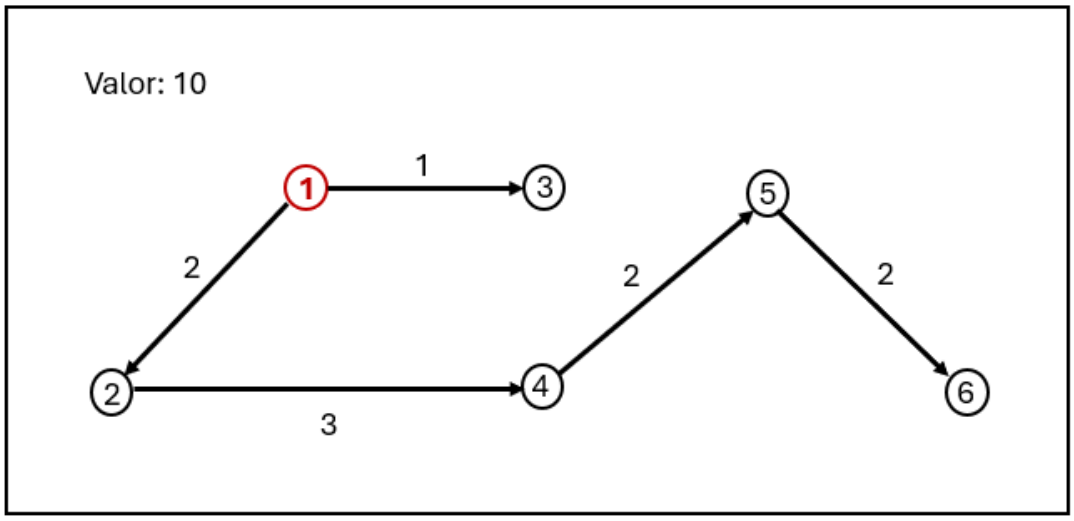
- Dado un grafo orientado G = (V, A), una arborescencia de unión (de entrada/salida) es una arborescencia G´ = (V, A´) (de entrada/salida), donde A´ ⊆ A.
- La arborescencia de unión podría no existir.



Arborescencia de Unión de Valor Mínimo:

- Partimos de una red orientada R = (V, A, p).
- Buscamos una red orientada R´ = (V, A´, p) de manera que G´ = (V, A´) es una arborescencia de unión del grafo G = (V, A), y de manera que minimiza ∑e∈A´pe.

Respecto al ejemplo anterior, la arborescencia de unión de valor mínimo sería la siguiente:



- AUVM fijado una raíz de salida: Buscamos una red orientada R´ = (V, A´, p) de manera que G´ = (V, A´) es una arborescencia de unión del grafo G = (V, A) con una raíz de salida v de manera que minimiza ∑e∈A´pe.
- AUVM fijado una raíz de entrada: Buscamos una red orientada R´ = (V,

A´, p) de manera que G´ = (V, A´) es una arborescencia de unión del grafo G = (V, A) con raíz de entrada v de manera que minimiza ∑e∈A´pe.

3. CAMINO DE MENOR VALOR

Objetivo:

- Partimos de una red orientada R = (V, A, p) conexa. Dados dos vértices i y j, podemos considerar los caminos que van desde el vértice i al vértice j. Dado un camino

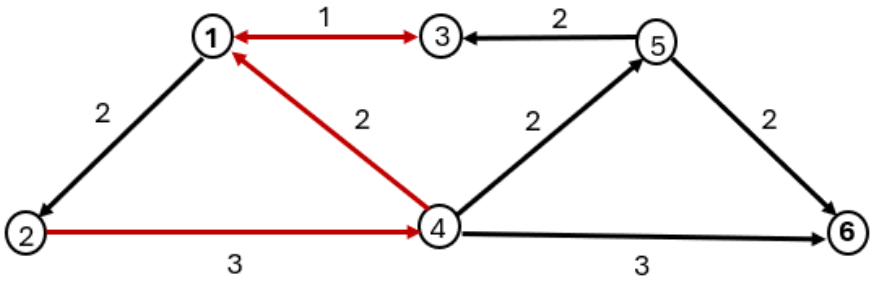
$$i \xrightarrow{p_{i,i_1}} i_1 \xrightarrow{p_{i_1,i_2}} i_2 \xrightarrow{p_{i_2,i_3}} \dots \xrightarrow{p_{i_{k-1},i_k}} i_k \xrightarrow{p_{i_k,j}} j$$

el valor del camino es la suma de los pesos de los arcos que lo forman:

$$p_{i,\,i1} + p_{i1,\,i2} + \dots + p_{ik-1,\,ik} + p_{ik,\,j.}$$

El Camino de Valor Mínimo es un camino que minimiza el valor del camino.

Ejemplo:



Para ir desde el vértice 2 hasta el vértice 3, tenemos, entre otras, las siguientes opciones:

Camino	Valor
$2 \xrightarrow{3} 4 \xrightarrow{2} 1 \xrightarrow{1} 1$	6
$2 \xrightarrow{3} 4 \xrightarrow{2} 5 \xrightarrow{2} 3$	7
$2 \xrightarrow{3} 4 \xrightarrow{3} 6 \xrightarrow{2} 5 \xrightarrow{2} 3$	10

Existencia de camino de menor valor:

El Camino de Menor Valor existirá si no hay ciclos de valor negativo. Es decir,

la suma de los pesos de cada camino no puede ser nunca negativa, aunque sí pueden existir arcos con pesos negativos.

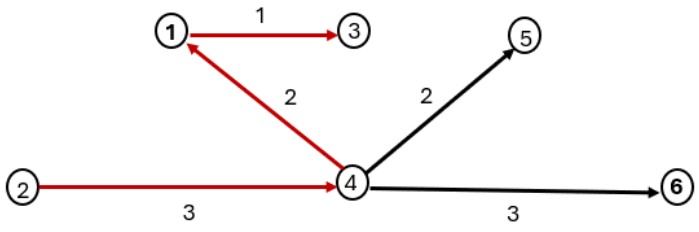
Algoritmos para la búsqueda del CMV

- Algoritmo de Dijkstra: se puede utilizar siempre que los pesos sean no negativos (p>=0).
- Algoritmo de Bellman-Ford: se utilizará cuando haya algún peso negativo.

Descripción del Algoritmo de Dijkstra:

- Objetivo: encontrar un CMV desde el vértice i al resto de vértices.
    - 1º Tomamos V´ = V \ {i}, y definimos d(v) = ∞ para todo v ∉ V´.
    - 2º Para todo v Є Γ(i), actualizo d(v) = pi, v y defino I(v) = i.
    - 3º Sea v Є V´ de manera que d(v) = min {d(j) | j Є V´}.
        - 1. Actualizo V´ = V \ {v}.
        - 2. Para todo j Є Γ(v) ⋂ V´, actualizo d(j) = min {d(j), d(v) + pv, j} y defino I(j) = v.
    - 4º Itero el paso 3º hasta que V´ sea vacío.

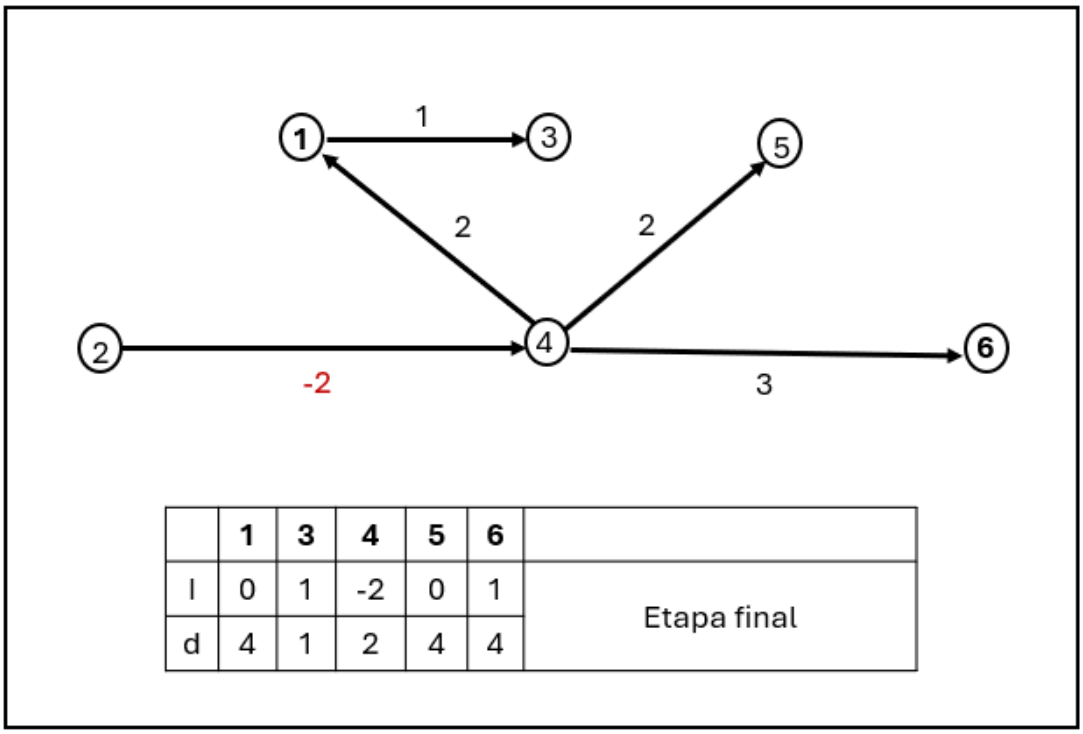
CMV desde el vértice 2 al resto de vértices



	1	3	4	5	6	Etapas
d	∞	∞	3	∞	∞	Inicial
I	-	-	2	-	-	
d	5	∞	3	5	6	Primera iteración
I	4	-	2	4	4	
d	5	6	3	5	6	Segunda iteración
I	4	1	2	4	4	
d	5	6	3	5	6	Tercera iteración
I	4	1	2	4	4	

Descripción del Algoritmo de Bellman-Ford:

- Objetivo: encontrar un CMV desde el vértice i al resto de vértices.
    - 1º Tomamos $V' = V \setminus \{i\}$, y definimos $d(v) = \infty$, $I(v) = \emptyset$ para todo $v \notin V'$. Sea $k = 1$.
    - 2º Para todo vértice $(u, v) \in A$, si $d(v) > d(u) + p_{u,v}$ actualizo:
        - $d(v) = \min \{d(v), d(u) + p_{u,v}\}$, $I(v) = u$.
    - 3º Si $k = n-1$, parar. En otro caso, tomar $k = k+1$ e iterar el paso 2º.



	1	3	4	5	6	
I	0	1	-2	0	1	Etapa final
d	4	1	2	4	4	

Cadena de menor valor:

- Partimos de una red no orientada R = (V, A, p) conexa. Dados dos vértices i y j, podemos considerar las cadenas que van desde el vértice i al vértice j.
- Dado una cadena

$$i \xrightarrow{p_{i,i_1}} i_1 \xrightarrow{p_{i_1,i_2}} i_2 \xrightarrow{p_{i_2,i_3}} \dots \xrightarrow{p_{i_{k-1},i_k}} i_k \xrightarrow{p_{i_k,j}} j$$

el valor de la cadena es la suma de los pesos de las aristas que la forman:

$$p_{i,i1} + p_{i1,i2} + \dots + p_{ik-1,ik} + p_{ik,j}$$

La Cadena de Valor Mínimo es la que minimiza el valor de la cadena

- Existencia de la cadena de valor mínimo:
    - La Cadena de Menor Valor existirá si no hay aristas de valor negativo
    - Como todos los pesos son no negativos, se puede utilizar el algoritmo de Dijkstra. Para ello, cada arista i→j se dobla en dos arcos i→j y j→i.

4. REDES DE FLUJO

Red de flujo:

Consideramos una red orientada R = (V, A, p) cumpliendo las siguientes propiedades:

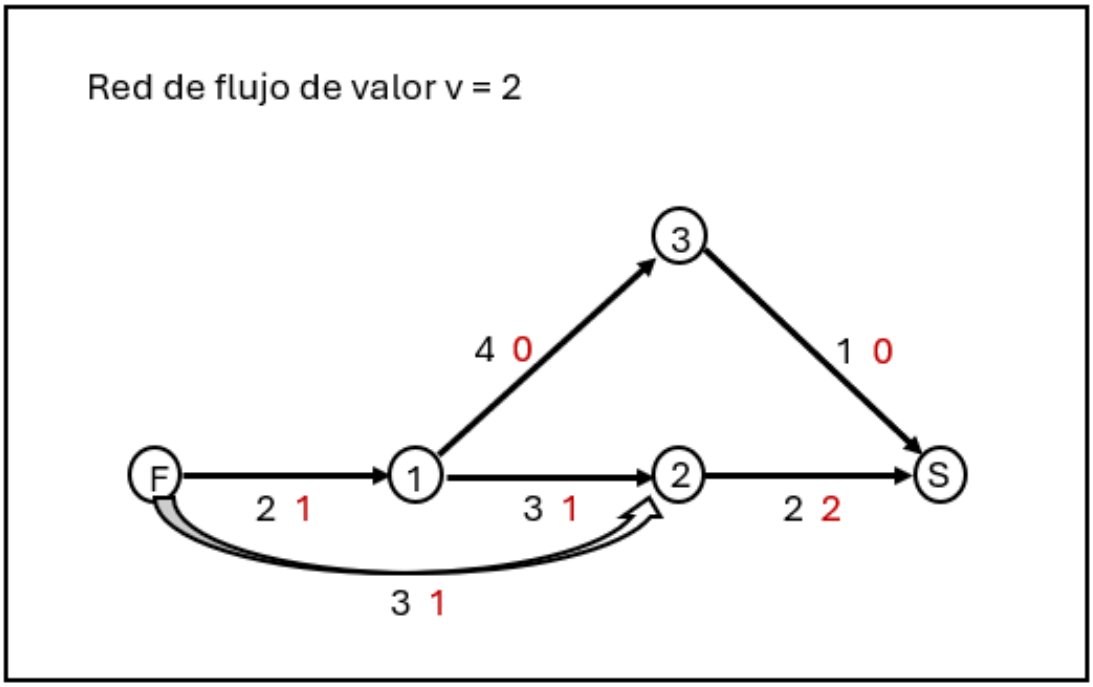
- G = (V, A) es un grafo conexo.
- Los pesos son no-negativos y reciben el nombre de capacidad del arco.
- Hay dos vértices llamados fuente (F) y salida (S) de manera que F no tiene antecesores y S no tiene sucesores.

Flujo:

Partimos de una red de flujo R = (V, A, p). El flujo de valor v es una colección de números que a cada arco (i, j) le asigna un valor fij, cumpliendo:

- Acotación de flujo: 0 <= fij <= pij.
- Conservación del flujo: para todo i ∈ V,

$$\sum_{j\in\Gamma(i)} f_{ij} - \sum_{k\in\Gamma^-(i)} f_{ki} = \begin{cases} v, & \text{si } i = F. \\ -v, & \text{si } i = S. \\ 0 & \text{en otro caso.} \end{cases}$$



Corte:

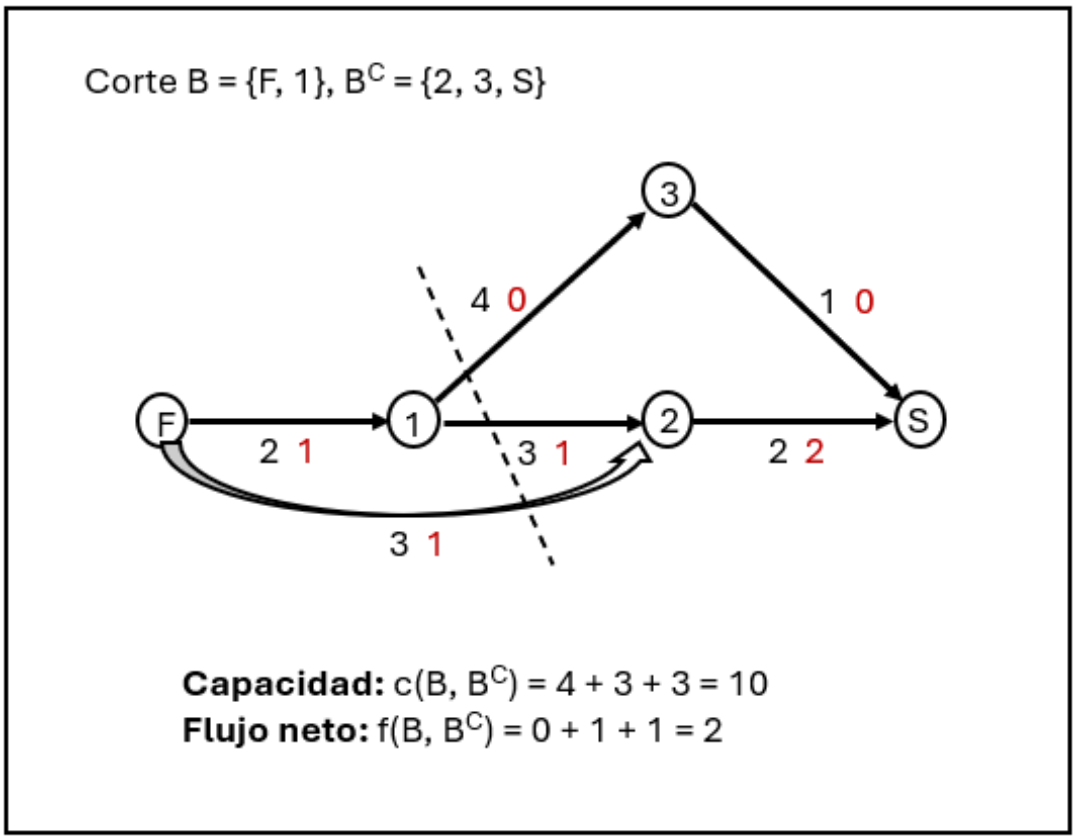
Dada una red de flujo R = (V, A, p) y una participación B, B^C de V, el corte de la red de flujo se denota por (B, B^C) y es el conjunto de arcos de A con origen en B y final B^C.

- La capacidad de corte (B, B^C) es el valor:

$$c(B, B^c) = \sum_{ij \in (B, B^c)} p_{ij}.$$

- El flujo neto del corte (B, B^C) es el valor:

$$f(B, B^c) = \sum_{i \in B, j \in B^c \cap \Gamma_i} f_{ij} - \sum_{i \in B, k \in B^c \cap \Gamma_i^-} f_{ki}.$$

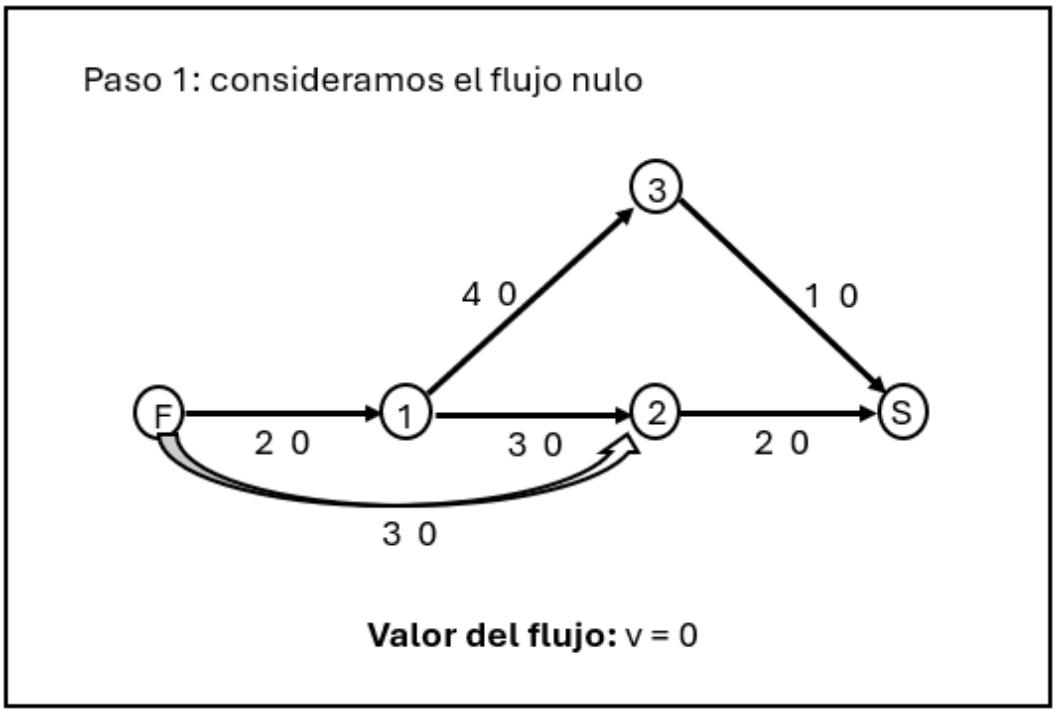


Relación flujo-corte:

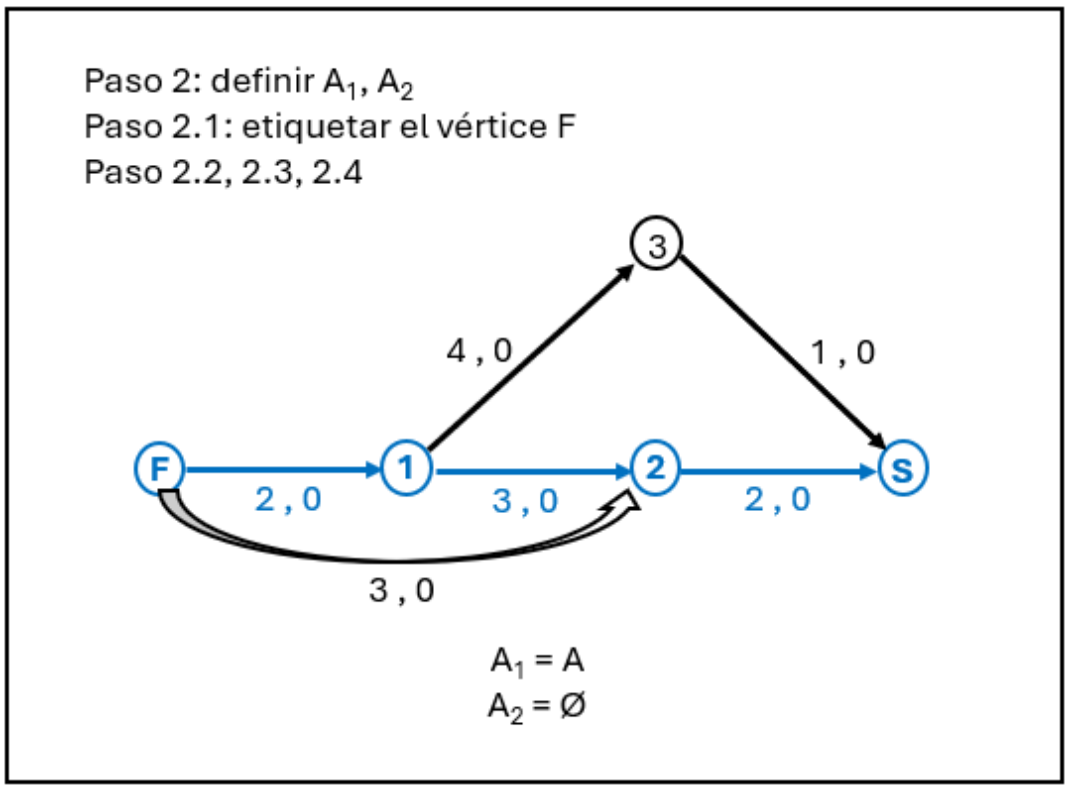
- Teorema: Sea R = (V, A, p) una red de flujo, f un flujo y (B, B^C) un corte. Entonces:
  - El valor del flujo está acotado por la capacidad del corte.
  - El valor del flujo coincide con el flujo neto.
- Objetivo: maximizar el flujo. Sea R = (V, A, p) una red de flujo
  - Flujo máximo: es un flujo f^* de manera que cualquier otro flujo f satisface $v_f <= v_{f^*}$.
  - Objetivo: encontrar el flujo máximo en R = (V, A, p).
  - Corte mínimo: es un corte (B, B^C) que para otro corte (C, C^C) cumple c (B, B^C) <= c (C, C^C).
  - Se cumple que el valor del flujo máximo es igual a la capacidad del corte mínimo.

Algoritmos:

- Ford-Fulkerson: Algoritmo clásico de etiquetado. Se trata de buscar caminos de flujo aumentable:
    - 1º Considera un flujo f en la red R = (V, A, p) (por ejemplo, f=0).

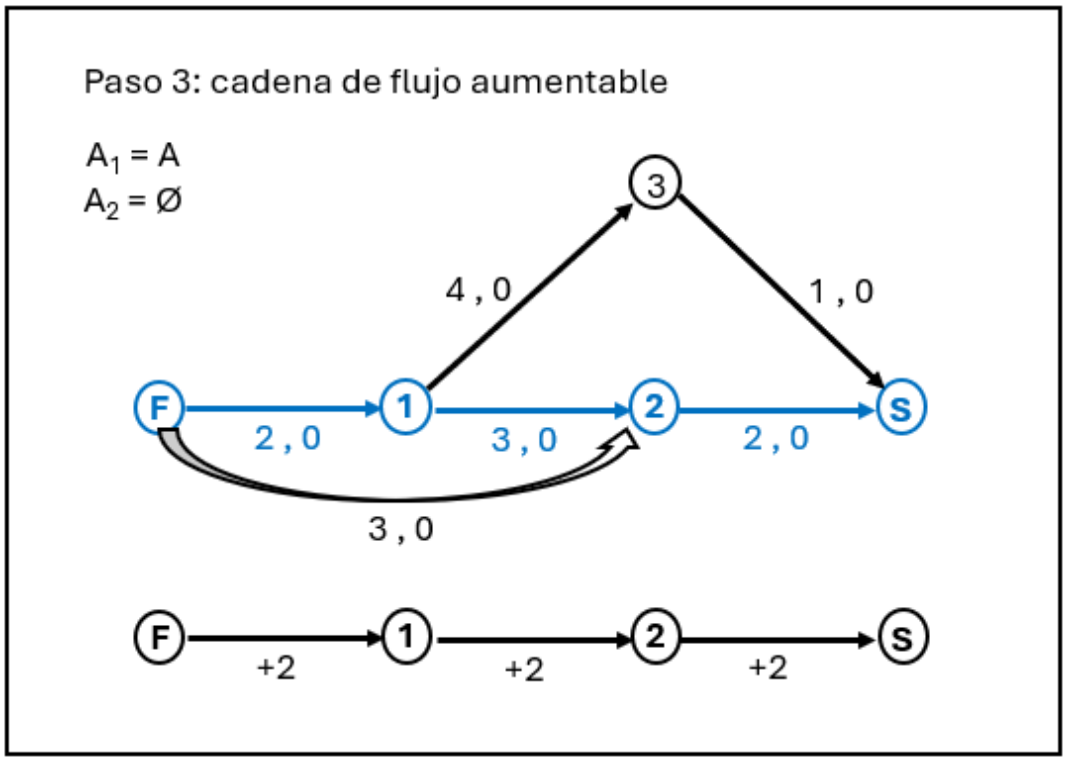


    - 2º Denotar por A_1 el conjunto de arcos no saturados ($p_{ij} > f_{ij}$) y por A_2 el conjunto de arcos con flujo positivo ($f_{ij} > 0$).
        - Etiquetado:
            - 1. Se le asigna la etiqueta + al vértice F.
            - 2. Si i está etiquetado, j no está etiquetado y $(i, j) \in A_1$, entonces etiqueto i y (i, j). (i, j) es un arco de salida.
            - 3. Si j no está etiquetado, i está etiquetado y (j, i) A_2, entonces etiquetar j y (j, i). (j, i) es un arco de entrada.
            - 4. Continuar el etiquetado hasta llegar a S o hasta que no sea posible etiquetar más arcos/vértices.



- 3º Si S no se ha etiquetado, el flujo es máximo. Si S se ha etiquetado, buscar una cadena formada por arcos etiquetados desde F hasta S. Definir:
  - k1 = min {fij | (i, j) ∈ C ⋂ A2}
  - k2 = min {pij - fij | (i, j) ∈ A1 ⋂ C}
  - k = min {k1, k2}

  Aumentar en k unidades el flujo de los arcos de C ⋂ A1 y disminuir en k unidades el flujo de los arcos de C ⋂ A2. Volver al paso 2º.



El fin del algoritmo llega cuando no se puede etiquetar más, que es cuando se alcanza el flujo máximo.

- Boykov-Kolmogorov: Algoritmo novedoso (2004) que mejora algoritmos previos tanto para el problema del flujo máximo como para el flujo a coste mínimo. Es el algoritmo utilizado por Matlab.

5. ADMINISTRACIÓN DE PROYECTOS

Redes de actividad:

- Un proyecto puede estar formado por muchas actividades.
- Cada actividad tiene una duración estimada.
- Algunas actividades no pueden comenzarse hasta que se hayan finalizado otras previas.
- Objetivos:
  - Buscar representaciones de las redes de actividad.
  - Estudiar la duración del proyecto.
  - Calcular la holgura de cada actividad.

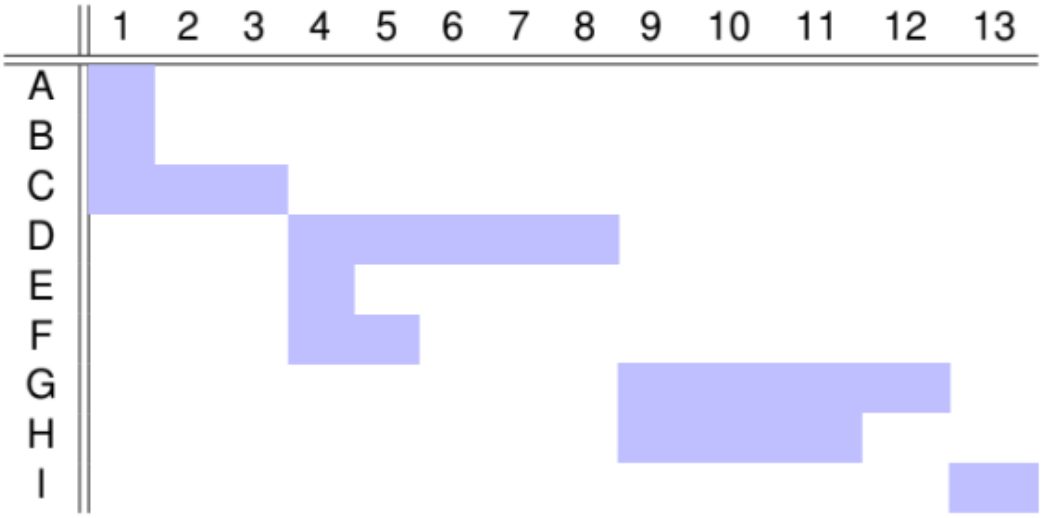
Ejemplo:

- Objetivo: montar una línea de tren que atraviese una montaña mediante un túnel.
- Pasos y duración:

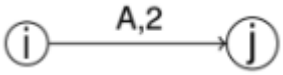
	Actividad	Predecesores	Duración (meses)
A	Contratar personal	-	1
B	Formar personal	-	1
C	Estudios previos y planificación	-	3
D	Tunelar	A, B, C	5
E	Comprar material para las vías	C	1
F	Comprar material electrónico	C	2
G	Montar las vías	D, E	4
H	Montar los sistemas electrónicos	D, F	3
I	Comprobar el funcionamiento	G, H	1

Existen dos tipos de representaciones:

- Gráfico de Gantt

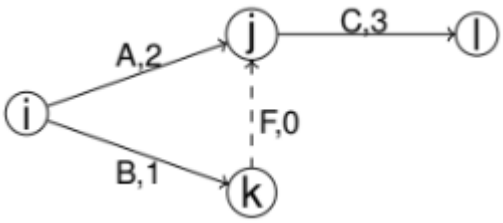


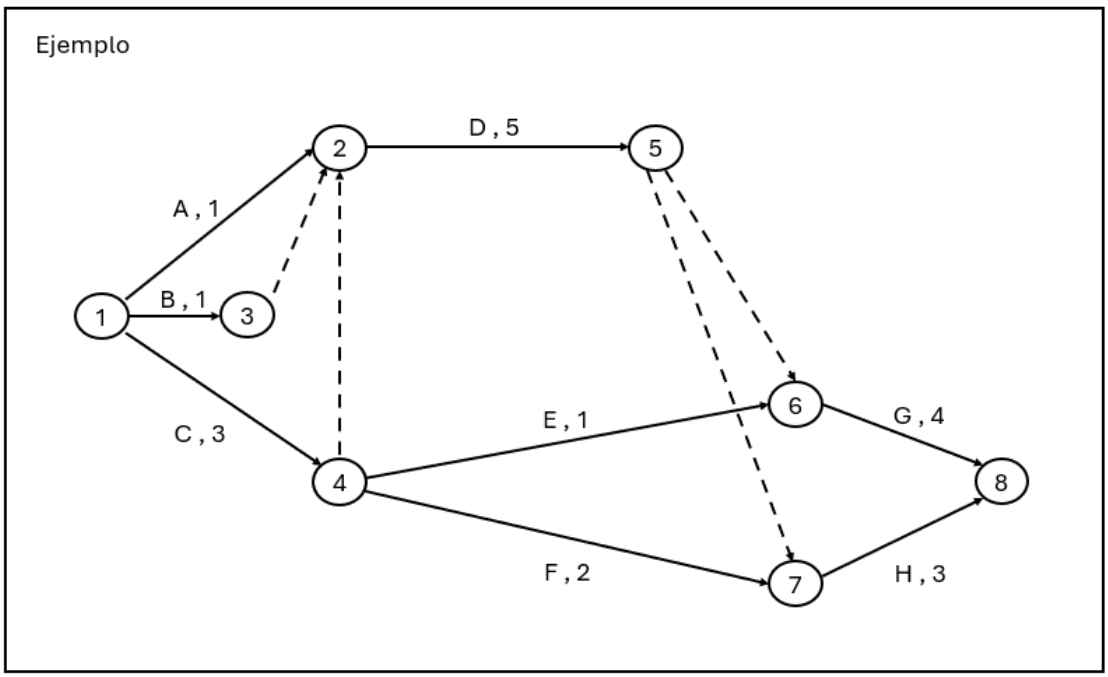
- Grafos orientados
    - Reglas para representar redes de actividad mediante grafos orientados:
        - 1º Cada arco (i,j) representa una actividad, los vértices inicial i, y final, j, representan el inicio y final de la actividad (i < j). Al arco se le añade un peso que es la duración de esa actividad.



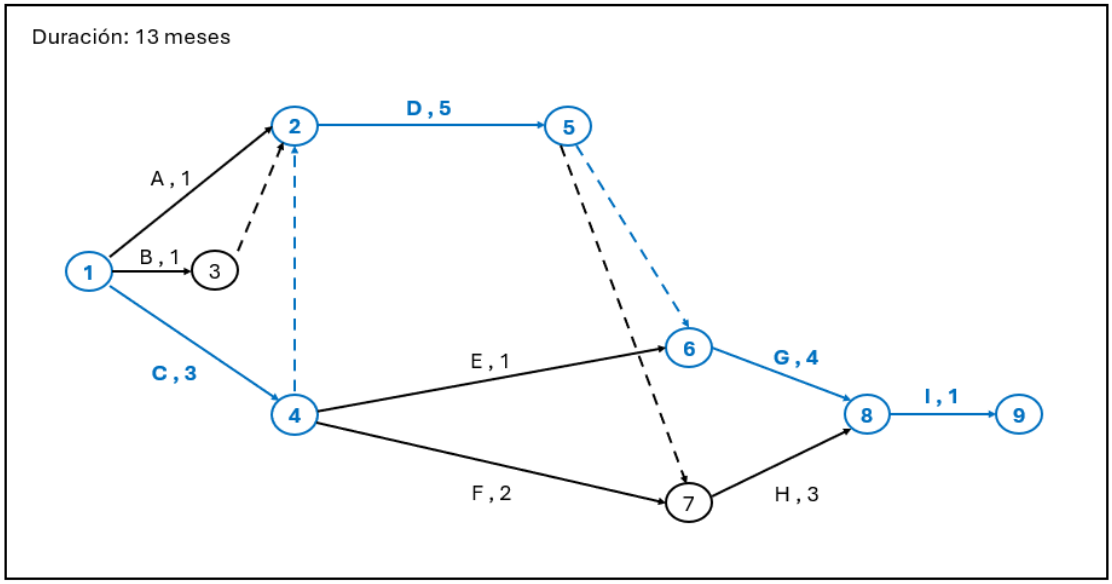
        - 2º Si A precede a B, A = (i,j) y B = (j,k), donde i < j < k.

        - 3º Dos actividades distintas no pueden tener los mismos vértices inicial y final. En tal caso hay que crear arcos ficticios con duración nula. Si A y B preceden a C:



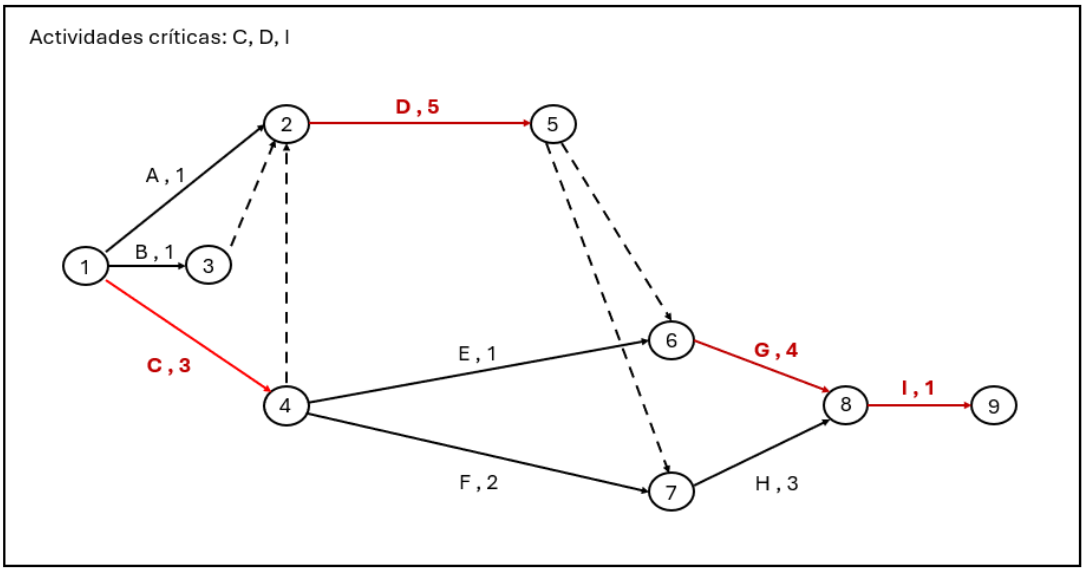


- Objetivo:
  - Duración del proyecto: es la duración del camino más largo, es decir, el camino de mayor valor.



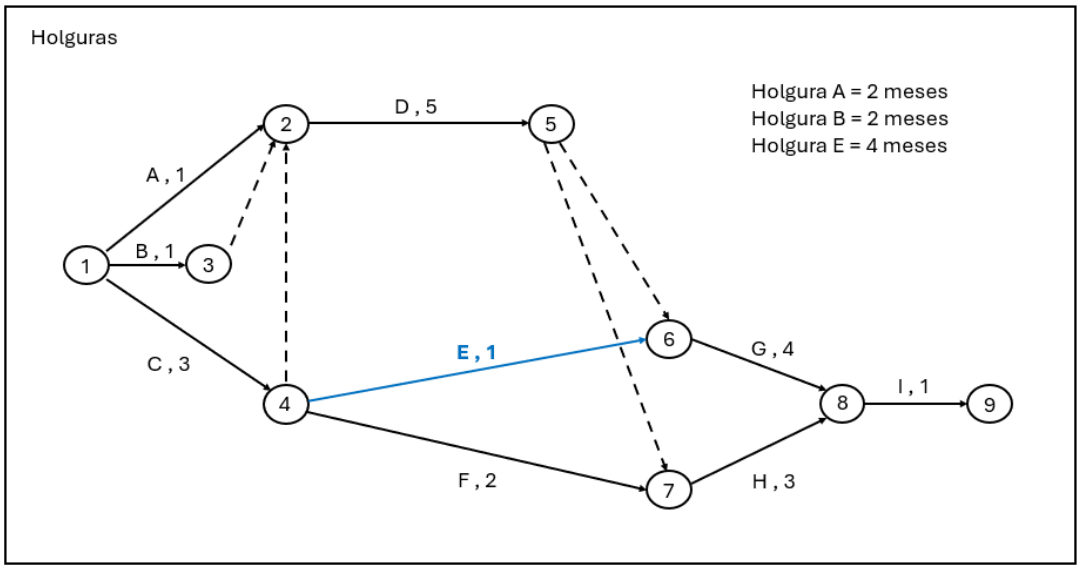
El camino de mayor valor (camino crítico) será {1, 4, 2, 5, 6, 8, 9} que tiene una duración de trece meses.

- Actividades críticas: actividades que no se pueden retrasar; su retraso supondría un retraso en tiempo total de realización del proyecto.



Las actividades críticas serán las correspondientes al camino crítico, es decir, las actividades C, H, G y I.

- Holgura de cada actividad: cuánto se puede retrasar cada actividad para que el proyecto no se retrase.



Las actividades críticas tendrán una holgura nula, ya que si se retrasan el proyecto terminará más tarde. Las holguras de las demás actividades se calculan en torno a la duración de las actividades críticas.

Algoritmos:

- Camino de mayor valor: Solamente da la duración del proyecto, no las actividades críticas ni las holguras.
- Critical path method (CPM): Se utiliza la duración del proyecto así como las holguras y las actividades críticas.

  Problemas: puede no tener sentido fijar la duración exacta de cada actividad.
- Program Evaluation and Review Technique (PERT): Se utiliza la duración del proyecto así como las holguras y las actividades críticas. A cada actividad se le asigna una distribución normal modelando la duración de la actividad.

  Problema: se supone que los tiempos de duración de cada actividad son independientes, y puede no ser realista.

Definiciones básicas:

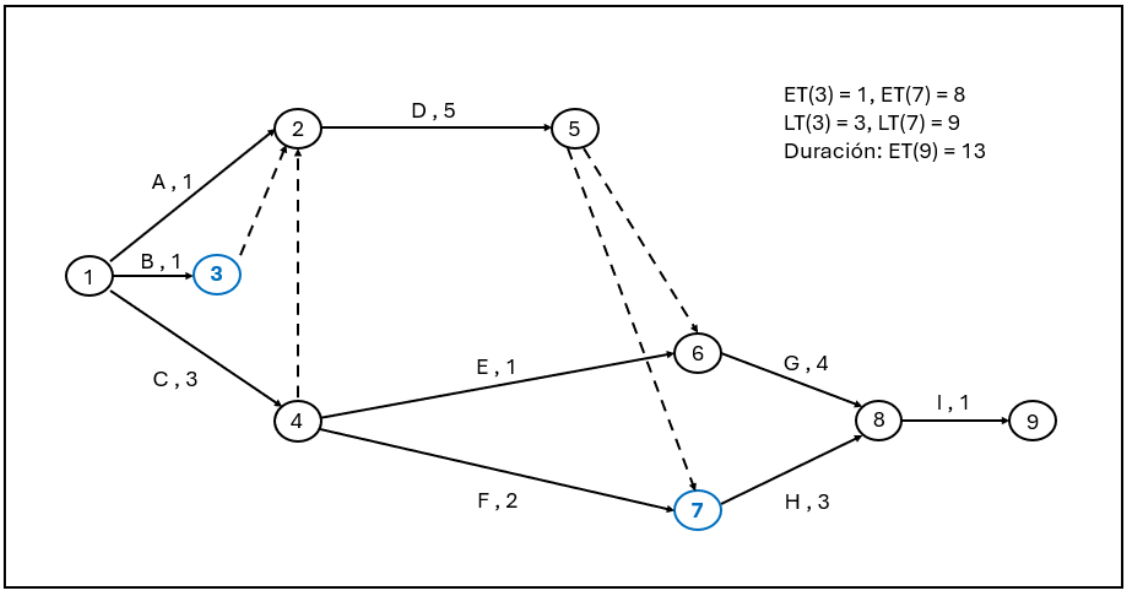
- Early event time (ET): del vértice i es el instante más temprano en el que el vértice i puede ocurrir.

  Cálculo del ET:

  - 1º Para cada vértice i, calcular sus predecesores.
  - 2º Para cada predecesor j de i, sumar a su ET(j) la duración de la actividad (j, i).
  - 3º ET(i) será el máximo de los valores calculados en el paso 2.
- Late event time (LT): del vértice i es el instante más tardío en el que el vértice i puede ocurrir sin retrasar la duración total del proyecto.

  Cálculo del LT:

  - 1º Para cada vértice i, calcular sus sucesores.
  - 2º Para cada sucesor j de i, sumar a su LT(j) la duración de la actividad (i, j).
  - 3º LT(i) será el mínimo de los valores calculados en el paso 2.
- Duración: es el ET o LT del último vértice, que siempre es el mismo en el último vértice.



- ET (3) = max {ET(1) + 3, ET(2) + 2} = 1.
- Para llegar al vértice 7, se necesita que las actividades predecesoras (D, F) se hayan realizado. El tiempo mínimo en realizar estas actividades, y por tanto de empezar la actividad H es de ocho meses (ET (7) = 8).
- LT (3) = min {LT (5) - 5} = 3.
- Para alcanzar el vértice 8 se necesita que la actividad G y H se hayan realizado. Como ambas actividades empiezan al mismo tiempo (tras finalizar la actividad D) y la G tarda un mes más que la H, se puede retrasar un mes la llegada al vértice 7 sin que esto suponga un retraso para el tiempo total del proyecto (LT (7) = 9).
- La duración se obtiene calculando el ET o el LT del último vértice (Para este caso, ET (9) = LT (9) = 13).

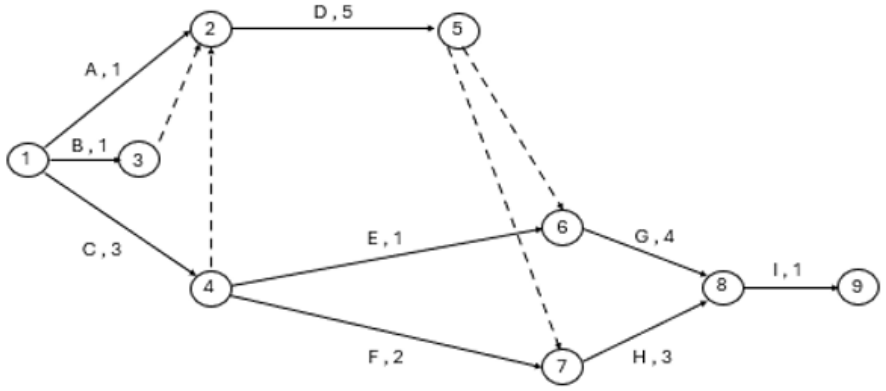
- Holgura total (HT): de la actividad (i, j) es el tiempo que se puede retrasar el inicio de la actividad (i, j) con respecto a ET(i) sin retrasar la duración del proyecto:

$$HT\ (i, j) = LT(j) - ET(i) - p_{ij}$$

- Holgura libre (HL): de la actividad (i, j) es el tiempo que se puede retrasar el inicio de la actividad (i, j) con respecto a ET(i) sin retrasar el inicio de las actividades posteriores:

$$HL\ (i, j) = ET(j) - ET(i) - p_{ij}$$

- Actividades críticas: son aquellas actividades con holgura total nula.



	1	2	3	4	5	6	7	8	9
ET	0	3	1	3	8	8	8	12	13
LT	0	3	3	3	8	8	9	12	13

	A	B	C	D	E	F	G	H	I
HT	2	2	0	0	4	4	0	1	0
HL	2	2	0	0	4	3	0	1	0

PERT:

- Problema del CPM: parte de la hipótesis de que la duración de cada actividad es conocida y determinista.
- PERT: asume que la duración de cada actividad es aleatoria y que los tiempos de las actividades son independientes.
    - a: duración mínima estimada
    - m: duración más probable
    - b: duración máxima estimada

  Se define la variable aleatoria T_A que indica la duración de la actividad

A de manera que:

$$E(T_A) = \frac{a + 4m + b}{6}, \qquad Var(T_A) = \frac{(b-a)^2}{36}.$$

Para un camino crítico, se define la duración total (que también es una variable aleatoria) como:

$$CP = \sum_{(i,j)\in CP} T_{ij} \longrightarrow \mathcal{N}$$

Suponiendo que las duraciones de las actividades son independientes:

$$CP = \sum_{(i,j)\in CP} T_{ij} \longrightarrow \mathcal{N}(\mu, \sigma)$$

donde:

- $\mu = \Sigma_{(i,\,j)\,\in CP} E\,(T_A)$
- $\sigma^2 = \Sigma_{(i,\,j)\,\in CP} \mathrm{Var}\,(T_A)$

6. TEORÍA DE GRAFOS CON MATLAB

En este capítulo vamos a explicar cuestiones introductorias sobre grafos con Matlab. Para ello comenzaremos explicando cómo se pueden definir grafos y como obtener información sobre sus elementos, vértices, aristas. También nos centraremos en este capítulo en el problema de la conexión de grafos.

6.1. Definición de un grafo de forma matricial

Se comienza definiendo una matriz $A_{n\times n}$ donde n es el número de vértices. Esta matriz se llama de adyacencia y se define como

$$A(i,j) = \begin{cases} 1 \; si \; ij \; es \; arista \\ 0 \; si \; no. \end{cases}$$

Si por el contrario buscamos un grafo ponderado, es decir, cuyas aristas tienen un peso definiremos

$$A(i,j) = \begin{cases} p_{ij} \; si \; ij \; es \; arista \\ 0 \; si \; no. \end{cases}$$

donde p_{ij} es el peso de la arista ij.

Una segunda cuestión es la orientación del grafo.

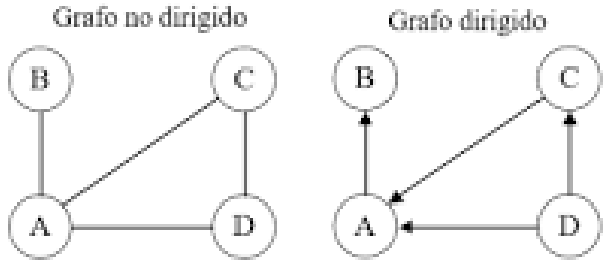
Una vez definida la matriz de adyacencia con pesos, debemos tener en cuenta si el grafo es dirigido o no dirigido. En el caso de grafos dirigidos, utilizaremos la instrucción:

>>G=digraph(A)

Y para el no dirigido

>>G=graph(A).

Esta última instrucción es válida sobre matrices simétricas como es el caso.



Las matrices asociadas son

$$A(no\ dirigido) = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix};$$

$$A(dirigido) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix};$$

Introducimos las matrices en Matlab con las siguientes instrucciones:

>>A1= [0 1 1 1;1 0 0 0;1 0 0 1;1 0 1 0];

>>A2= [0 1 0 0;0 0 0 0 ;1 0 0 0;1 0 1 0];

Definimos los grafos asociados a dichas matrices. La simétrica es no dirigida y la no simétrica dirigida.

>> G=graph(A1)

G = graph with properties:

Edges: [4×2 table]

Nodes: [4×0 table]

>> G=digraph(A2)

G = digraph with properties:

Edges: [4×2 table]

Nodes: [4×0 table]

Como vemos, la salida nos informa de que el grafo orientado G tiene 4 ejes y 4 nodos y que el no orientado también, pero en este último las aristas ac y ca son la misma.

Podemos darles un nombre a los nodos.

>> Nodos= {‘Nodo A’, ‘Nodo B’, ‘Nodo C’,’Nodo D’};

>> G=digraph(A2, Nodos)

.

Una vez definido el grafo, podemos solicitar por pantalla la información de vértices, aristas o pesos con las instrucciones:

>> G. Nodes

>> G. Edges

>> G. Edges. Weight

EndNodes	Weight
{‘Nodo A’} {‘Nodo B’}	1

{‘Nodo C’} {‘Nodo A’}	1
{‘Nodo D’} {‘Nodo A’}	1
{‘Nodo D’} {‘Nodo C'}	1

Nos indica que las aristas dirigidas son nodo A a nodo B, nodo C a nodo A y nodo D al A y al C. Todas con peso 1.

6.2. Definir un grafo a través de los arcos

a. Grafos orientados: Se definen dos vectores con la misma dimensión, s y t. donde (s(i), t(i)) indican los puntos inicial y final, del arco i-ésimo. La instrucción es:

   >> G=digraph (s, t, p) donde p es un vector de pesos opcional.

b. Grafos no orientados: El mecanismo es el mismo que en el caso anterior y la instrucción es

   >> G=graph (s, t, p)

La instrucción G=digraph (s, t, peso, Nodos) es la misma que la de antes pero habiendo dado nombre a los nodos como en los casos matriciales.

6.3. Adición o eliminación de vértices o aristas.

a. Añadir vértices. Si por ejemplo deseamos añadir dos nodos, podemos escribir:

   >> H = addnode (G,2)

   O bien:

   >> H = addnode (G, {‘Nodo 1’,‘Nodo 2’})

Ejemplo: Vamos a definir el grafo G dirigido con arcos

```
>> s = [1 3 4 4];
>> t = [2 1 1 3];
```

>> peso = [1 1 1 1];

Con el orden introducido, estaríamos diciendo que el primer arco es 1 −→ 2 con un peso de 1, el segundo arco 3 −→ 1 con un peso de 1 y así sucesivamente. Ahora haríamos

>> G= digraph (s, t, peso)

G = digraph with properties:

Edges: [4×2 table]

Nodes: [4×0 table]

Si ahora escribimos

>>H = addnode (G, 3) le hemos añadido 3 nodos a G.

La eliminación de nodos, la adición o eliminación de artistas las haremos como en los siguientes ejemplos.

La instrucción

>>H = rmnode (G, 3)

elimina el nodo 3 del grafo.

Para añadir aristas consideremos el siguiente caso:

>>s = [2 2];

>> t = [3 4];

>> peso = [4 5];

>> H = addedge (G, s, t, peso)

Esto añade al grafo G las aristas dirigidas 2→3 y 2→4 dándoles el paso 4 y 5 respectivamente. Si quisiéramos volver a eliminarla escribiríamos

>> s = [2 2];

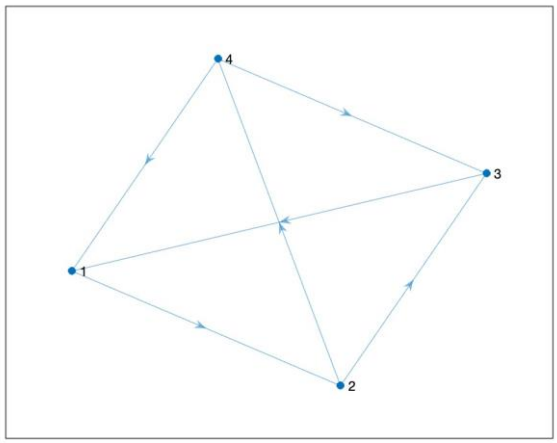
>> t = [3 4];

>> J = rmedge (H, s, t)

6.4. Representación gráfica.

Si el grafo se ha almacenado con el nombre G, utilizaremos la instrucción:

>> plot (H)



Grafo H definido en el punto anterior. Aquí habíamos añadido las aristas 2→3 y 2→4.

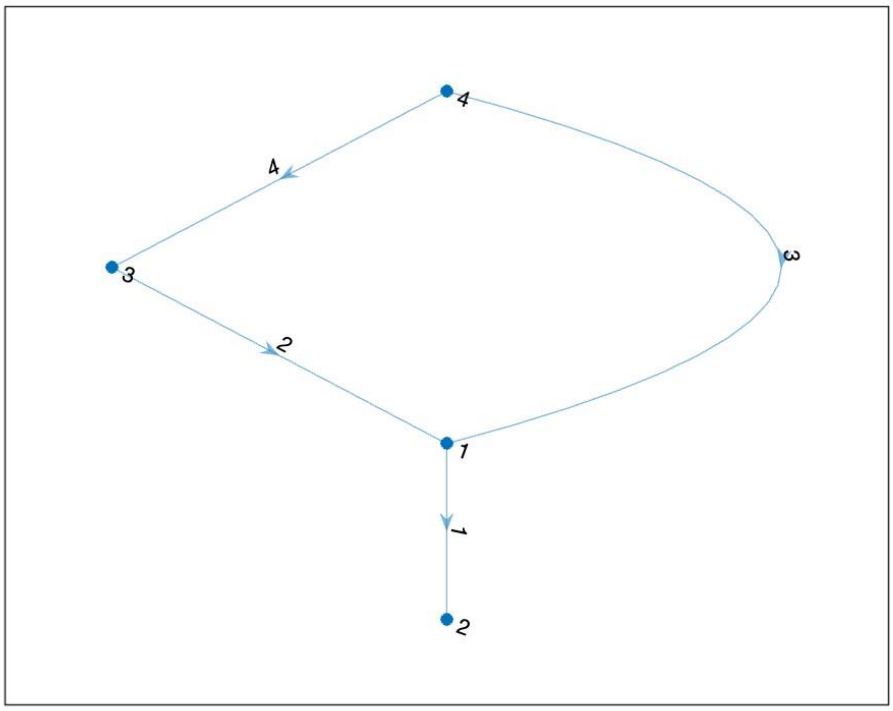
Opciones del dibujo:

a. layout.

   Podemos solicitar que MatLab dibuje los vértices y arcos de manera diferente.

b. labeledge.

   >> labeledge (h, 1: numedges (G), peso) asignará el valor del vector de su peso a cada arco del grafo G. Aquí h almacena el dibujo del grafo como >> h = plot (G).



c. highlight. Con este comando podemos remarcar nodos o arcos de un grafo no orientado, para dibujarlos en distinto color o tamaño. Por ejemplo la instrucción: ↵ >> highlight (h, [1 2], 'NodeColor', 'r') pinta de rojo los vértices 1 y 2 del grafo almacenado en h. ↵

6.5. Información sobre el grafo

Cuando hayamos definido el grafo, podemos solicitar información sobre él.

Grafos orientados

a. Grado de incidencia de entrada de un nodo. Número de aristas que lo contienen como nodo final. Para calcularlo, utilizaremos el comando indegree:

   >> indegree(G)

   ans =

   2 1 1 0

b. Grado de incidencia de salida. Número de aristas que lo contienen como nodo inicial. Para calcularlo, utilizaremos el comando outdegree:

>> outdegree(G)

ans =

1 0 1 2

c. Predecesores de un vértice. Los predecesores de un vértice i son los vértices j de manera que j −→ i es un arco., usaremos la instrucción para su cálculo.

   >> predecessors (G, 1) o bien >> predecessors (G, 'Nodo 1') si los nodos tienen nombre.

   ans =

   3 4

   Es decir aristas 3→1 y 3→4.

d. Sucesores de un vértice. Al igual que antes.

   >> sucessors (G, 1)

   ans =

   2

   Es decir la arista 1→2

e. Determinar si hay ciclos.

   >> isdag (G)

   Resulta 1 si no hay ciclos y 0 si los hay.

6.6. Conexión

a. Conexión en grafos no orientados. Un grafo no orientado es conexo cuando existe una cadena entre cada par de vértices.

b. Conexión débil en grafos orientados. Un grafo orientado es débilmente conexo cuando su grafo no orientado asociado es conexo.

c. Conexión en grafos orientados. Un grafo orientado es conexo cuando entre cada par de vértices i, j existe un camino de i a j y otro camino de j a i.

Cuando un grafo no es conexo se intentan calcular sus componentes conexas que son subgrafos maximales conexos.

a. Si G es un grafo no orientado usamos >> conncomp (G)

   Como resultado obtendremos un vector con tantos valores como vértices, de manera que para el vértice i indica a qué componente conexa pertenece. Si solamente hay una componente conexa, el grafo G será conexo.

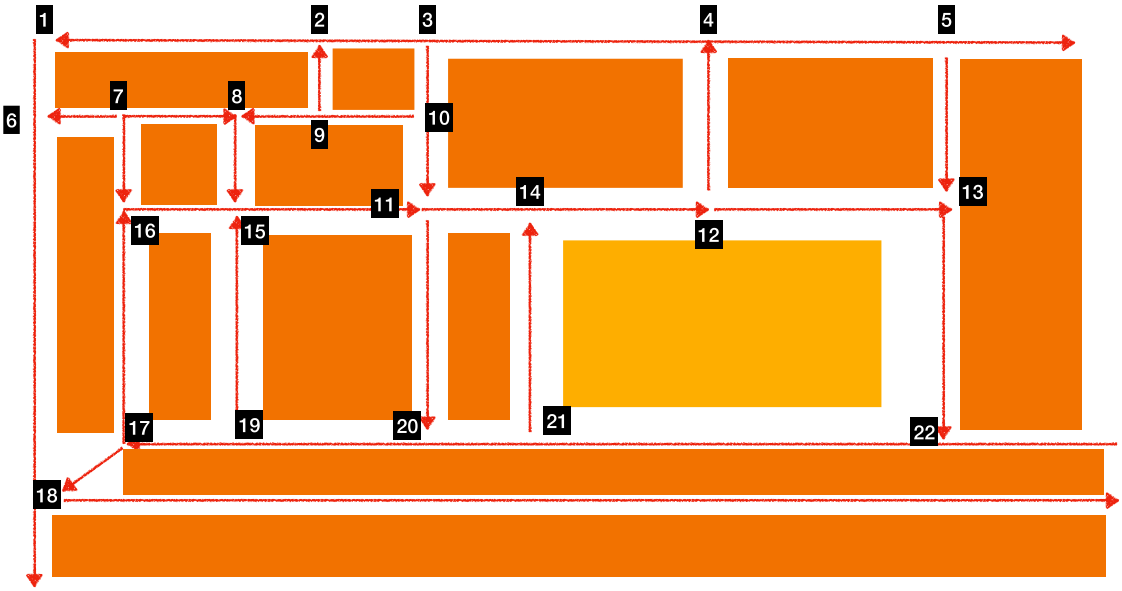
b. Si G es un grafo orientado y queremos estudiar su posible conexión, seguiremos los mismos pasos del apartado anterior.

c. Si G es un grafo orientado y queremos ver si es débilmente conexo, esto es, si su grafo no orientado asociado es conexo, usaremos la siguiente instrucción:

   >> conncomp (G) ans 3 4 2 1. Hay tantas componentes conexas como vértices.

   >> conncomp (G, 'Type', 'weak') ans = 1 1 1 1 que indica que es débilmente conexo.

6.7. Aplicación al barrio de Bami (Sevilla)



Plano del Barrio de Bami, donde la zona amarilla es su plaza central y las naranjas bloques de edificios.

Para ello, vamos a expresar un problema de conectividad de un grafo. Lo primero

que hemos de hacer es expresar nuestro plano como un grafo. Lo que haremos será definir un vértice para cada cruce, y entre dos vértices "consecutivos" definiremos un arco de acuerdo con la dirección del tráfico.

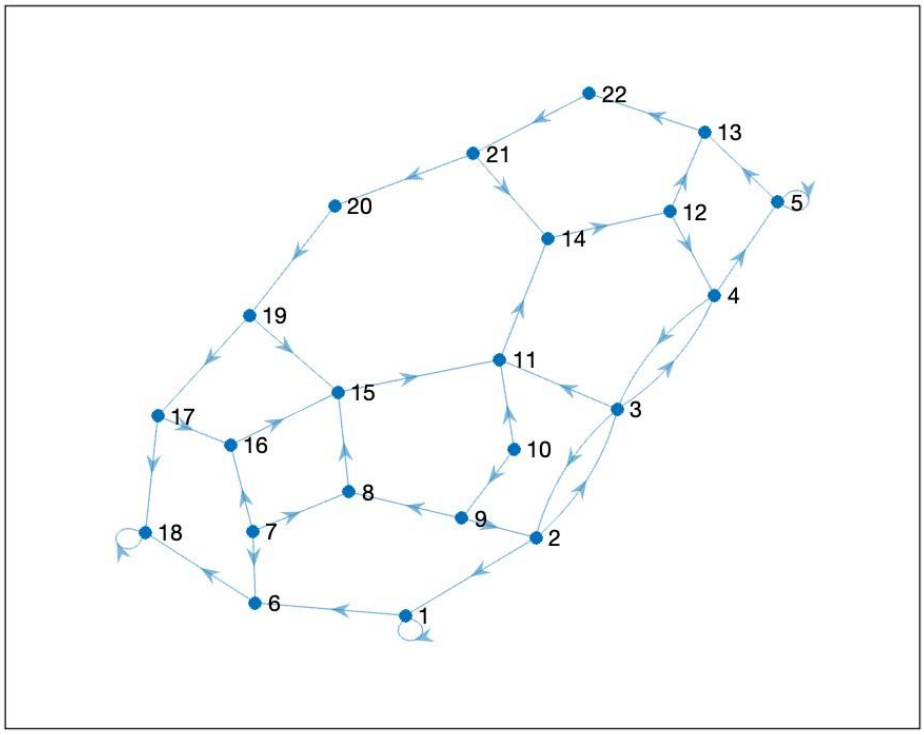
No tenemos pesos luego nos bastaría la matriz de adyacencia. En nuestro caso son 22 nodos. Una vez calculada calculamos nodos y ejes.

G = digraph(A)

G = digraph with properties:

Edges: [36×2 table]

Nodes: [22×0 table]



Tenemos 36 ejes y 22 nodos. Los nodos 1, 18 y 5 son salidas al exterior del barrio.

Nuestro objetivo es estudiar si la planificación urbanística es correcta, es decir, si desde cada punto del barrio es posible acceder a todos los demás cruces tanto a pie como en vehículo. En términos de grafos, este problema se traduce en estudiar la conexión del grafo G. En el primer caso, un peatón a pie puede ir siempre en ambas direcciones, así

que debemos de analizar si el grafo es débilmente conexo (cada arco se vuelve una arista y nos “olvidamos” de las direcciones):

```
>> conncomp (G, 'Type', 'weak')

ans =

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

Como sólo tenemos una componente conexa, efectivamente a pie podemos ir a todos los cruces.

Pasemos a analizar si la planificación en vehículo es también correcta. En este caso sé debemos de tener en cuenta las direcciones, y por lo tanto tendremos que ver si el grafo es conexo:

```
>> conncomp (G)

ans =

  Columns 1 through 13

    6    5    5    5    5    7    3    4    2    1    5    5    5

  Columns 14 through 22

    5    5    5    5    8    5    5    5    5
```

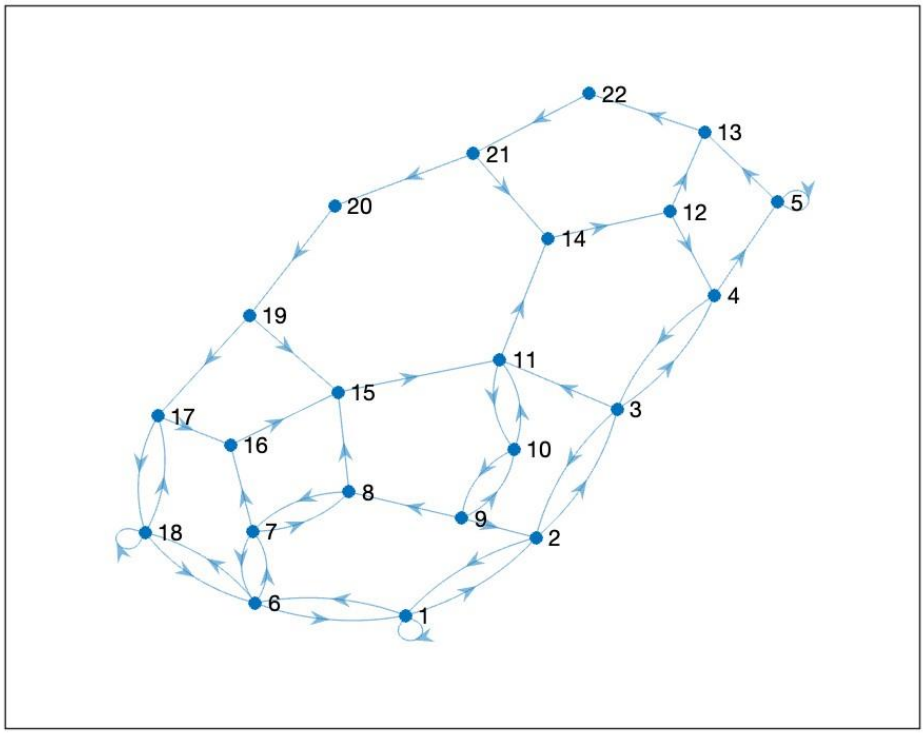
Nos resultan 8 componentes conexas. El nodo 1 a la componente 6, los nodos 2-5, 11-17 y 19-22 a la componente 5, el nodo 6-10 está cada una en su componente y la dieciocho igual. Luego entre los nodos de la componente 5 podemos trasladarnos en coche sin problemas. Para solucionarlo calculamos predecesores y sucesores:

```
>> successors (G, 1)

ans = 8

>> predecessors(G,1)

ans = 2
```

Realmente si conectamos 1 con 6, A (6, 1) = 1 y con el 2, A (1, 2) = 1 acabamos con el problema de salir o entrar desde el nodo 1.

Seguimos con el 6 cuyos predecesores son 1 y 7, tomamos 7 ya que 1 se ha usado. Es decir A (6, 7) = 1 y lo unimos también al sucesor A (18, 6) = 1. Ahora vemos que el 7 no tiene predecesores sino dos sucesores 6 y 8. Escogemos el 8 ya que ya ha salido el

6 y así A (8, 7) = 1. Los sucesores de 9 ya han salido y son el 2 y el 8 y tiene un predecesor que es el 10, así ponemos A (9, 10) = 1. El diez no tiene sucesores pero dos predecesores el 9 y el 11. En este caso escogeríamos el 11 haciendo A (11, 10) = 1. Del 11 al 17 pertenecen a la componente 5 luego pasamos al 18 que se arregla conectando con el 17 A (18, 17) = 1. Rehaciendo la matriz de adyacencia llegamos a



Nueva distribución de direcciones de calles.

Analizamos si hay una sola componente:

>>conncomp(G)

ans =

Columns 1 through 13

1 1 1 1 1 1 1 1 1 1 1 1 1

Columns 14 through 22

1 1 1 1 1 1 1 1 1

Como vemos hay una sola componente luego con las nuevas direcciones podemos

ir de un nodo a cualquier otro nodo del barrio.

7. ÁRBOLES DE UNIÓN DE MENOR VALOR

En esta práctica estudiaremos cómo calcular el árbol de unión de menor valor (AUVM) usando MatLab como venimos haciendo. Conviene tener presente que un árbol es un grafo no orientado conexo y acíclico. Empezaremos a partir de un grafo no orientado G = (V, A) y unos pesos *p* de forma que cada arista *e* tendrá asociado un peso concreto p_e. Lo que buscaremos será encontrar un subgrafo de G que sea un árbol que minimice el valor total de los pesos.

7.1. El comando minspantree

Minspantree es un comando que nos permitirá encontrar un AUVM. Una vez definido un grafo no orientado G, solo tendremos que escribir la siguiente instrucción:

```
>> T = minspantree (G)
```

Es aconsejable guardar el resultado almacenándolo en un valor T, ya que la salida de este comando será un nuevo grafo, subgrafo de G y que contendrá únicamente los arcos que minimizarán el valor total del árbol. Una vez obtenido el grafo T, éste nos proporcionará toda la información que necesitemos como hemos hecho en la práctica anterior. A modo de ejemplo, si queremos saber las aristas que se han incluido en el subgrafo, escribimos:

```
>> T. Edges
```

Si el grafo G lo hemos definido introduciendo las aristas con los vectores s y t, y otro vector para indicar los pesos, podemos representar gráficamente el grafo y su AUVM con las siguientes instrucciones:

```
>> G = graph (s, t, peso)

>> h = plot (G)

>> labeledge (h, 1: numberedges (G), peso)

>> T = minspantree (G)

>> highlight (h, T, 'EdgeColor', 'red')
```

Con la instrucción labeledge introducimos los pesos a las aristas, y con highlight

obtendremos las aristas del árbol en color rojo.

En algunos casos, puede ser útil crear un AUVM a partir de un vértice dado. Para esto, el vértice en cuestión se añadirá como raíz. Si por ejemplo, el vértice escogido es el segundo, la instrucción a utilizar será la siguiente:

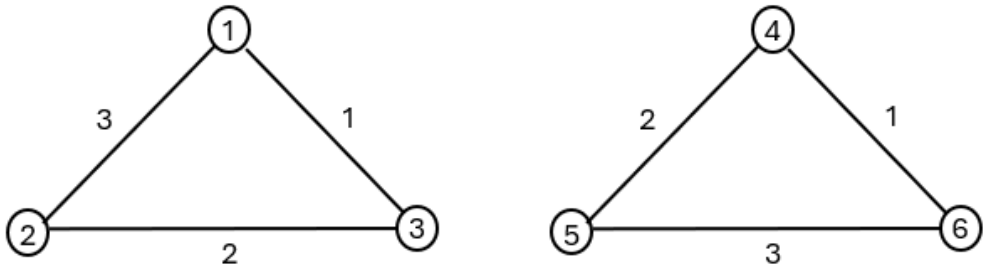
>> T = minspantree (G, ‘Root’, 3)

Si no se indica el vértice inicial, se cogerá como raíz al primer vértice.

Si el grafo introducido no fuera conexo, el comando minspantree únicamente calcularía el árbol asociado a la componente conexa a la que pertenece su raíz. En el caso de que el grafo original no sea conexo, una opción será calcular el bosque de menos valor. En este caso, MatLab nos proporcionará el AUVM para cada una de las componentes conexas. Para obtener el bosque en vez del árbol, usaremos la siguiente instrucción:

>> T = minspantree (G, ‘Type’, ‘forest’)

Si consideramos el siguiente grafo no orientado:



Lo primero será definir el grafo en MatLab con los siguientes comandos:

>> s = [1 1 2 4 4 5]

>> t = [2 3 3 5 6 6]

>> peso = [3 1 2 1 2 3]

>> G = graph (s, t, peso)

G = graph with properties:

Edges: [6x2 table]

Nodes: [6x0 table]

Podemos ver, que el grafo no es conexo, usando:

>> conncomp (G)

ans =

1 1 1 2 2 2

Usando minspantree sin concretar la raíz, tomará por defecto como vértice e1 y obtendremos lo siguiente:

>> T1 = minspantree (G)

T1 = graph with properties:

Edges: [2x2 table]

Nodes: [6x0 table]

>> T1. Edges

ans =

EndNodes		Weight
1	3	1
2	3	2

Ahora, por ejemplo vamos a buscar el AUVM indicando como raíz al vértice número 4 y obtendremos lo siguiente:

>> T2 = minspantree (G, 'Root', 4)

T2 = graph with properties:

Edges: [2x2 table]

Nodes: [6x0 table]

>> T2. Edges

ans =

EndNodes		Weight
4	5	1
4	6	2

En este caso, el árbol solo hace referencia a los vértices 4, 5 y 6, es decir los vértices de la segunda componente conexa, quedando aislados los vértices 1, 2 y 3.

Por lo contrario, si queremos obtener el bosque escribimos:

>> F = minspantree (G, ‘Type’, ‘forest’)

F = graph with properties

Edges: [4x2 table]

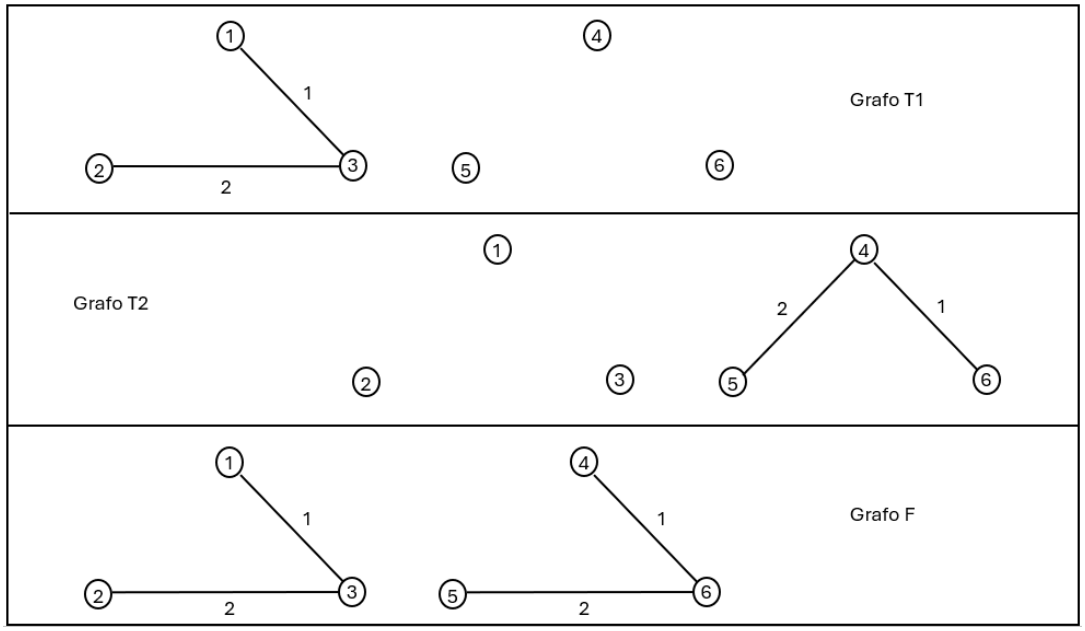
Nodes: [6x0 table]

>> F. Edges

ans =

EndNodes		Weights
1	3	1
2	3	2
4	5	1
4	6	2

En este segundo caso, el programa ha obtenido el AUVM en cada una de las componentes formando el bosque de unión de valor mínimo. Los grafos obtenidos son los siguientes:



También podríamos solicitar un vector que contuviese los predecesores de cada nodo, empezando a partir de la raíz, que tendría valor de 0 al no tener predecesor. En el caso del bosque antes obtenido, los predecesores se obtienen de la siguiente manera:

>> [F, pred] = minspantree (G, ‘Type’, ‘forest’)

F = graph with properties:

Edges: [4x2 table]

Nodes: [6x0 table]

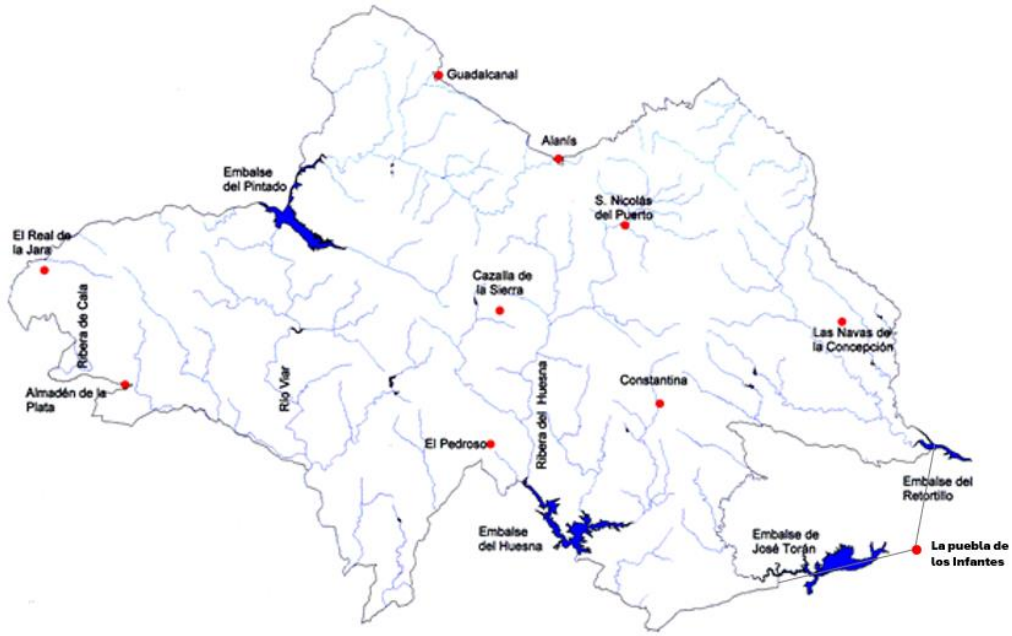
pred =

0 3 1 0 4 4

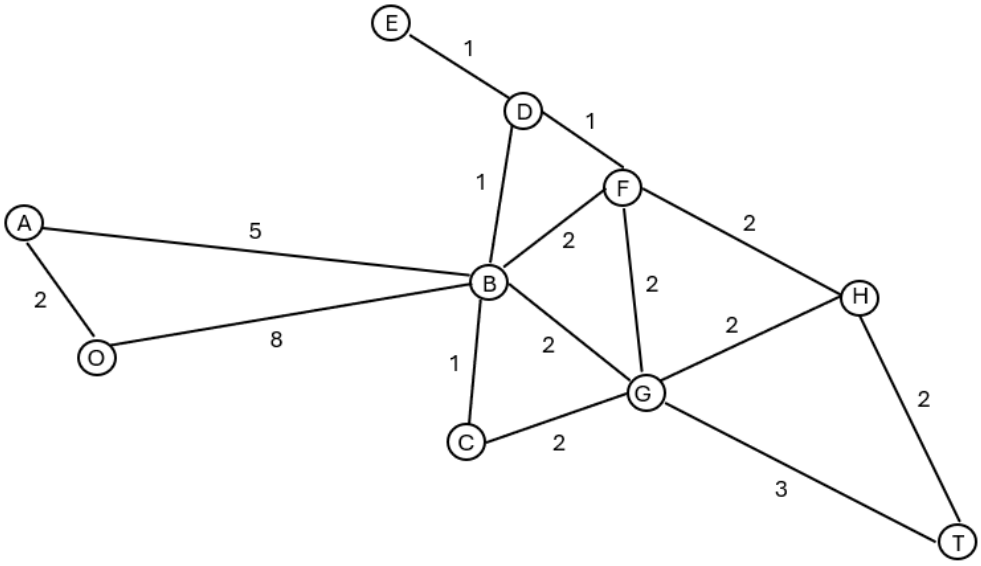
El vector pred nos indica la procedencia de cada vértice. También nos muestra los vértices tomados como raíces, que son los que tiene un cero (en nuestro caso el 1 y el 4). Además, el predecesor del vértice 2 es el 3, el del 3 es el 1 y los vértices 5 y 6 están ambos precedidos por el 4.

7.2. Aplicación a la conexión entre los pueblos de la sierra norte de Sevilla

En la zona norte de la provincia de Sevilla hay diez municipios que son los puntos rojos que se indican en el siguiente mapa:



Los pesos de los arcos se han tomado según la relación de una unidad por cada diez kilómetros. Por otro lado, los nodos son cada uno de los pueblos. El grafo queda de la siguiente manera:



El objetivo es reducir al máximo la longitud de las carreteras, lo que hará minimizar los costes de transportes de las distribuidoras. Lo que buscamos es encontrar árbol de unión de valor mínimo. Empezamos definiendo el grafo en MatLab:

>> s = [1 1 2 3 3 3 3 4 5 5 7 7 8 8 9];

>> t = [2 3 3 4 5 7 8 8 6 7 8 9 9 10 10];

>> peso = [2 8 5 1 1 2 2 2 1 1 2 2 2 3 2];

>> G = graph (s, t, peso, {‘O’ , ‘A’ , ‘B’ , ‘C’ , ‘D’ , ‘E’ , ‘F’ , ‘G’ , ‘H’ , ‘T’})

G =

Graph with properties:

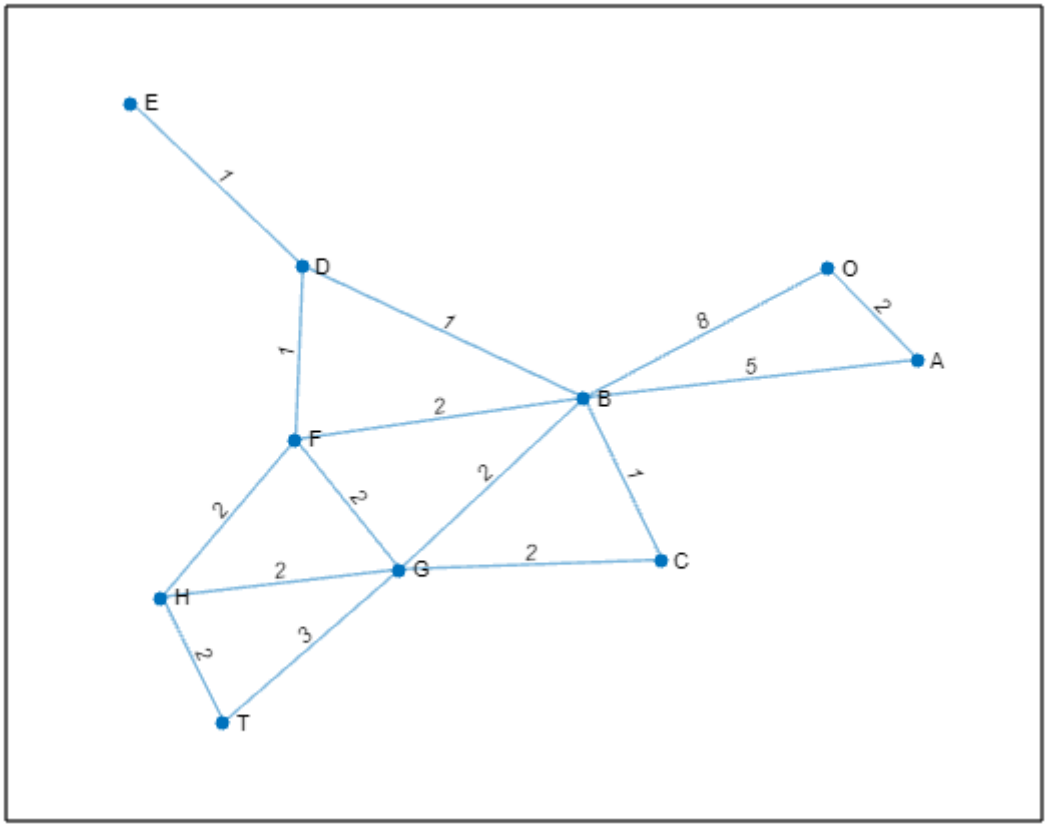
Edges: [15x2 table]

Nodes: [10x1 table]

>> h = plot (G)

>> labeledge (h, 1: numedges (G), peso)

Con estos comandos hemos definido el grafo y obtenido una representación de nuestro grafo:



Ahora buscaremos el AUVM:

>> T = minspantree (G)

T =

Graph with properties:

Edges: [9x2 table]

Nodes: [10x1 table]

>> T. Edges

ans =

EndNodes		Weights
{‘O’}	{‘A’}	2
{‘A’}	{‘B’}	5
{‘B’}	{‘C’}	1
{‘B’}	{‘D’}	1
{‘B’}	{‘G’}	2
{‘D’}	{‘E’}	1

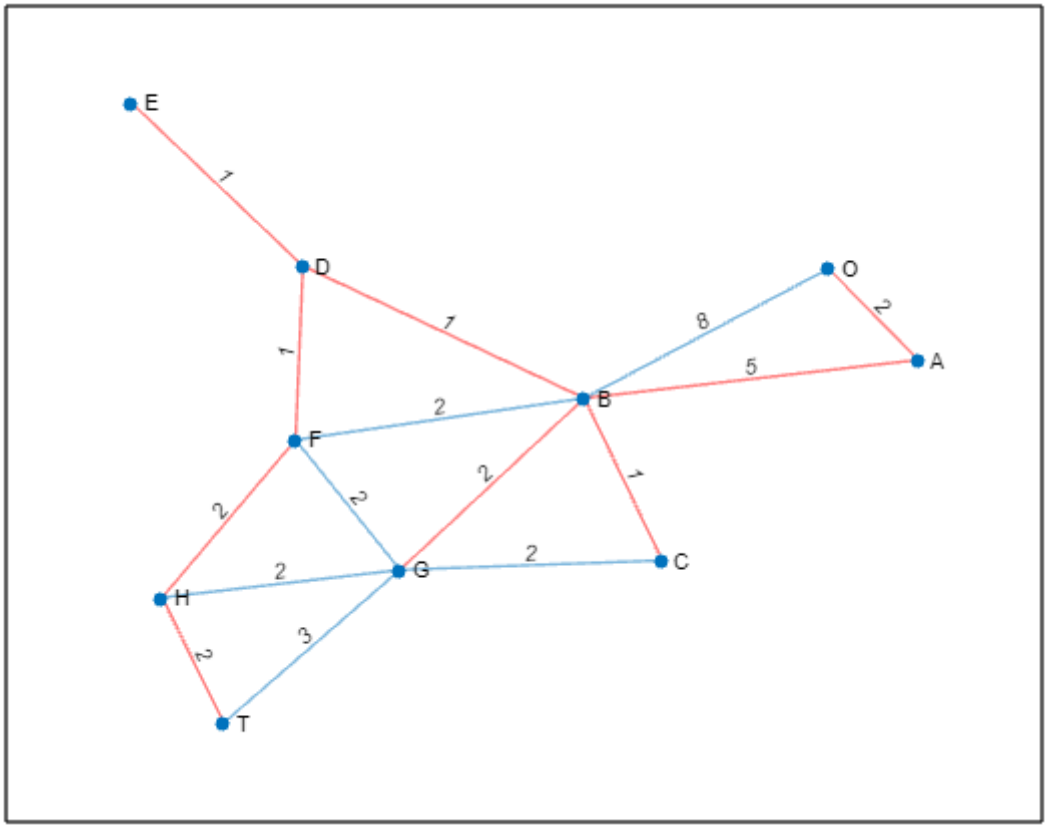
{‘D’} {‘F’} 1

{‘F’} {‘H’} 2

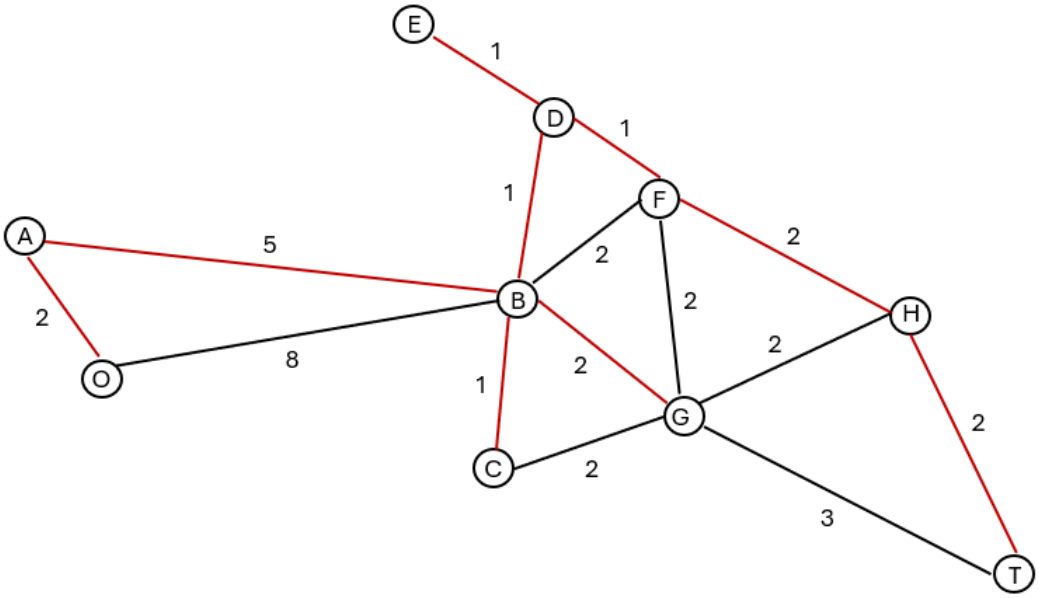
{‘H’} {‘T’} 2

Con el comando T. Edges obtenemos información de las aristas escogidas para formar una parte del AUVM. Mediante otras instrucciones podríamos obtener más información como la matriz de adyacencia (adjacency (T)) pero eso excede de nuestra aplicación. Ahora representaremos gráficamente el árbol escribiendo:

>> highlight (h, T, ‘EdgeColor’, ‘red’)



También podemos verlo usando la representación del grafo original:



8. CAMINOS Y CADENAS DE MENOR VALOR

En las prácticas anteriores hemos visto los primeros pasos para aplicar la Teoría de Grafos en MatLab. Hemos definido grafos, arcos y vértices, así como obtener gráficas y analizar las conexiones. Ahora veremos cómo resolver el problema de la ruta más corta con MatLab. La fórmula del problema de la ruta más corta, como hemos visto anteriormente, comienza con una red R = (V, A, p) conexa. Es decir, un grafo conexo con arcos, vértices y un vector con los pesos, que indica el valor que se le da a cada arco, de forma que cualquier vértice sea alcanzable desde cualquier otro. Lo que buscaremos será, conociendo un par de vértices i y j, descubrir el camino que menor valor tenga para ir desde i a j. Es decir, el objetivo será minimizar el valor del camino. Existen algunos algoritmos con los que podremos resolver este problema, en función de cómo sean los datos que tenemos:

- Algoritmo Dijkstra: cuando todos los pesos sean no negativos
- Algoritmo Floyd: si hay algún peso negativo. Habrá solución si no hay ciclos de valor negativo.

Si el grafo es no orientado el camino de menor valor existirá únicamente si los pesos son no negativos, por lo que se usará el algoritmo Dijkstra.

Ahora veremos cómo encontrar el CMV con MatLab.

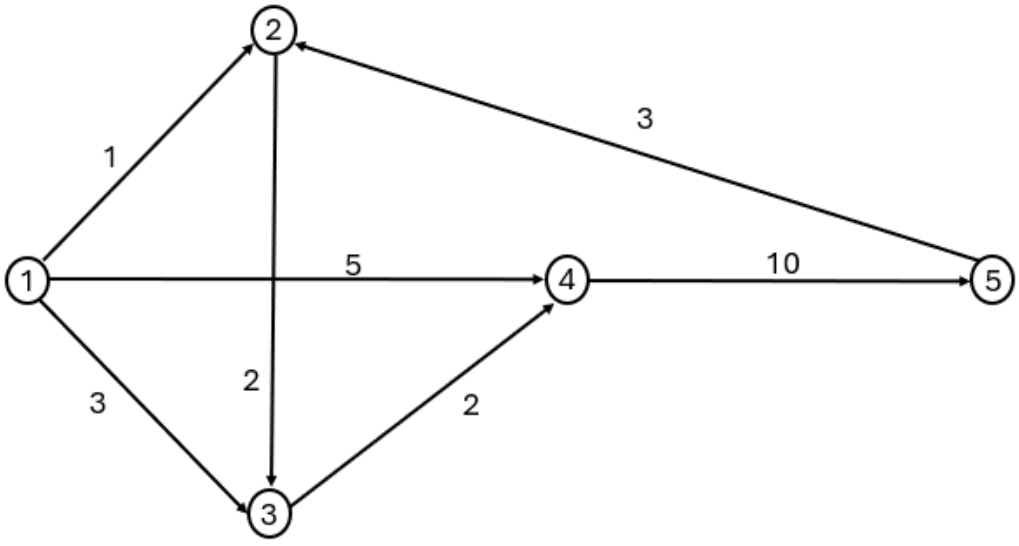
8.1. El comando shortestpath en MatLab

Dado un grafo G definido como venimos haciendo, que tiene asociado un vector de pesos p que asocia un valor a cada arco, el comando con el que el programa nos permitirá resolver el problema de la ruta más corta será shortestpath. Al utilizar esta orden, deberemos añadir el nombre dado al grafo (generalmente G) y los vectores s y t. MatLab nos dará el CMV desde el vértice s(i) hasta el vértice t(i). Escribimos por tanto:

```
>> shortestpath (G, s, t)
```

y obtendremos por pantalla los vértices que definen la ruta de menor valor procurada.

A modo de ejemplo, buscaremos el CMV asociado a este grafo de cinco vértices y siete arcos, desde el primer vértice hasta el último:



Primero definimos el grafo:

```
>> s = [1 1 2 3 3 4 5];

>> t = [2 4 3 1 4 5 2];

>> peso = [1 5 2 3 2 10 3];
```

>> G = digraph (s, t, peso)

G =

digraph with properties:

Edges: [7x2 table]

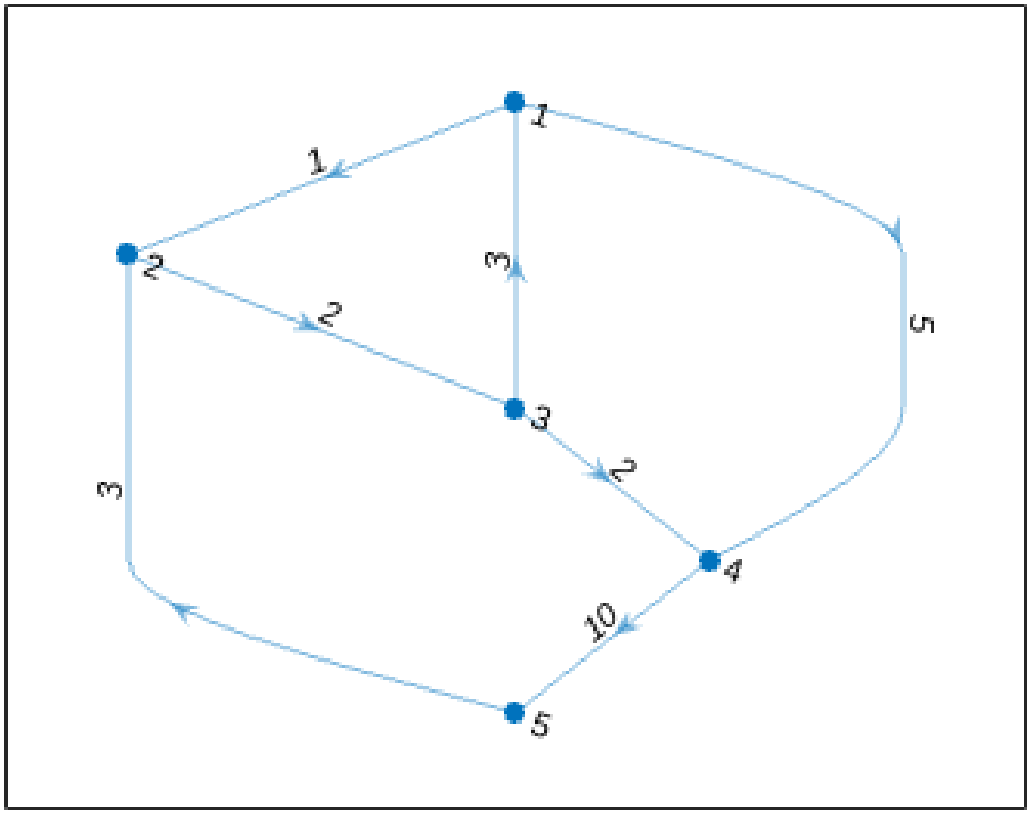
Nodes: [5x0 table]

Para obtener la representación gráfica, como ya hemos visto en las prácticas anteriores utilizamos las siguientes instrucciones:

>> h = plot (G, ‘Layout’, ‘layered’)

>> labeledge (h, 1: numedges (G), peso)

y obtenemos:



y ahora resolvemos el CMV:

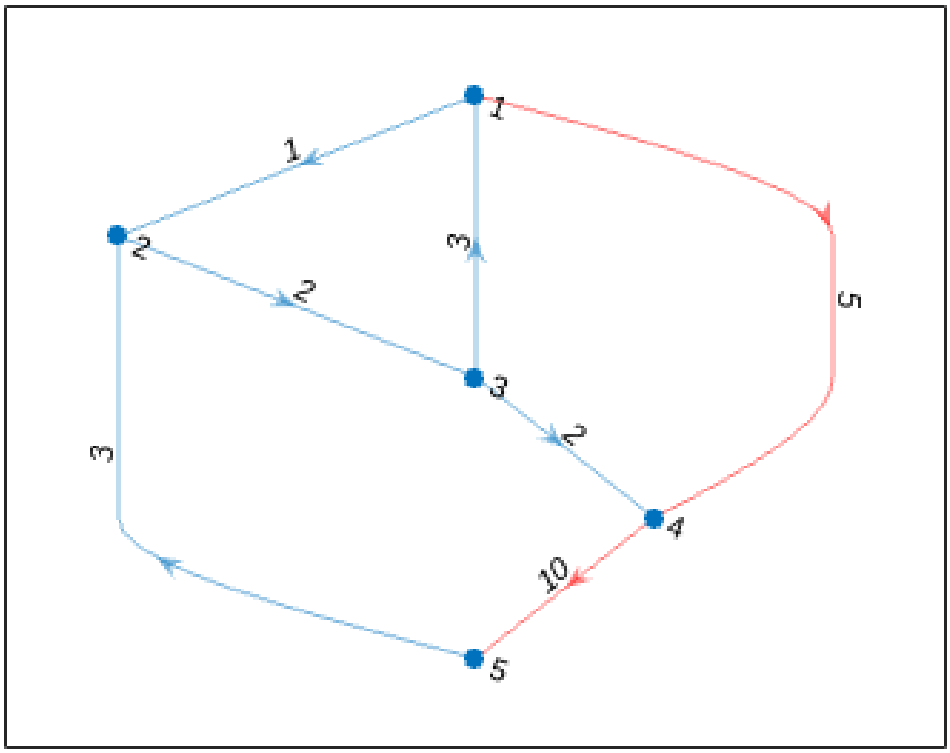
>> CMV = shortestpath (G, 1, 5)

CMV =

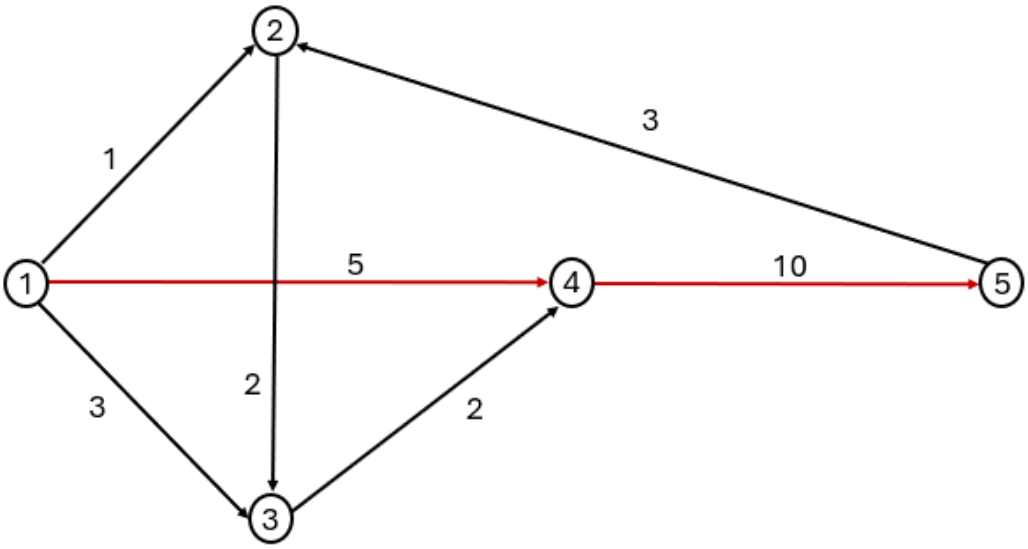
1 4 5

Por consiguiente, el camino de menor valor es 1→4→5, con un valor igual a 5 + 10 = 15. El CMV se puede representar también en MatLab con el siguiente comando:

>> highlight (h, CMV, 'EdgeColor', 'r')



y también se puede representar este camino en el grafo inicial



Si el grafo G no fuese orientado, el camino de menor valor existiría únicamente si todos los pesos fuesen no negativos, como hemos dicho anteriormente. Si usáramos

el comando shortestpath con un grafo no orientado con pesos negativos, MatLab me daría el siguiente error:

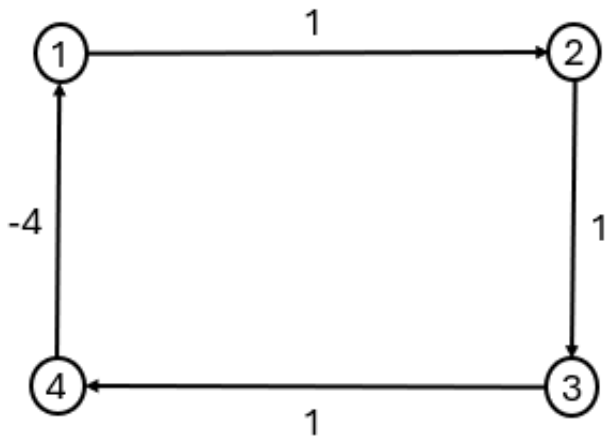
Error using graph/shortestpath

Graph edge weights must be nonnegative

Mensaje que nos indica que es imposible usar el comando si hay aristas con pesos negativos.

Por otro lado, si fuese un grafo orientado y hubiera aristas con valores negativos, el camino de menor valor existiría si no hubiese ciclos negativos. Si los hubiera, MatLab nos daría un error advirtiéndonos la existencia de dichos ciclos.

Consideremos ahora otro ejemplo con un grafo orientado con ciclos negativos como el siguiente:



Definimos en MatLab:

>> A = [0 1 0 0 ; 0 0 1 0 ; 0 0 0 1 ; - 4 0 0 0];

>> G = digraph (A)

G =

digraph with properties:

Edges: [4x2 table]

Nodes: [4x0 table]

Buscamos ahora obtener el CMV desde el vértice 1 al 4:

>> shortestpath (G, 1, 4)

Obtenemos el siguiente error:

Error using digraph/shortestpath (line 122). Shortest path distance undefined because the graph contains a negative cycle of length -1 (nodes 1 2 3 4 1).

El programa advierte un fallo al identificar un ciclo con valor negativo: 1→2→3→4→1, con valor total de 1 + 1 + 1 - 4 = - 1. Por tanto, al existir un ciclo negativo, no hay CMV.

Si el grafo no fuese conexo, MatLab proporcionaría el camino de menor valor si los dos vértices indicados estuviesen en la misma componente conexa.

8.2. Opciones del comando shortestpath

Esta instrucción ofrece las siguientes posibilidades muy útiles:

- Longitud del CMV. Además de obtener el CMV, este comando ofrece la posibilidad de entregar también la longitud de dicho camino escribiendo:

  >> [CMV, d] = shortestpath (G, s, t)

- Shortestpath por defecto. El uso del comando como venimos haciendo, usa los siguientes comandos:

  - Algoritmo Dijkstra: para usar este algoritmo debemos escribir lo siguiente:

    >> shortestpath (G, s, t, 'Method', 'positive')

    De esta manera, obligamos a MatLab a usar este algoritmo. Si no pudiese ser, por la existencia de pesos negativos, el programa no daría el siguiente error:

    Error using digraph/shortestpath

    Method value can be 'positive' only if all edge weights are nonnegative.

  - Algoritmo Bellman - Ford. También se puede forzar al programa a que aplique este algoritmo con la siguiente instrucción:

    >> shortestpath (G, s, t, 'Method', 'mixed')

Este método lo podremos usar únicamente cuando el grafo en cuestión sea orientado. No obstante, si el grafo es orientado con pesos no negativos, será más eficiente el algoritmo Dijkstra.

- Grafos acíclicos. Cuando el grafo es acíclico, existe un algoritmo más eficiente que los anteriores, que se aplica escribiendo:

  >> shortestpath (G, s, t, 'Method', 'acyclic')

Para asegurarnos que el grafo es acíclico podemos usar el comando isdag. Una vez asegurados de es acíclico, conviene usar este método al ser más eficiente.

8.3. Problema de la ruta más corta

Cuando lo que buscamos es la ruta más corta entre dos vértices, con el menor número de arcos, existen dos opciones:

Dar a todos los vértices un valor de uno, lo cual hará que el valor del camino sea igual a número de arcos.

Uso del comando shortestpath con el siguiente método:

>> shortestpath (G, s, t, 'Method', 'unweighted')

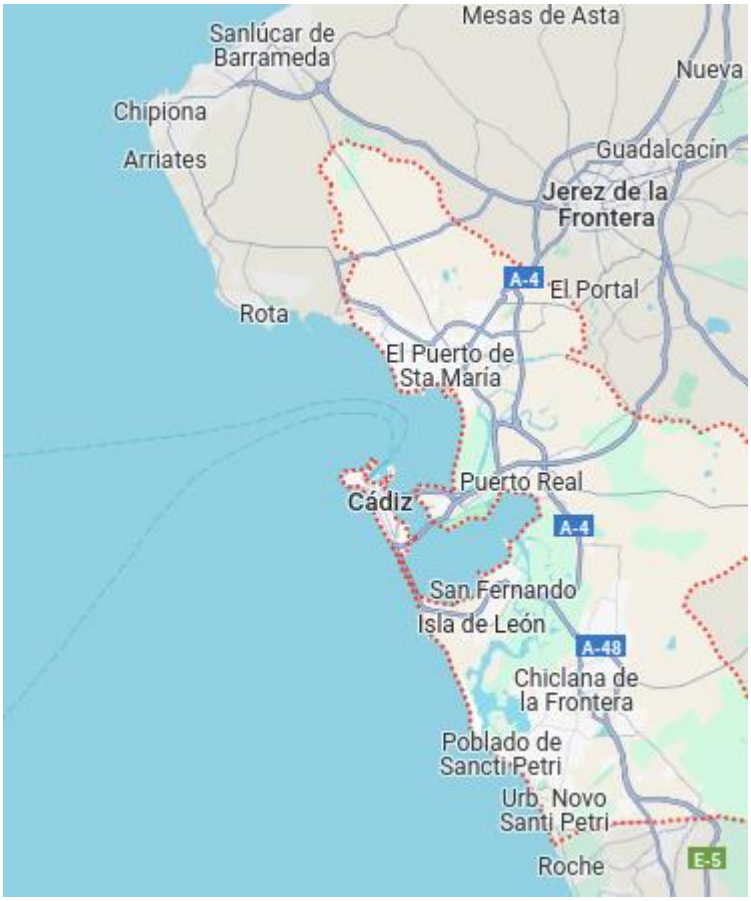
8.4. Aplicación ilustrativa

Vamos a considerar para nuestro ejemplo los núcleos de población más importantes de la bahía de Cádiz y alrededores que son los siguientes:

Núcleo urbano	Identificador
Cádiz	1
Puerto Real	2
Puerto de Santa María	3
Rota	4
Chipiona	5
Sanlúcar de Barrameda	6
San Fernando	7
Chiclana de la Frontera	8

Santi Petri	9
Roche	10

Representados en el siguiente mapa:

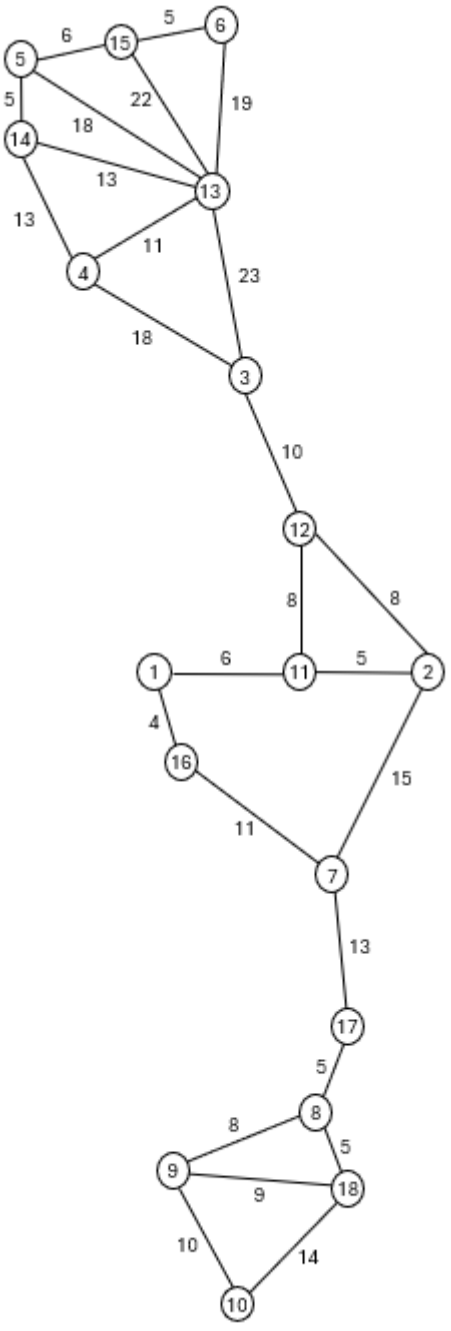


También se han elegido como vértices algunos lugares destacados como núcleos más pequeños, parques o barrios que son importantes para conectar unas zonas con otras. Estos puntos se recogen en la siguiente tabla:

Lugar destacado	Identificador
Barriada Río San Pedro	11
Parque Natural Los Toruños	12
Base Rota - Puerta de Jerez	13
Urbanización Costa Ballena	14

Venta “el menuito”	15
La Cortadura	16
Parque natural Bahía de Cádiz	17
Chiclana Park	18

Con el programa MatLab y las aplicaciones, algoritmos y métodos explicados en esta práctica, trataremos de encontrar el camino de menor valor del siguiente grafo que contiene todos los núcleos urbanos y lugares destacados indicados anteriormente:

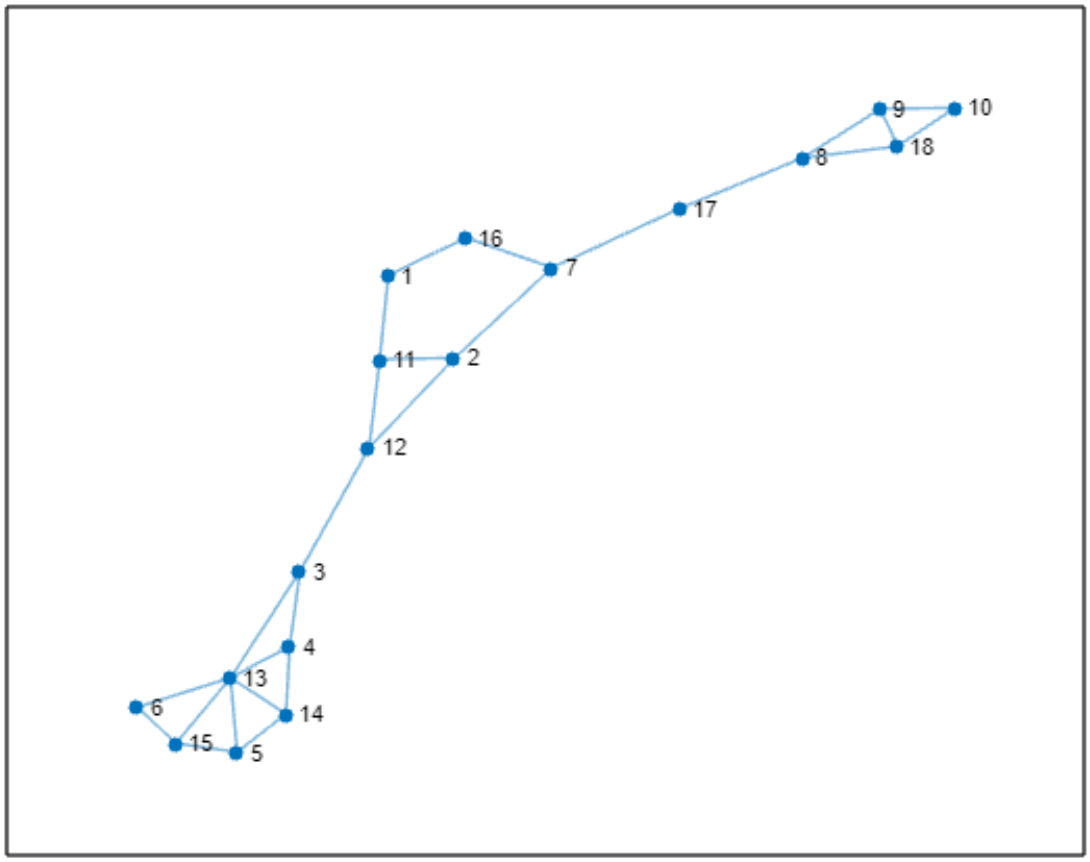


A continuación, vamos a buscar los caminos de menor valor desde Cádiz al resto de núcleos urbanos de nuestro ejemplo. Para ello, definimos el grafo en MatLab usando el comando graph, por tratarse de un grafo no orientado:

```
>> s = [1 1 2 2 2 3 3 3 4 4 5 5 5 6 6 7 7 8 8 8 9 9 10 11 13 13];

>> t = [11 16 7 11 12 4 12 13 13 14 13 14 15 13 15 16 17 9 17 18 10 18 18 12 14
15];

>> peso = [6 4 15 5 8 18 10 23 11 13 18 5 6 19 5 11 13 8 5 5 10 9 14 8 13 22];

>> G = graph (s, t, peso)

G =

graph with properties

Edges: [26x2 table]

Nodes: [18x0 table]
```

Para obtener la representación gráfica escribimos:

```
>> h = plot (G)
```



Para calcula el CMV para ir desde Cádiz hasta Puerto Real usaremos la siguiente instrucción:

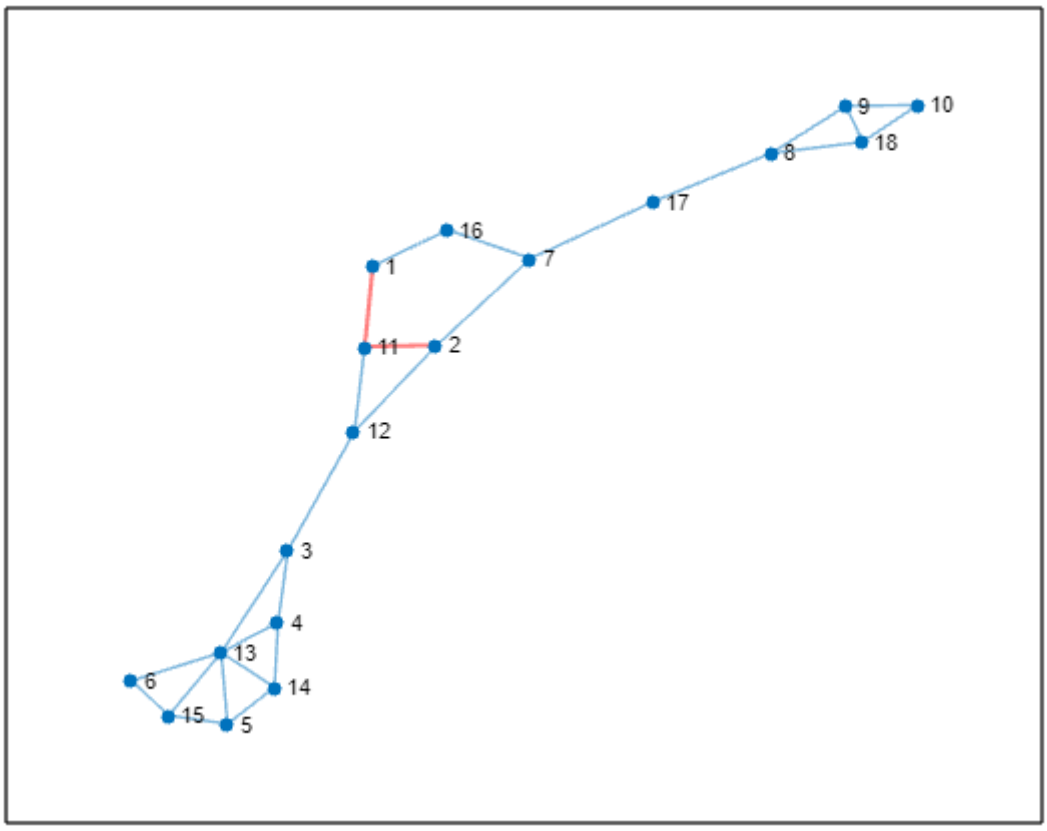
>> P2 = shortestpath (G, 1, 2)

P2 =

1 11 2

Y para obtener la representación gráfica de dicho camino escribimos:

>> highlight (h, P2, 'EdgeColor', 'r', 'LineWidth', 1.5)



De la misma manera, tras calcular los CMVs desde Cádiz al resto de núcleos urbanos podemos rellenar la siguiente tabla:

Desde	Hasta	CMV	Longitud	Distancia
Cádiz	Puerto Real	1→11→2	2	11
Cádiz	El puerto de Santa María	1→11→12→3	3	24
Cádiz	Rota	1→11→12→3→4	4	42
Cádiz	Chipiona	1→11→12→3→4→14→5	6	60
Cádiz	Sanlúcar de Barrameda	1→11→12→3→13→6	5	66
Cádiz	San Fernando	1→16→7	2	15
Cádiz	Chiclana de la frontera	1→16→7→17→8	4	33
Cádiz	Santi Petri	1→16→7→17→8→9	5	41
Cádiz	Roche	1→16→7→17→8→9→10	6	51

9. BIBLIOGRAFÍA

1. R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding minimum-costflows by double scaling. Mathematical Programming, 53(1):243–266, 1992.

2. D. Bader and K. Madduri. Design and implementation of the HPCS graph analysisbenchmark on symmetric multiprocessors. In Proc. 12th Internat. Conf. HighPerform. Comput., volume 3769 of Lecture Notes Comput. Sci., pages 465–476.2005.

3. D. A. Castanon. Efficient algorithms for finding the k best paths through a trellis.IEEE Trans. Aerospace and Electronic Systems, 26(2):405–410, 1990.

4. M. Charikar, M. Hajiaghayi, and H. Karloff. Improved approximation algorithmsfor label cover problems. In Proc. 17th Annu. European Sympos. Algorithms, vol-ume 5757 of Lecture Notes Comput. Sci., pages 23–34. 2009.

5. G. Even, G. Kortsarz, and W. Slany. On network design problems: fixed cost flowsand the covering steiner problem. ACM Trans. Algorithms, 1(1):74–101, 2005.

6. M. Garey and D. S. Johnson. Computers and intractability : A guide to the theoryof NP-completeness. W. H. Freeman, 1979.

7. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. J. ACM,45(5):783–797, 1998.

8. Y. Kobayashi and C. Sommer. On shortest disjoint paths in planar graphs. DiscreteOptimization, 7(4):234–245, 2010.

9. S. O. Krumke, H. Noltemeier, S. Schwarz, H.-C. Wirth, and R. Ravi. Flow im-provement and network flows with fixed costs. In Proc. Internat. Conf. Oper. Res.:OR-98, pages 158–167, 1998

(15) (PDF) Finding Paths with Minimum Shared Edges.

[10] Bor˚uvka O, (1926) About a certain minimal problem. Pr´ace moravsk´e pr´ırodovedeck´e spolecnosti v Brne 3:37–58, In Czech.

[11] Bird C (1976) On cost allocation for a spanning tree: a game theoretic approach. Networks 6(4):335-350.

[12] Dutta B, Kar A (2004) Cost monotonicity, consistency and minimum cost spanning tree games. Games Econom Behav 48(2):223-248.

[13] Feltkamp V, Tijs S, Muto S (1994) On the irreducible core and the equal remaining obligation rule of minimum cost extension problems. Technical Report 106, CentER DP 1994, Tilburg University.

[14] Fern´andez F.R, Hinojosa M.A, Puerto J (2002) Core Solutions in Vector-Valued Games. Journal of Optimization Theory and Applications 112(2):331-360.

[15] Fern´andez F.R, Hinojosa M.A, Puerto J (2003) Multi-criteria minimum cost spanning tree games. Eur J Oper Res 158(2):399-408.

[16] Fraga Canosa M.C, (2022) Cooperation on minimum cost arborescence problems. Repositorio M´aster en T´ecnicas Estad´ısticas - USC.

17. Real Academia Española. «grafo : Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio.». Diccionario de la lengua española (23.ª edición). Consultado el 14 de agosto de 2019.

18. Trudeau, Richard J. (1993). Dover Pub., ed. Introduction to Graph Theory (Edición corregida y aumentada.). ISBN 978-0-486-67870-2.

19. Trudeau, Richard J. (1993). Introduction to Graph Theory (Corrected, enlarged republication. edición). New York: Dover Pub. p. 19. ISBN 978-0-486-67870-2. Archivado desde el original el 5 de mayo de 2019. Consultado el 8 de agosto de 2012. «A graph is an object consisting of two sets called its vertex set and its edge set.»

20 J. J. Sylvester (February 7, 1878) "Chemistry and algebra," Archivado el 12 de abril de 2023 en Wayback Machine. Nature, 17 : 284. doi 10.1038/017284a0. From page 284: "Every invariant and covariant thus becomes expressible by a graph precisely identical with a Kekuléan diagram or chemicograph."

21J. J. Sylvester (1878) "On an application of the new atomic theory to the graphical representation of the invariants and covariants of binary quantics, – with three appendices," Archivado el 4 de febrero de 2023 en Wayback Machine. American Journal of Mathematics, Pure and Applied, 1 (1) : 64–90. doi 10.2307/2369436. JSTOR 2369436

22 Gross, Jonathan L.; Yellen, Jay (2004). Handbook of graph theory. CRC Press. p. 35. ISBN 978-1-58488-090-5. Archivado desde el original el 4 de febrero de 2023. Consultado el 16 de febrero de 2016.