3. Estructuras de datos Grafo

Introducción

El mundo de hoy puede ser representado en grafos para entenderlo casi a plenitud. Los modelos de grafos para gestión de bases de datos son extremadamente potentes y nos permiten conocer información que está escondida a simple vista.

Vamos a profundizar el mundo de los grafos para que descubramos juntos sus potencialidades, tipos, orden y algunas herramientas de visualización que puedes utilizar para tus proyectos.

¿Qué son grafos?

Los grafos son una composición interesante de conjuntos de objetos que denominamos nodos. En ellos se almacena diferentes tipos de elementos o datos que podemos utilizar para procesar o conocer con fines específicos.

Adicionalmente estos nodos, suelen estar unidos o conectados a otros nodos a través de elementos que denominamos aristas.

Los nodos pertenecientes a un grafo pueden contener datos estructurados o no estructurados y al interrelacionarse con otros nodos producen relaciones interesantes que podemos analizar con diferentes finalidades.

Potencialidades de los grafos

Estos elementos matemáticos que conocemos como grafos son muy utilizados en el estudio de ciencias naturales y otras áreas del conocimiento.

En el ámbito empresarial y tecnológico se han empezado a explotar sus potencialidades.

Su alcance con fines empresariales recorre desde la posibilidad de estudiar las relaciones de los clientes con los productos que ofrecemos, hasta la posibilidad de entender a profundidad todos y cada uno de los procesos relacionales que se desarrollan dentro de una organización y que son imperceptibles a simple vista.

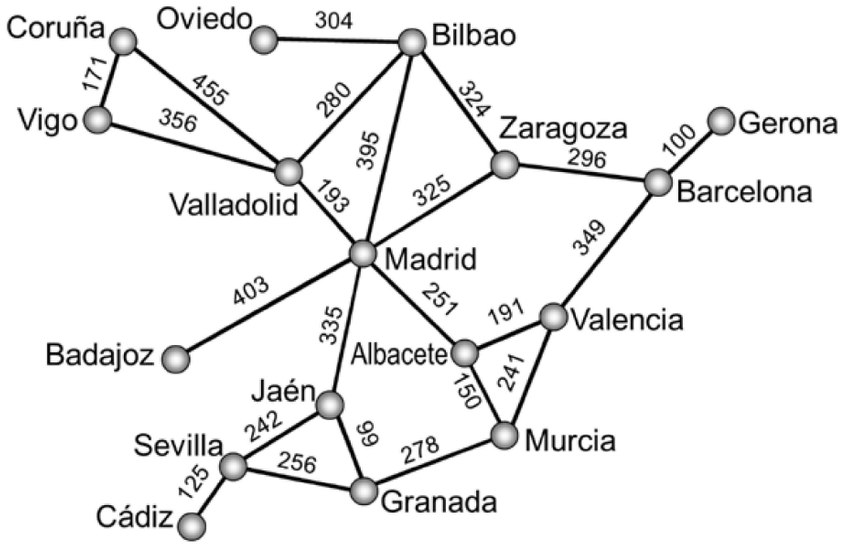
Los grafos inclusive son útiles para combatir el crimen. Recientemente se han aprovechado sus ventajas inigualables para estudiar patrones sospechosos de fraude electrónico y bancario.

Para mayor información consulte el siguiente link:

3.1. Ejemplos de aplicación de grafos.

Ejemplos de en donde se pueden aplicar las estructuras de datos Gafo

- **Ejemplo 1:** Grafo de carreteras entre ciudades.



Problemas que se pueden plantear con el grafo entre ciudades

¿Cuál es el camino más corto de Murcia a Badajoz?

¿Existen caminos entre todos los pares de ciudades?

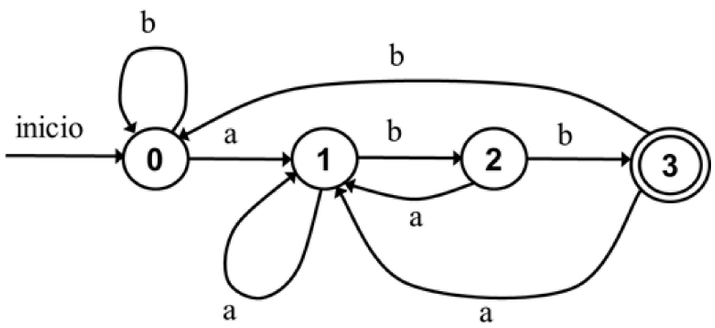
¿Cuál es la ciudad más lejana a Barcelona?

¿Cuál es la ciudad más céntrica?

¿Cuántos caminos distintos existen de Sevilla a Zaragoza?

¿Cómo hacer un tour entre todas las ciudades en el menor tiempo posible?

- **Ejemplo 2:** Grafo de transiciones de un AFD (Autómata Finito Determinista).



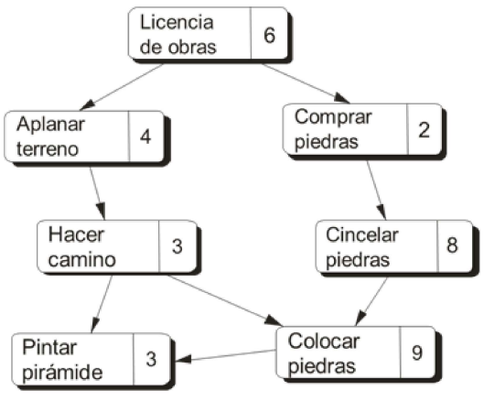
Problemas que se pueden plantear con el grafo de transiciones de un AFD.

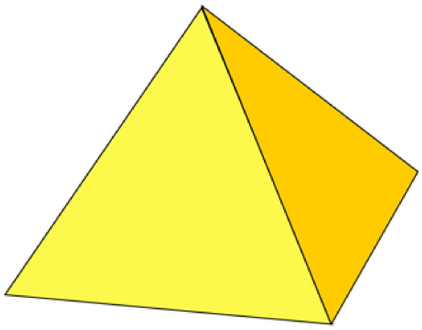
¿La expresión: a b b a b a b b b a, es una expresión válida del lenguaje?

¿Cuál es la expresión válida más corta?

Transformar el grafo en una expresión regular y viceversa.

- **Ejemplo 3:** Grafo de planificación de tareas.





Problemas que se pueden plantear con el grafo de planificación de tareas.

¿En cuánto tiempo, como mínimo, se puede construir la pirámide?

¿Cuándo debe empezar cada tarea en la planificación óptima?

¿Qué tareas son más críticas (es decir, no pueden sufrir retrasos)?

¿Cuánta gente necesitamos para acabar las obras?

3.2. Conceptos básicos

Conceptos básicos, notación y definiciones.

Definición de Grafo:

- Un grafo G es una tupla G= (V, A), donde V es un conjunto no vacío de vértices o nodos y A es un conjunto de aristas o arcos.

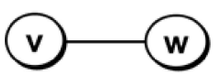
- Cada arista es un par (v, w), donde v, w ∈ V.

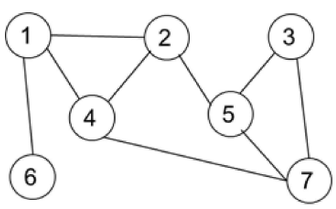
Tipos de grafos

- **Grafo no dirigido**.

Las aristas no están ordenadas:

(v, w) = (w, v)





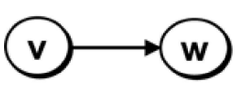
G = { **V** = {1.2,3,4,5,6,7}
A = { (1,2),(1,4),(1,6),(2,4),(2,5),
(3,5),(3,7),(4,7),(5,7) }
}

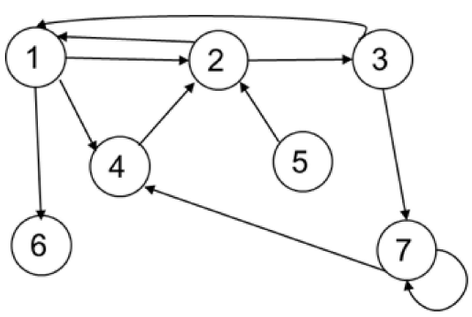
- **Grafos dirigidos** (o **digrafos**).

Las aristas son pares ordenados:

<v, w> ≠ <w, v>

<v, w> ⇒ w = cabeza de la arista, v = cola.





G = { **V** = {1.2,3,4,5,6,7}
A = { <1,2>,<1,4>,<1,6>,<2,1>,<2,3>,
<3,1>,<3,7>,<4,2>,<5,2>,<7,4>,<7,7>)
}
}

Terminología de grafos.

- **Nodos adyacentes a un nodo v**: todos los nodos unidos a v mediante una arista.

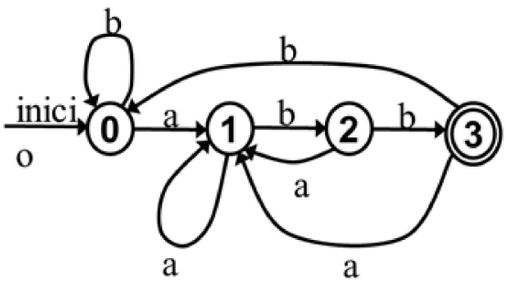
- En grafos dirigidos:

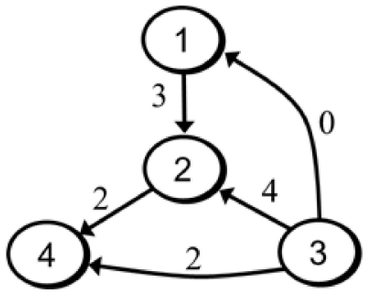
– **Nodos adyacentes a v**: todos los w con <v, w> ∈ A.

– **Nodos adyacentes de v**: todos los u con <u, v> ∈ A.

- Un grafo está **etiquetado** si cada arista tiene asociada una etiqueta o valor de cierto tipo.
- **Grafo con pesos o ponderado**: es un grafo etiquetado con valores numéricos.
- Grafo etiquetado: G= (V, A, W), con **W**: A → TipoEtiq

Grafos dirigidos etiquetados con pesos.





- **Camino de un vértice w_1 a w_q**: es una secuencia $w_1, w_2, ..., w_q \in V$, tal que todas las aristas $(w_1, w_2), (w_2, w_3), ..., (w_{q-1}, w_q) \in A$.
- **Longitud de un camino**: número de aristas del camino = n° de nodos -1.
- **Camino simple**: aquel en el que todos los vértices son distintos (excepto el primero y el último que pueden ser iguales).
- **Ciclo**: es un camino en el cual el primer y el último vértice son iguales. En grafos no dirigidos las aristas deben ser diferentes.
- Se llama **ciclo simple** si el camino es simple.

Definición de Subgrafo:

Un **subgrafo** de G=(V, A) es un grafo G'=(V', A') tal que V' ⊆ V y A' ⊆ A.

Dados dos vértices v, w, se dice que están conectados si existe un camino de v a w.

Un grafo es conexo (o conectado) si hay un camino entre cualquier par de vértices.

Si es un grafo dirigido, se llama fuertemente conexo.

Un componente (fuertemente) conexo de un grafo G es un subgrafo maximal (fuertemente) conexo.

- Un grafo es **completo** si existe una arista entre cualquier par de vértices.
- Para n nodos, ¿cuántas aristas tendrá un grafo completo (dirigido o no dirigido)?
- **Grado de un vértice v**: número de arcos que inciden en él.
- Para grafos dirigidos:

  - **Grado de entrada de v**: n° de aristas con <x, v>

  - **Grado de salida de v**: n° de aristas con <v, x>

Operaciones elementales con grafos:

- **Crear un grafo vacío** (o con n vértices).

- **Insertar un nodo o una arista.**
- **Eliminar un nodo o arista.**
- **Consultar si existe una arista** (obtener la etiqueta).

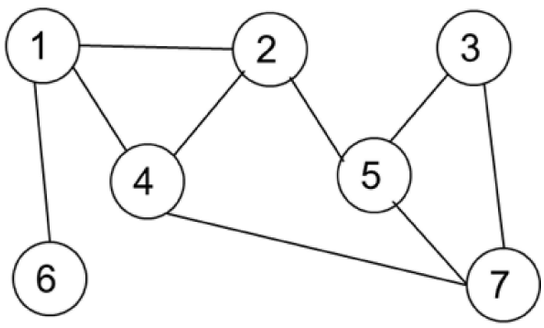
Iteradores sobre las aristas de un nodo:

para todo nodo **w adyacente a v** hacer
acción sobre w

para todo nodo **w adyacente de v** hacer
acción sobre w (*Mucho menos frecuente*)

Pregunta de Elección Múltiple

Dado el siguiente grafo, responda las siguientes preguntas:



¿Cuál de los siguientes caminos de 4 a 3 no es válido?

- ○ 1) (4,1), (1,2), (2,5), (5,7), (7,3)
- ○ 2) (4,7), (7,5), (5,3)
- ○ 3) (4,2), (2,7), (7,3)

Incorrecto

Incorrecto

Opción correcta

Solución

1. **Incorrecto**
2. **Incorrecto**
3. **Opción correcta**

¿Cuál de los siguientes caminos no es un camino simple?

- ○ 1) (3,5), (5,7), (7,4), (4,1)
- ○ 2) (3,5), (5,2), (2,4), (4,7), (7,5), (5,2), (2,1)
- ○ 3) (3,7), (7,4), (4,1)

Incorrecto

Opción correcta

Incorrecto

Solución

1. **Incorrecto**
2. **Opción correcta**
3. **Incorrecto**

¿Cuál de los siguientes caminos no es un ciclo?

- ○ 1) (1,4), (4,2), (2,1)
- ○ 2) (4,7), (7,3), (3,5), (5,7), (7,4)
- ○ 3) (5,3), (3,7), (7,4), (4,5)

Incorrecto

Incorrecto

Opción correcta

Solución

1. **Incorrecto**
2. **Incorrecto**
3. **Opción correcta**

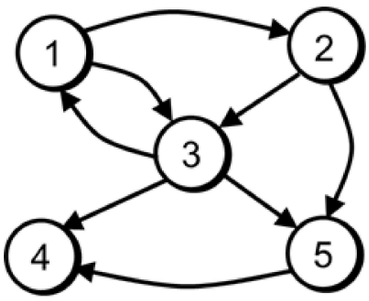
3.3. Representación de grafos.

Representaciones básicas de Grafos

- Para la representación de grafos se deben considerar:

  - Representación del conjunto de **nodos** o **vértices**, **V**.

  - Representación del conjunto de **aristas**, **A**.

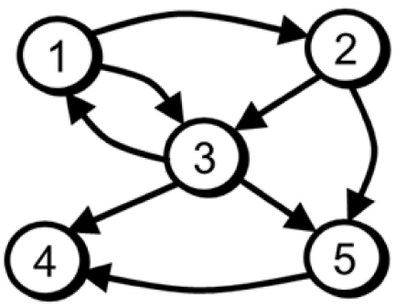


- Ojo: las aristas son relaciones “muchos a muchos” entre nodos...

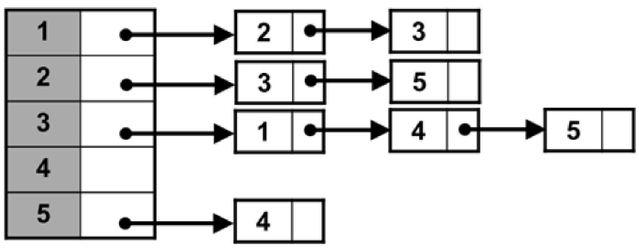
Representación del conjunto de aristas A para un grafo dirigido.

- Mediante matriz de adyacencia.

M	1	2	3	4	5
1		T	T		
2			T		T
3	T			T	T
4					
5				T	

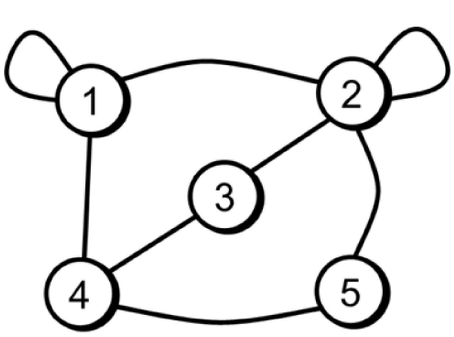


- Mediante listas de adyacencia.



Matriz de adyacencia para un grafo no dirigido.

- Sea M de tipo GrafoNoEtiq, G= (V, A).
- M[v, w] = cierto ⇔ (v, w) ∈ A



M	1	2	3	4	5
1	T	T		T	
2	T	T	T		T
3		T		T	
4	T		T		T
5		T		T	

Grafo no dirigido → Matriz simétrica: M[i, j] = M[j, i].

Resultado: se desperdicia la mitad de la memoria.

Ventajas y desventajas al utilizar Matrices de adyacencia.

Uso de memoria

- k_2 bytes/etiqueta
- Memoria usada: k_2n^2

Ventajas

- Representación y operaciones muy sencillas.
- Eficiente para el acceso a una arista dada.

Inconvenientes

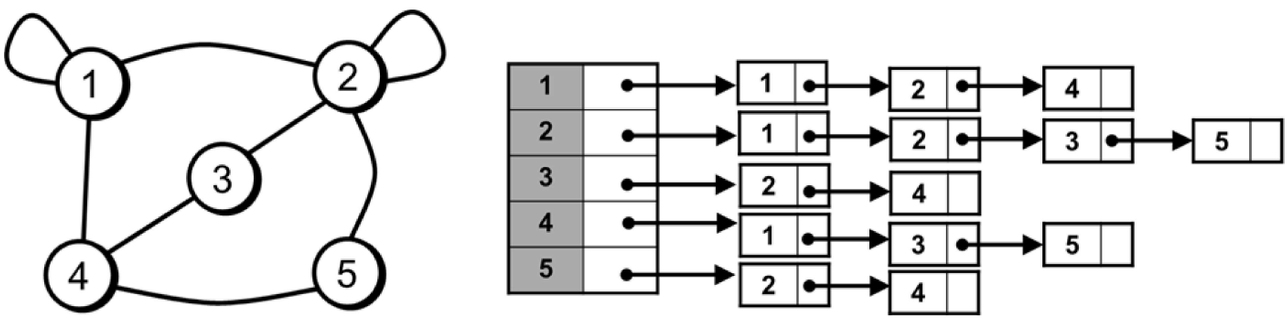
- El número de nodos del grafo no puede cambiar.
- Si hay muchos nodos y pocas aristas ($a<<n^2$) se desperdicia mucha memoria.

Listas de adyacencia para un grafo no dirigido.

tipo Nodo= entero (1..n)

tipo GrafoNoEtiq= array [1..n] de Lista[Nodo]

- Sea R de tipo GrafoNoEtiq, G= (V, A).
- La lista R[v] contiene los w tal que (v, w) ∈ A.



Grafo no dirigido à Las aristas están repetidas.

Resultado: también se desperdicia memoria.

Ventajas y desventajas al utilizar Listas de adyacencia.

Uso de memoria

- k_1 bytes/puntero, k_2 bytes/etiqueta o nodo
- Memoria usada: $k_1(n+a) + 2k_2a$
- Con matrices de adyacencia: k_2n^2
- ¿Cuál usa menos memoria?

Ventajas

- Más adecuada cuando $a << n^2$.

Inconvenientes

- Representación más compleja.
- Es ineficiente para encontrar las aristas que llegan a un nodo.

Ver el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=460 <https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=460>

3.4. Problemas y algoritmos sobre grafos.

Dentro de los problemas más comunes a los cuales nos enfrentamos en la teoría de grafos tenemos:

- Recorridos sobre grafos.
- Árboles de expansión mínimos.
- Problemas de caminos mínimos.

3.4.1. Recorridos sobre grafos.

- Idea similar al recorrido en un árbol.
- Se parte de un nodo dado y se visitan los vértices del grafo de manera ordenada y sistemática, moviéndose por las aristas.

Tipos de recorridos:

- **Búsqueda primero en profundidad**. Equivalente a un recorrido en preorden de un árbol.
- **Búsqueda primero en amplitud o anchura**. Equivalente a recorrer un árbol por niveles.
- Los recorridos son una herramienta útil para resolver muchos problemas sobre grafos.
- El recorrido puede ser tanto para grafos dirigidos como no dirigidos.
- Es necesario llevar una cuenta de los nodos visitados y no visitados, en este caso se utiliza un arreglo de tamaño igual al

número de vértices que contiene el grafo.

```
var
        marca: array [1, ..., n] de (visitado, noVisitado)
```

En seguida se inicializa el arreglo en todas sus entradas como noVisitado

```
operación BorraMarcas()
        para i:= 1, ..., n hacer
                marca[i]:= noVisitado
```

Consultar el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=468 <https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=468>

3.4.1.1 Recorrido a lo profundo.

Búsqueda primero en profundidad

Algoritmo recursivo de búsqueda primero en profundidad.

operación **bpp** (v: nodo)

```
marca[v]:= visitado
para cada nodo w adyacente a v hacer
    si marca[w] == noVisitado entonces
                bpp(w)
finpara
```

Método en donde se hace el llamado al algoritmo bpp()

operación BúsquedaPrimeroEnProfundidad

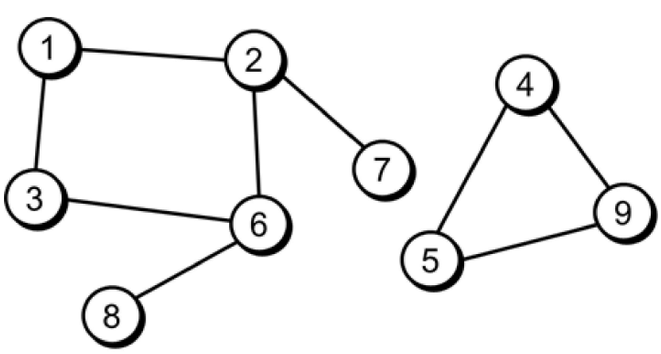
```
BorraMarcas
para v:= 1, ..., n hacer
        si marca[v] == noVisitado entonces
                bpp(v)
finpara
```

El recorrido no es único: depende del nodo inicial y del orden de visita de los adyacentes.

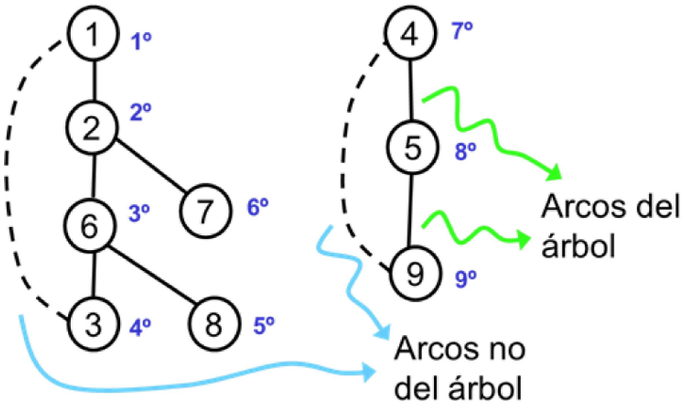
El orden de visita de unos nodos a partir de otros puede ser visto como un árbol: árbol de expansión en profundidad asociado al grafo.

Si aparecen varios árboles: bosque de expansión en profundidad.

Ejemplo 1 Grafo no dirigido.



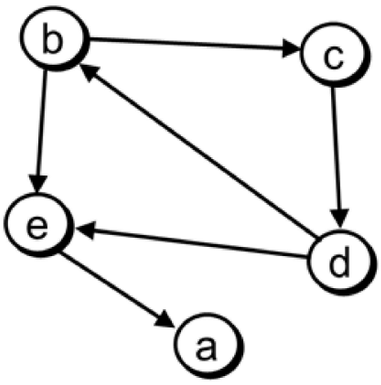
Árbol de expansión en profundidad:



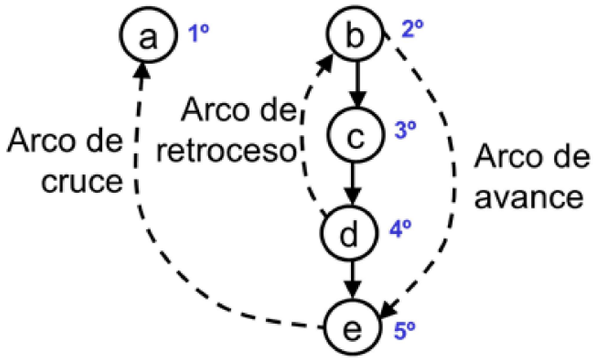
Arcos no del árbol: si marca[v] == noVisitado ...

→ se detectan cuando la condición es falsa.

Ejemplo 2 Grafo dirigido.



Árbol de expansión en profundidad:



- ¿Cuánto es el tiempo de ejecución de BPP?
- Imposible predecir las llamadas en cada ejecución.
- **Solución**: medir el “trabajo total realizado”.

3.4.1.2. Recorrido a lo ancho.

Búsqueda primero en anchura (o amplitud)

- **Búsqueda en anchura empezando en un nodo v:**

– Primero se visita v.

– Luego se visitan todos sus adyacentes.

– Luego los adyacentes de estos y así sucesivamente.

- El algoritmo utiliza una cola de vértices.
- Operaciones básicas:

– Sacar un elemento de la cola.

– Añadir a la cola sus adyacentes no visitados.

Método donde se hace el llamado a bpa() (búsqueda primero en anchura)

```
operación BúsquedaPrimeroEnAnchura
        BorraMarcas
        para v:= 1, ..., n hacer
                si marca[v] = noVisitado entonces
                        bpa(v)
        finpara
```

Algoritmo de búsqueda primero en anchura

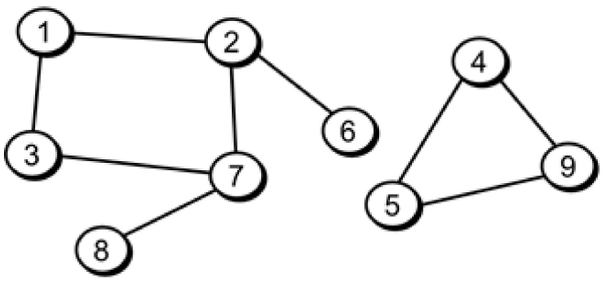
```
operación bpa (v: Nodo)
var  C: Cola[Nodo]
     x, y: Nodo
        marca[v]:= visitado
        InsertaCola (v, C)
        mientras NOT EsVacíaCola (C) hacer
            x:= FrenteCola (C)
            SuprimirCola (C)
            para cada nodo y adyacente a x hacer
                si marca[y] == noVisitado entonces
                        marca[y]:= visitado
                        InsertaCola (y, C)
```

finsi

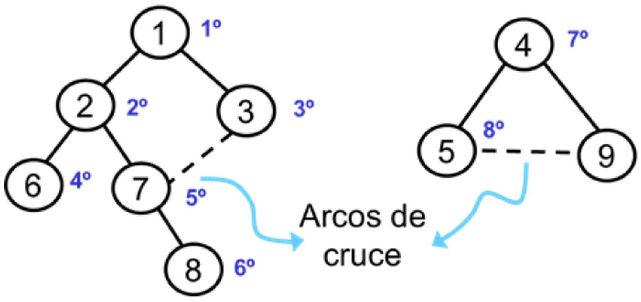
finpara

finmientras

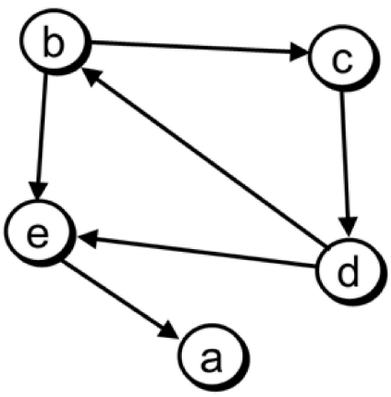
Ejemplo 1 Grafo no dirigido.



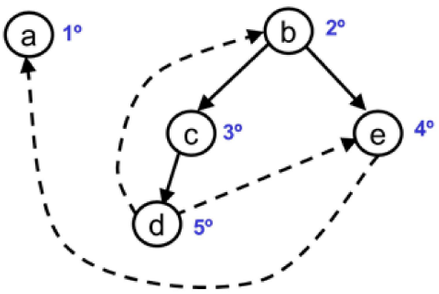
Árbol de expansión en anchura.



Ejemplo 2 Grafo dirigido.



Árbol de expansión en anchura.



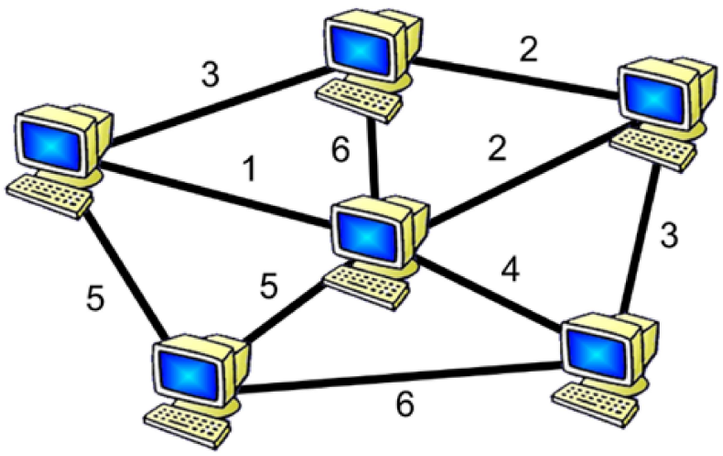
- ¿Cuánto es el tiempo de ejecución de la BPA?
- ¿Cómo comprobar si un arco es de avance, cruce, etc.?
- **Solución:** Construir el bosque explícitamente.

3.4.2. Árboles de expansión mínimos.

- **Definición:** Un **árbol de expansión** de un grafo G=(V, A) no dirigido y conexo es un subgrafo G'=(V, A') conexo y sin ciclos.
- **Ejemplo:** los árboles de expansión en profundidad y en anchura de un grafo conexo.
- En grafos con pesos, el **costo del árbol de expansión** es la suma de los costes de las aristas.

- *Problema del árbol de expansión de costo mínimo*:

Dado un grafo ponderado no dirigido, encontrar el árbol de expansión de menor costo.



- **Problema:** conectar todas las computadoras con el menor costo total.
- **Solución:** algoritmos clásicos de Prim y Kruskal.

3.4.2.1. Algoritmo de Prim.

Esquema del algoritmo de Prim:

Empezar en un vértice cualquiera v. El árbol consta inicialmente sólo del nodo v.
Del resto de vértices, buscar el que esté más próximo a v (es decir, con la arista (v, w) de coste mínimo).
Añadir w y la arista (v, w) al árbol.
Buscar el vértice más próximo a cualquiera de estos dos. Añadir ese vértice y la arista al árbol de expansión.
Repetir sucesivamente hasta añadir los n vértices.

- La solución se construye **poco** a **poco**, empezando con una solución "vacía".
- Implícitamente, el algoritmo maneja los **conjuntos**:

– **V**: Vértices del grafo.

– **U**: Vértices añadidos a la solución.

– **V-U:** Vértices que quedan por añadir.

– ¿Cómo implementar eficientemente la búsqueda: encontrar el vértice de **V-U** más próximo a alguno de los de **U**?

- Se usan dos arrays:

– **MAS_CERCANO:** Para cada vértice de **V-U** indica el vértice de **U** que se encuentra más próximo.

– **MENOR_COSTO:** Indica el costo de la arista más cercana.

–

Estructura del algoritmo de Prim: **C[v, w] Matriz de costos**

1. Inicialmente U= {1}. MAS_CERCANO[v]= 1.

MENOR_COSTO[v]= C[1, v], para v= 2..n

(en este caso se supone que se inicia desde el vértice 1, pero puede ser desde cualquier otro vértice v)

2. Buscar el nodo v, con MENOR_COSTO mínimo.
Asignarle un valor muy grande (para no volver a cogerlo).

3. Recalcular MAS_CERCANO y MENOR_COSTO de los nodos
de **V-U**. Para cada w de **V-U**, comprobar si **C[v, w]** es menor que **MENOR_COSTO[w]**.

4. Repetir los dos puntos anteriores hasta que se hayan añadido los n nodos.

Pseudocódigo del Algoritmo de Prim

Prim (Grafo **G**, nodo inicial **s**)

visitado [n] = { false , . . . , false } / / indica si un nodo ya fue visitado

distancia [n] = { Infinito , . . . , Infinito } / / guarda las distancias de cada nodo al conjunto visitado

para cada w en V[G] hacer

si existe arista entre s y w **entonces**

distancia [w] = peso (s, w)

distancia [s] = 0

visitado [s] = true

mientras que no estén visitados todos **hacer**

v = nodo de menor distancia del conjunto que no ha sido visitado aun (iterar el arreglo)

visitado [v] = true

para cada w en los vecinos de v **hacer**

si distancia[w] > peso (v, w) **entonces**

distancia [w] = peso (v, w)

padre[w] = v

finsi

finpara

finmientras

Consultar el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=503
<https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=503>

3.4.2.2. Algoritmo de Kruskal.

Esquema del algoritmo de Kruskal: G= (V, A)

1. Empezar con un grafo sin aristas: G'= (V, Ø)
2. Seleccionar la arista de menor coste de A.
3. Si la arista seleccionada forma un ciclo en G', eliminarla. Si no, añadirla a G'.
4. Repetir los dos pasos anteriores hasta tener n-1 aristas.

- **¿Cómo saber si una arista (v, w) provocará un ciclo en el grafo G'?**

Implementación del algoritmo

- Necesitamos:
  - Ordenar las aristas de A, de menor a mayor: **O(a log a)**.
  - Saber si una arista dada (v, w) provocará un ciclo.
- ¿Cómo comprobar rápidamente si **(v, w)** forma un ciclo?
- Una arista **(v, w)** forma un ciclo si **v** y **w** están en el mismo componente conexo.
- La relación "estar en el mismo componente conexo" es una **relación de equivalencia**.
- Usamos la estructura de relaciones de equivalencia con punteros al padre:

– Inicialización: crear una **relación de equivalencia** vacía (cada nodo es un componente conexo).

– Seleccionar las aristas (v, w) de menor a mayor.

– La arista forma ciclo si: **Encuentra(v) = Encuentra(w)**

– Añadir una arista **(v, w)**: **Unión(v, w)** (juntar dos componentes conexos en uno).

- Analizar y responder la siguiente pregunta.

¿Cuál es el orden de complejidad del algoritmo?

Conclusiones

- Ambos algoritmos (**Prim** y **Kruskal**) encuentran siempre la solución óptima.
- ¿ La solución obtenida será la misma, o no...?
- La estructura de los dos algoritmos es muy parecida:

– Empezar con una solución "vacía".

– Añadir en cada paso un elemento a la solución (**Prim**: **un nodo**; **Kruskal**: **una arista**).

– Una vez añadido un elemento a la solución, no se quita (no se "deshacen" las decisiones tomadas).

Consultar el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=506
<https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=506>

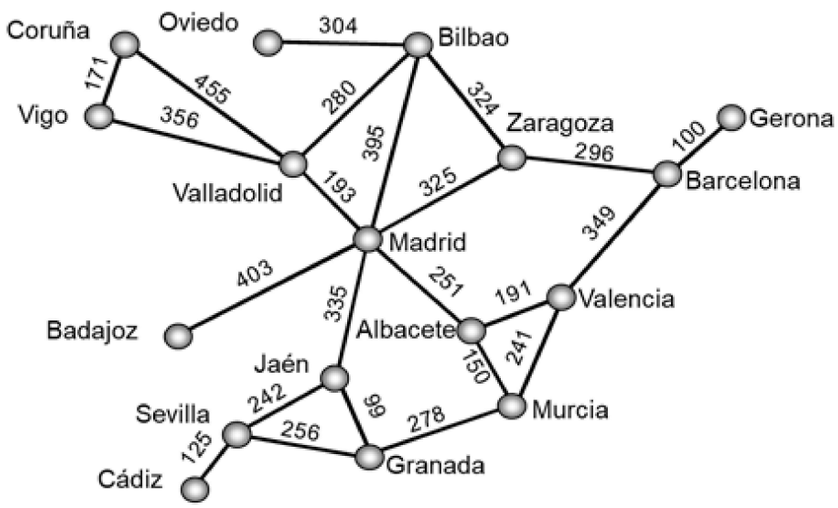
3.4.3. Problemas de caminos mínimos.

- **Costo de un camino**: suma de los costos de las aristas por las que pasa.
- **Algunos problemas de caminos mínimos:**

  – Camino mínimo entre dos nodos, **v** y **w**.

  – Caminos mínimos entre un nodo **v** y todos los demás.

  – Caminos mínimos entre todos los pares de nodos.



3.4.3.1. Algoritmo de Dijkstra.

Caminos mínimos desde un origen

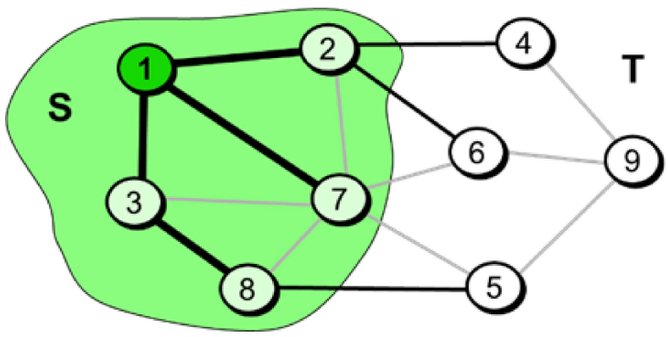
Algoritmo de Dijkstra

- Supongamos un grafo **G**, con pesos positivos y un nodo origen **v**.
- El algoritmo trabaja con dos conjuntos de nodos:

– **Escogidos**: **S**. Nodos para los cuales se conoce ya el camino mínimo desde el origen.

– **Candidatos**: **T**. Nodos pendientes de calcular el camino mínimo, aunque conocemos los caminos mínimos desde el origen pasando por nodos de **S**.

- **Camino especial**: camino desde el origen hasta un nodo, que pasa sólo por nodos escogidos, **S**.



- **Idea:** En cada paso, coger el nodo de **T** con menor distancia al origen. Añadirlo a **S**.
- **Recalcular los caminos mínimos** de los demás candidatos, pudiendo pasar por el nodo escogido.

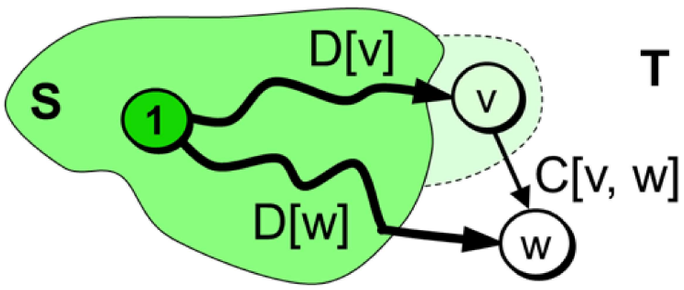
Esquema del Algoritmo de Dijkstra

- Inicialización: **S**= {1}, **T**= {2, ..., n}, caminos especiales mínimos = caminos directos.
- Repetir n-1 veces:
  - Seleccionar el nodo **v** de **T** con el camino especial más corto.
  - **Proposición**: el camino mínimo para este nodo **v**, coincide con su camino especial.
  - Recalcular los caminos especiales para los nodos de **T**, pudiendo pasar por **v**.

Implementación del algoritmo de Dijkstra

- Suponemos que el origen es el nodo 1 (para esta descripción, pero el nodo inicial puede ser cualquier nodo vI).
- **D**: array [2..n] de reales. **D[v]** almacena el costo del camino especial mínimo para el nodo **v**.
- **P**: array [2..n] de enteros. **P[v]** almacena el último nodo en el camino especial mínimo de **v**.
- **Inicialización**: **D[v]**:= **C[1, v]**, **P[v]**:= 1 (recordar que en esta descripción el vértice inicial es 1, pero puede ser cualquier vértice vI)
- **Nodo seleccionado**: nodo de **T** con mínimo **D[v]**

- **Actualización**: para todos los **w** de **T** hacer
  si **D[v]** + **C[v, w]** < **D[w]** entonces
  D[w]:= **D[v]** + **C[v, w]**
  P[w]:= **v**
  finsi



- **Camino especial para w:**
  - Sin pasar por **v**: **D[w]**
  - Pasando por **v**: **D[v]** + **C[v,w]**
  - Nos quedamos con el menor.
- Si el menor es pasando por **v** entonces: **P[w]**= **v**.
- Camino especial para **w**:

1 → ... → P[P[P[w]]] → P[P[w]] → P[w] → w

Pseudocódigo del Algoritmo de Dijkstra

- **Entrada**:
  C: array [1..n, 1..n] de real → Matriz de costes
- **Salida**:
  D: array [2..n] de reales → Costos de caminos mínimos
  P: array [2..n] de enteros → Nodos de paso
- Datos para cálculos intermedios:
  S: array [2..n] de booleano → Nodos escogidos
- **Inicialización**: **para** v:= 2, ..., n **hacer**
  D[v]:= C[1, v]
  P[v]:= 1
  S[v]:= FALSE
  finpara

para i:= 1, ..., n-1 **hacer**

v:= vértice con D[v] mínimo y S[v]==FALSE ⇒ S[v]:= TRUE

para cada vértice w adyacente a v **hacer**

si (S[w]==FALSE) y (D[v]+C[v,w]<D[w]) **entonces**

D[w]:= D[v] + C[v, w]

P[w]:= v

finsi

finpara

finpara

Pseudocódigo del algoritmo recursivo para Imprimir el camino más corto

ImprimeCamino(v: integer)

inicia

si v<> 1 entonces (recordar que aquí se supone que el vértice inicial es 1, pero puede ser cualquier vértice vI)

ImprimeCamino(P[v]);

escribe(v);

termina;

Eficiencia del algoritmo de Dijkstra

- **Con matrices de adyacencia:**

– Inicialización: **O(n)**

– Ejecutar n-1 veces:

- Buscar el nodo con mínimo **D[v]** y **S[v]** = falso: **O(n)**
- Actualizar los valores de los candidatos: **O(n)**

– En total: **$O(n^2)$**

- **Con listas de adyacencia:**

– Seguimos teniendo un **$O(n^2)$**

– Podemos modificar la implementación y conseguir un **O(a·log n)**. Será adecuada cuando **a << n^2**.

Consultar el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=495 <https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=495>

3.4.3.2. Algoritmo de Floyd.

Caminos mínimos entre todos los pares.

- **Problema**: Calcular los caminos mínimos entre todos los pares de nodos del grafo.

  Posibilidades

- Aplicar el algoritmo de Dijkstra n veces, una por cada posible nodo origen:

– Con matrices de adyacencia: **$O(n^3)$**

– Con listas de adyacencia: **$O(a \cdot n \cdot \log n)$**

- **Aplicar el algoritmo de Floyd**:

– Con listas o matrices: **$O(n^3)$**

Pero más sencillo de programar...

- **Entrada**:

  C: array [1..n, 1..n] de real → Matriz de costos

- **Salida**:

  D: array [1..n, 1..n] de reales → Costos caminos mínimos

Algoritmo de Floyd

```
D:= C
para k:= 1, ..., n hacer
    para i:= 1, ..., n hacer
        para j:= 1, ..., n hacer
            D[i, j]:= min ( D[i, j] , D[i, k] + D[k, j] )
```

- **¿En qué se basa el algoritmo de Floyd?**

- En cada paso k, la matriz **D** almacena los caminos mínimos entre todos los pares pudiendo pasar por los k primeros nodos.

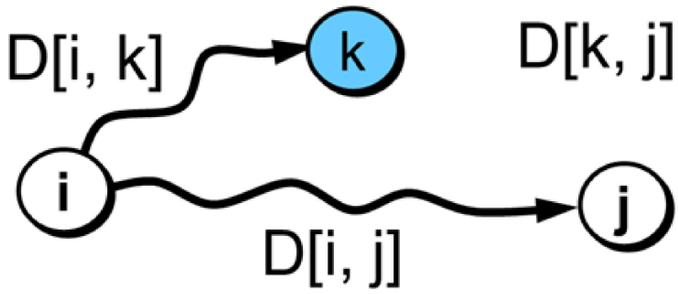
- **Inicialización**: **D** almacena los caminos directos.

- **Paso 1**: Caminos mínimos pudiendo pasar por el 1.

- ...

- **Paso n**: Caminos mínimos pudiendo pasar por cualquier nodo → Lo que buscamos.

- En el paso k, el nodo k actúa de pivote.



- Camino mínimo entre i y j, en el paso k:
  - Sin pasar por k: **D[i, j]**
  - Pasando por k: **D[i, k]** + **D[k, j]**
  - Nos quedamos con el menor.
- Ojo: Falta indicar cuáles son los caminos mínimos.
- **P**: array [1..n, 1..n] de entero. **P[i, j]** indica un nodo intermedio en el camino de i a j.

i → ... → **P[i, j]** → ... → j

Algoritmo de Floyd

```
D:= C
P:= 0
para k:= 1, ..., n hacer
    para i:= 1, ..., n hacer
        para j:= 1, ..., n hacer
            si D[i, k] + D[k, j] < D[i, j] entonces
                D[i, j]:= D[i, k] + D[k, j]
                P[i, j]:= k
            finsi
```

Analizar y responder la siguiente pregunta.

¿Cuánto es el orden de complejidad del algoritmo?

- El algoritmo de Floyd se basa en una descomposición recurrente del problema:

$$D_k(i,j) = \begin{cases} C(i,j) & \text{Si } k = 0 \\ \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)) & \text{Si } k > 0 \end{cases}$$

- Como la fila y columna k no cambian en el paso k, se usa una sola matriz D.
- ¿Cómo recuperar el camino?

```
operación camino (i, j: entero)
```

```
k:= P[i, j]
si k ≠ 0 entonces
    camino (i, k)
    escribe (k)
    camino (k, j)
finsi
```

Consultar el siguiente Link:

https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=499 <https://elibro.bibliotecabuap.elogim.com/es/ereader/bibliotecasbuap/50117?page=499>

3.5. Evaluación teórica.

Actividad desplegable

Coloque las palabras correctas en las siguientes afirmaciones:

a. Un grafo es una pareja G = (V, A), donde V representa un conjunto de puntos, llamados [ ] , y A es un conjunto de [ ] .
b. Un [ ] es un camino, es decir una sucesión de aristas adyacentes, donde no se recorre dos veces la misma arista, y donde se regresa al punto inicial.
c. Un grafo es [ ] si cada par de vértices está conectado por un camino.
d. Un grafo es [ ] si existe una arista entre cualquier par de vértices.
e. El algoritmo de [ ] , es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en el grafo.
f. El algoritmo de [ ] encuentra un subconjunto de aristas que forman un árbol con todos los pesos, donde la suma total de todas las aristas en el árbol es la mínima posible.
g. La manera más simple de representar un grafo es usando una [ ] de adyacencia.
h. Un dígrafo se dice que es [ ] si hay un camino entre cualquier par de vértices.
i. Un grafo se dice [ ] si tiene asociado un peso o valor en cada arista.
j. El grado de entrada y grado de salida se aplica a grafos [ ] .

3.6. Evaluación práctica

Actividad Práctica de Grafos

Para la realización de esta actividad leer cuidadosamente las indicaciones en el archivo de esta actividad

- Actividad_practica_Grafos.pdf (Ventana nueva)

Rúbrica para la evaluación de la actividad práctica

Rúbrica para evaluar el trabajo en equipo Aplicar

	3 Satisfactorio	2 Mejorable	1 Insuficiente
Participación y colaboración	Todos los miembros del equipo han participado activamente en las tareas propuestas y han colaborado ayudando a los demás. Presentan código fuente completo, documentado y con referencias correctas (4)	La mayor parte de los miembros del equipo han participado activamente en las tareas propuestas y han colaborado ayudando a los demás. Presentan código fuente completo sin documentar y sin referencias correctas. (3)	La mitad de los miembros del equipo ha participado activamente en las tareas propuestas y han colaborado ayudándose entre sí. No presentan código fuente completo, sin documentar, ni referencias correctas. (2)
Distribución de las tareas	Las tareas se han repartido de forma equitativa entre todos los miembros del equipo. Presentan documento pdf que contiene: descripción del algoritmo y la descripción de la ejecución del programa, con la documentación de las pantallas de ejecución completa. (4)	La mayor parte de las tareas se han repartido de forma equitativa entre todos los miembros del equipo. Presentan documento pdf que contiene: descripción del algoritmo y la descripción de la ejecución del programa, con la documentación de las pantallas de ejecución incorrecto y/o incompleto. (3)	Solo la mitad de las tareas se ha repartido de forma equitativa entre todos los miembros del equipo. No presentan documento pdf con descripción del algoritmo ni la descripción de la ejecución del programa. (2)

	3 Satisfactorio	2 Mejorable	1 Insuficiente
Integración ente los miembros del equipo	Durante la realización de todas las tareas, los miembros del equipo han expresado libremente sus opiniones y puntos de vista, han escuchado las opiniones de los demás y han sido capaces de llegar a un consenso. Presentan ejecución correcta de su programa en un link de Youtube (2)	Durante la realización de la mayor parte de las tareas, los miembros del equipo han expresado sus opiniones con libertad, han escuchado a los demás y han sido capaces de llegar a un consenso. Presentan ejecución incorrecta y/o incompleta de su programa en un link de Youtube. (1)	Durante la realización de las tareas, solo la mitad de los miembros del equipo ha expresado libremente sus opiniones, ha escuchado las de los demás y han logrado ponerse de acuerdo. No presentan ejecución de su programa en un link de Youtube. (0)

CEDEC <http://cedec.intef.es/> . *Rúbrica para evaluar el trabajo en equipo* (CC BY-SA <http://creativecommons.org/licenses/>)