

Algoritmos y Estructuras de Datos

Grafos No Dirigidos



Universidad
Católica del
Uruguay

Grafos No Dirigidos

- Los grafos son modelos naturales para representar relaciones entre objetos de datos.
- Un grafo consiste de un conjunto finito de vértices V y de un conjunto de arcos A .

$$G = (V, A).$$

- Sea el conjunto de vértices o nodos $V = \{v_1, v_2, \dots, v_n\}$ entonces el conjunto de arcos o aristas es $A = \{(v_i, v_j)\}$. un conjunto de pares de vértices.
- **Si las aristas son no dirigidas, es decir $(v_i, v_j) = (v_j, v_i)$, el grafo se llama no dirigido.**

Grafos No Dirigidos

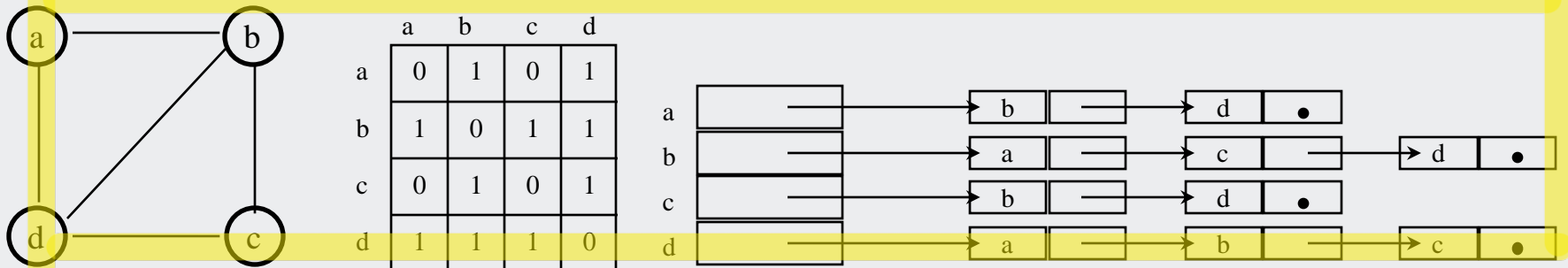
- Sea $G = (V, A)$ un grafo no dirigido, entonces si (v, w) pertenece a A , resulta $(v, w) = (w, v)$.
- Si (v, w) es una arista, se dice que es incidente sobre los vértices v y w , y los vértices son adyacentes entre sí.
- Un camino es una secuencia de vértices v_1, v_2, \dots, v_n , tal que (v_i, v_{i+1}) es una arista.
- Un grafo es conexo si todos sus pares de vértices están conectados.
- Un ciclo (simple) es un camino (simple) de longitud mayor o igual a tres (3), que conecta un vértice consigo mismo.

Árboles libres

- Un grafo no dirigido *conexo acíclico* se conoce también como **árbol libre**
- Un *árbol libre* puede convertirse en uno ordinario eligiendo un vértice como *raíz*.
- **Todo árbol libre con $n \geq 1$ vértices tiene exactamente $n-1$ aristas.**
- **Si se agrega cualquier arista a un árbol libre, resulta un ciclo.**

Grafos No Dirigidos

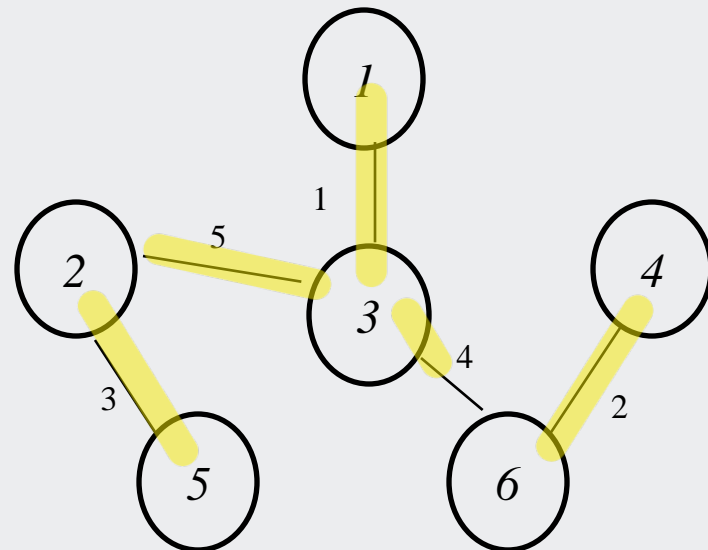
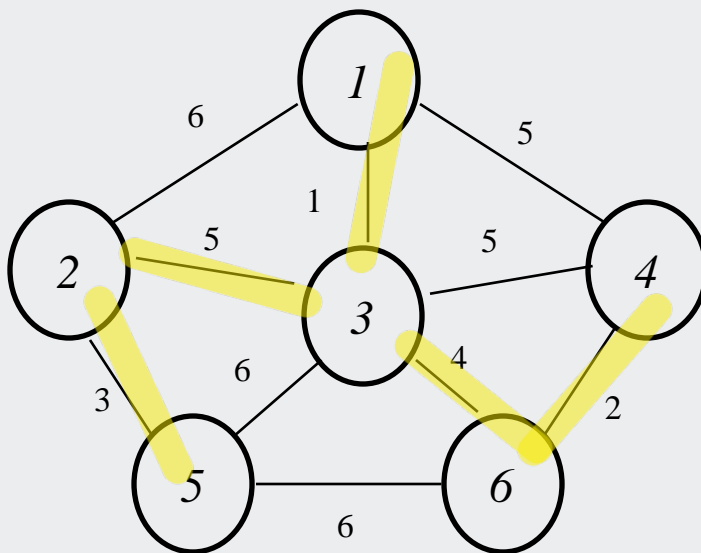
- Métodos de representación de grafos no dirigidos
 - Se pueden usar los mismos que para grafos dirigidos: **matrices** o **listas de adyacencias**.
 - **Una arista no dirigida entre v y w se representa mediante dos aristas dirigidas de v a w y de w a v .**
 - Notar que la **matriz de adyacencias es simétrica**.



ATENCIÓN: REVISAR EJERCICIO DOMICILIARIOS PD1- EJERCICIO 1, CÓDIGO IMPLEMENTACION DE GRAFO NO DIRIGIDO!

Árboles abarcadores de costo mínimo

- Dado un grafo $G = (V, A)$ donde cada arista (u,v) de A tiene un costo asociado $c(u,v)$:
 - Un **árbol abarcador** de G es un **árbol libre** que conecta **todos** los vértices de V .
 - El **costo** de ese árbol es la suma de los costos de todas las aristas.



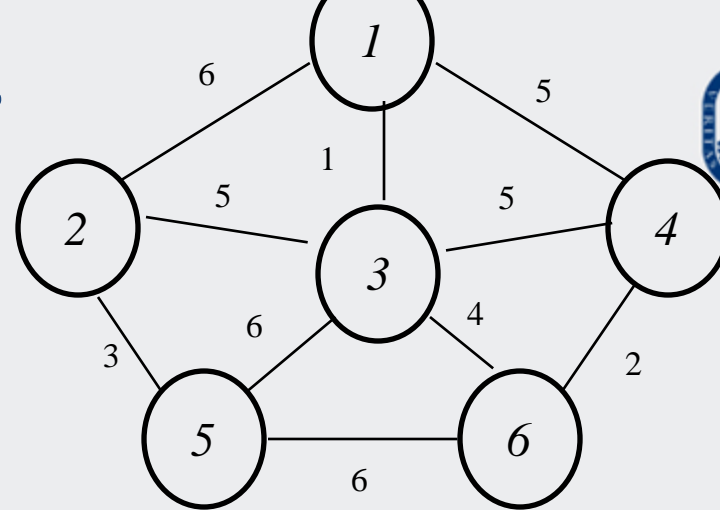
Propiedad AAM (Árbol Abarcador de costo Mínimo)

- Sea $G = (V, A)$ un grafo conexo con una función de costo definida para sus aristas. Sea U algún subconjunto propio del conjunto de vértices V .
 - Si (u,v) es una arista de **costo mínimo** tal que u pertenece a U y v pertenece a $V-U$, existe un AAM que incluye a (u,v) entre sus aristas.
- Dos algoritmos hacen uso de esta propiedad: Prim y Kruskal

Algoritmo de Prim

- $G = (V, A)$
- $V = \{1, 2, 3, \dots, n\}$ y una función de costo definida en las aristas de A
- El algoritmo de Prim comienza cuando se asigna a un conjunto U un vértice inicial $\{1\}$, en el cual el árbol abarcador “crece” arista por arista.
- En cada paso, localiza la arista más corta (u, v) que conecta U y $V - U$, y después agrega v , el vértice en V , a U .
- Este paso se repite hasta que $U = V$.

Algoritmo de Prim.



Método TGRAFO.Prim (conjunto de aristas T);

U: conjunto de vértices;

u, v: vértice;

// el TGRAFO representado por un conjunto de vértices **V** y un conjunto de Aristas **A**

COMIENZO

T. Vaciar;

U.Agregar (1);

MIENTRAS U <> V **hacer**

elegir una arista (u,v) de costo mínimo

tal que u está en U y v está en V-U;

T.agregar (u,v);

U.agregar(v);

FIN MIENTRAS

FIN;

Estructuras auxiliares para Prim

- ~~TArista~~
 - Origen, Destino : Comparable
 - Costo: numérico
- ~~TAristas // colección de elementos del tipo Tarista~~
 - buscar(origen, destino: TEtiqueta) : TArista
 - buscarMin(U,V: listas de vertices): Tarista
- ~~TGrafoNoDirigido~~
 - Agregar Aristas: TAristas
 - ~~Modificar el TDA Grafo y el constructor para también poner las aristas en “Aristas”~~
 - PRIM..... //devuelve un nuevo grafo, que es el AAM correspondiente

Ej. Implementación PRIM

Dada nuestra implementación de
TGrafoNoDirigido

- Constructor: recibe colección de Vértices y colección de Aristas
- TArista
 - Origen, Destino (comparable), costo
- TAristas // por ejemplo, podría heredar de LinkedList
 - buscar(origen, destino: comparable) : TArista
 - buscarMin(U,V: colecciones de vertices): TArista

- Dada una instancia ya creada del Grafo No Dirigido (vértices, aristas), ejecuta el algoritmo de Prim y devuelve ***un nuevo grafo, que es el AAM***
- **PRECONDICIONES???**
- Auxiliares:
 - Colección de vértices “**U**”, colección de vértices “**V**”
 - **V** contiene las etiquetas de los vértices del grafo original
 - “**AristasAAM**” del tipo “**TAristas**” en donde se irán agregando las aristas de costo mínimo
 - tempArista de tipo TArista para ir llevando la mín.

TGrafoNoDirigido.Prim(): TGrafoNoDirigido

U.agregar(V.quitarprimero)

Mientras V no vacío hacer

tempArista <- aristas.**buscarMin**(U,V)

aristasAAM.Insertar(tempArista)

V.quitar(tempArista.etiquetaDestino)

U.agregar(tempArista.etiquetaDestino)

costoPrim <- costoPrim + tempArista.costo

Fin mientras

Devolver nuevo TGrafoNoDirigido(U, AristasAAM)

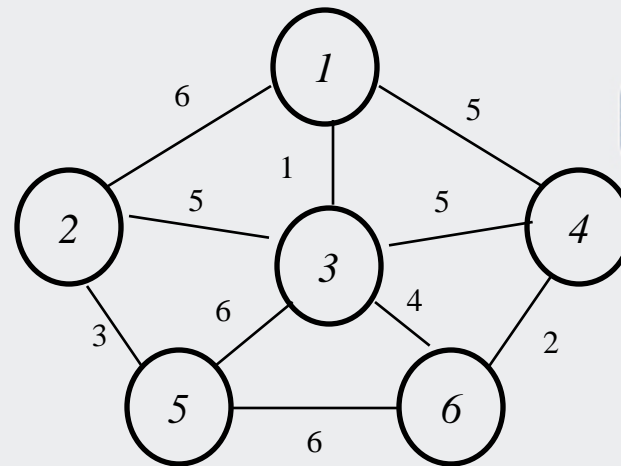
TAristas extends LinkedList

- buscar(origen, destino: comparable) : TArista
 - // ¿cómo sería?
- **buscarMin**(U,V: colecciones de vertices): TArista
 - TArista minArista = null, int minCosto = maxInt
 - Para cada **u** en **U**
 - Para cada **v** en **V**
 - tempArista <- buscar(u,v)
 - Si tempArista <> null
 - si tempArista.costo < minCosto
 - minArista <- tempArista
 - minCosto <- tempArista.costo
 - Fin Si
 - **Devolver minArista**

Algoritmo de Kruskal

- $G = (V, A)$, $V = \{1, 2, 3, \dots, n\}$ y una función de costo definida en las aristas de A
- Se empieza con un grafo $T = (V,)$, constituido sólo por los vértices de G y sin aristas. Cada vértice es un componente conexo en sí mismo.
- Al avanzar, habrá siempre una colección de componentes conexos
- Para cada componente se seleccionarán las aristas que formen un árbol abarcador.
- Para construir componentes cada vez mayores, se agrega la arista de costo mínimo que conecte dos componentes distintos.
- La arista se descarta si conecta dos vértices que están en el mismo componente conexo, pues crearía un ciclo.
- Cuando todos los vértices están en un sólo componente, T es un árbol abarcador de costo mínimo para G .

Algoritmo de Kruskal



Método TGrafo.Kruskal;

F conjunto de aristas;

COM

F.Vaciar;

Repetir

Elegir una arista de costo mínimo tal que no esté en F ni haya sido elegida;

*Si la arista **no** conecta dos vértices del mismo componente entonces agregarla a F;*

hasta que *todos los vértices estén en un solo componente;*

FIN

Recorridos de grafos no dirigidos

- Visitar sistemáticamente todos los vértices del grafo.
- Existen dos técnicas: búsqueda en profundidad y búsqueda en amplitud.
- Ejemplo de aplicación: determinar eficientemente si todos los vértices están conectados a un vértice dado.

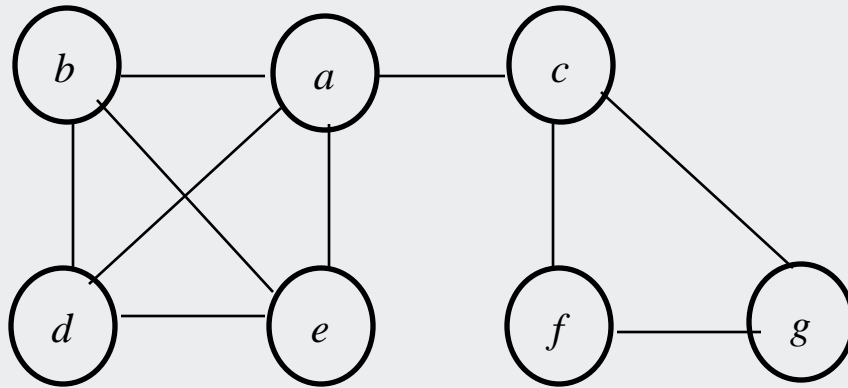
Búsqueda en profundidad.

- Se puede emplear el mismo algoritmo definido para grafos dirigidos.
- En este caso, si el grafo es conexo, de la búsqueda en profundidad se obtiene un sólo árbol.
- Para grafos no dirigidos, hay dos clases de arcos: de árbol y de retroceso.

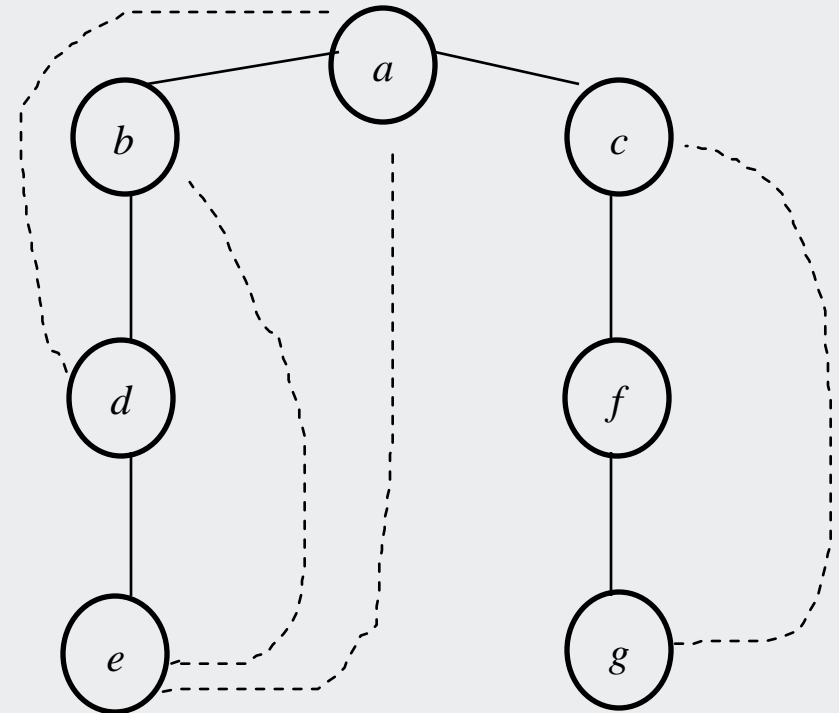
Búsqueda en amplitud.

- Se denomina “en amplitud” porque, desde cada vértice v se visitan todos los adyacentes, para luego visitar los descendientes.
- Al realizar una búsqueda en amplitud también se puede construir un bosque abarcador.
- Si el grafo no es conexo, la búsqueda en amplitud debe realizarse a partir de un vértice de cada componente.

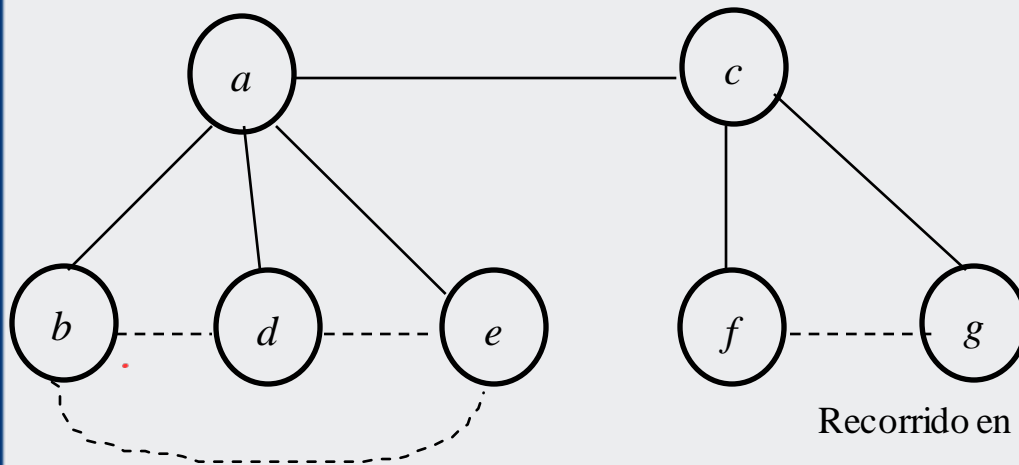
Búsqueda en Amplitud y Profundidad



Grafo G



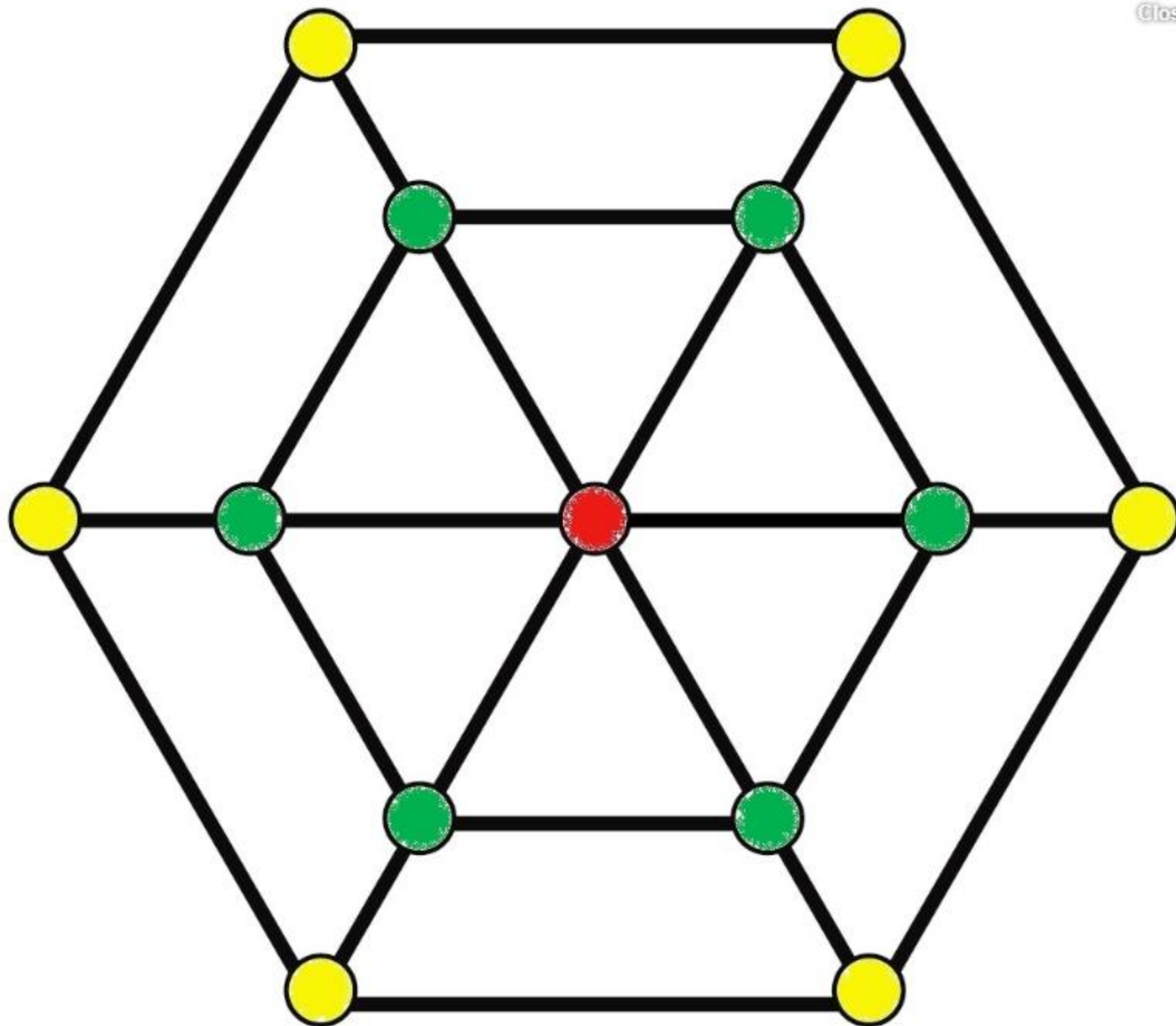
Recorrido en profundidad



Recorrido en amplitud

Algoritmo de búsqueda en amplitud.

- Método Tvertice **bea** : String
- //{bea visita todos los vértices conectados a **v** usando búsqueda en amplitud.}
- C: ColaDeVértices;
- x,y : Vértice;
- tempstr : String
- **COM**
- Visitar();
- C.Insertar (this);
- tempstr <- tempstr + etiqueta
- **mientras no** vacía(C) **hacer**
- x ← C.eliminar () // en x queda el elemento frente de la cola
- **para** cada vértice **y** adyacente a **x** **hacer**
- **Si no** y.Visitado() **entonces**
- y.Visitar();
- C.insertar (y);
- tempstr <- tempstr + y.etiqueta
- **fin si**
- **fin para cada;**
- **fin mientras;**
- Devolver tempstr
- **FIN;** {bea}

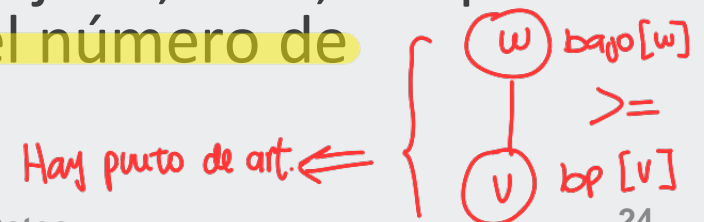


Puntos de articulación y componentes biconexos

- **Punto de articulación:** vértice v tal que, cuando se elimina, junto con todas las aristas incidentes sobre él, se divide un componente conexo en dos o más partes.
- A un grafo sin puntos de articulación se le llama “**grafo biconexo**”.
- Un grafo tiene **conectividad k** si la eliminación de $k-1$ vértices cualesquiera no lo desconecta.
- La búsqueda en profundidad es muy útil para encontrar los componentes biconexos de un grafo.

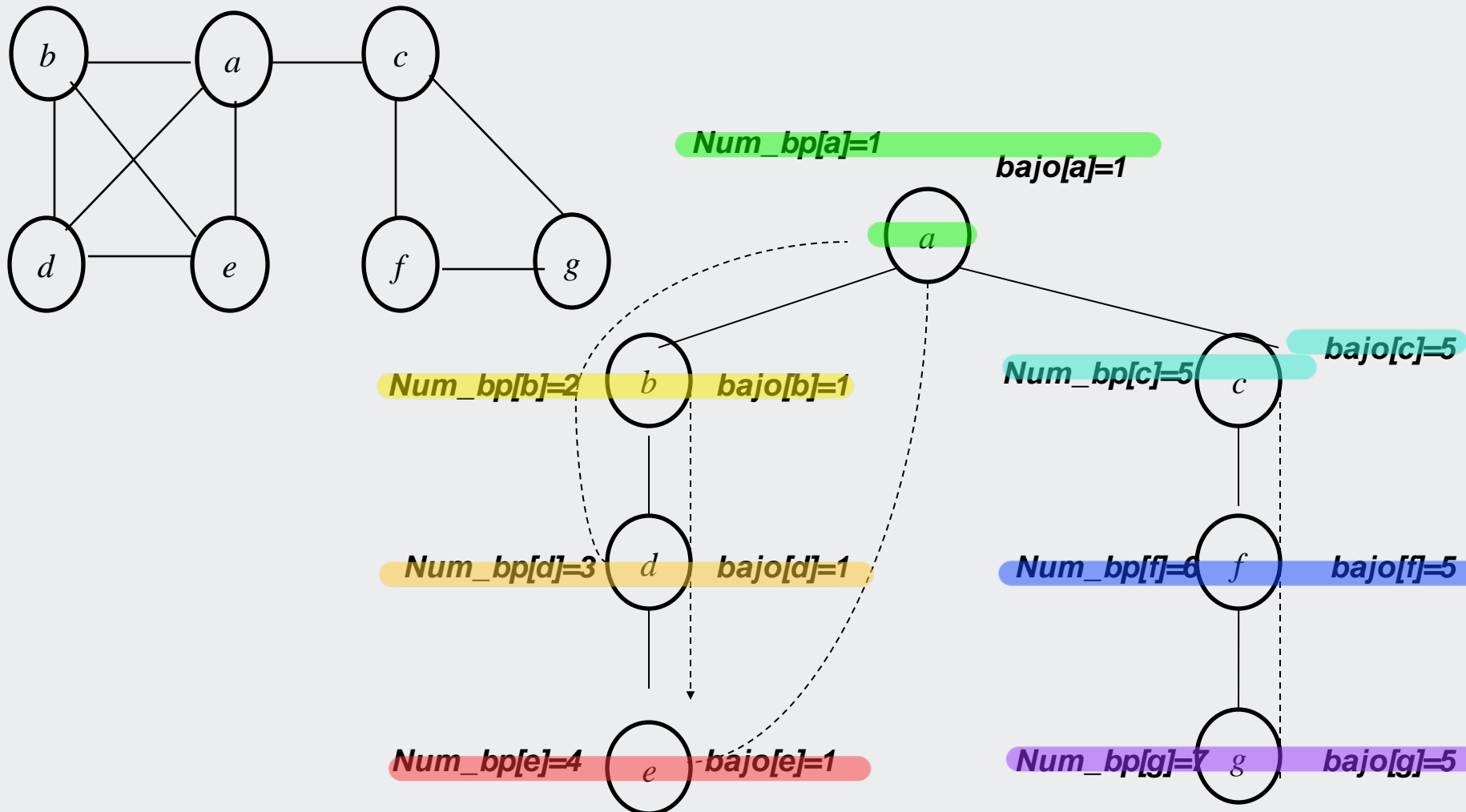
Algoritmo para encontrar puntos de articulación

- Realizar búsqueda en profundidad numerando en **orden previo** los vértices (obtener **número_bp[v]**).
- Por cada vértice **v** obtener **bajo[v]**.
- Los puntos de articulación se encuentran como sigue:
 - la raíz es un punto de articulación si y sólo si tiene **dos o más** hijos.
 - un vértice **v** distinto de la raíz es un punto de articulación si y sólo si hay un hijo **w**, de **v**, tal que el **bajo[w]** es mayor o igual que el número de **número_bp[v]**.



- El número **bajo** de un vértice v es el número más pequeño de ese nodo v o de cualquier otro w accesible desde él, siguiendo cero o más aristas de árbol hasta un descendiente x de v (x puede ser v), y después seguir una arista de retroceso (x, w) .
- Se calcula **bajo**(v) para todos los vértices v visitándolos en un recorrido en orden posterior. Cuando se procesa v , se ha calculado **bajo**(y) para todo hijo y de v . Entonces se toma **bajo**(v) como el mínimo de :
 - 1. **número_bp** de v .
 - 2. **número_bp** de z para cualquier vértice z para el cual haya una arista de retroceso (v, z) .
 - 3. **bajo**(y) para cualquier hijo y de v .

Puntos de articulación

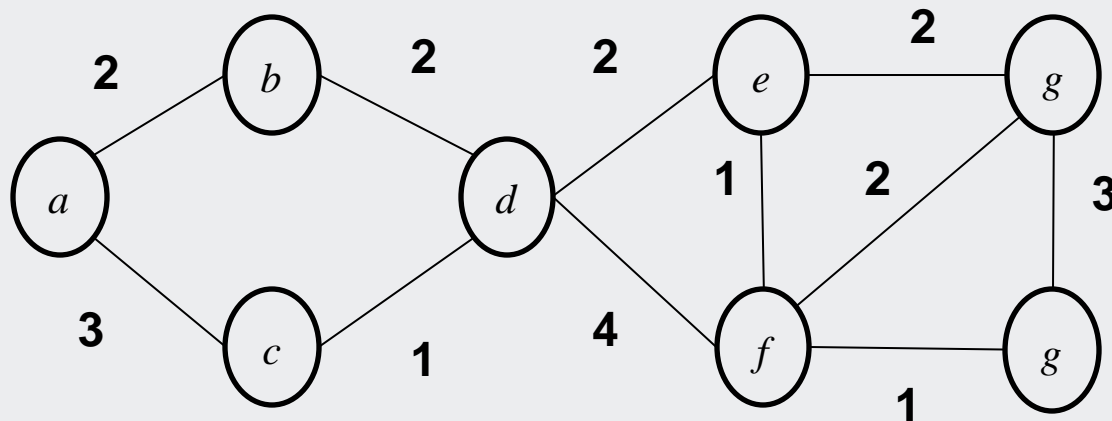


Ejercicios grafos no dirigidos

- Define árbol abarcador de costo mínimo
- Describe el algoritmo de Kruskal.
- Describe el algoritmo de Prim
- Define búsqueda en amplitud y desarrolle el algoritmo correspondiente
- ¿qué son los puntos de articulación de un grafo no dirigido?
- ¿qué es un grafo biconexo?
- Define conectividad de un grafo y escriba un algoritmo para identificarla
- Indica posibles aplicaciones prácticas de hallar los árboles abarcadores de costo mínimo de un grafo, y los puntos de articulación del mismo

Ejercicios de grafos

- Dado el grafo de la figura, encuentra:
 - Un árbol abarcador de costo mínimo usando el algoritmo de Prim
 - Un árbol abarcador de costo mínimo usando el algoritmo de Prim
 - Un árbol abarcador en profundidad empezando en los vértices ***a*** y ***d***
 - Un árbol abarcador en amplitud empezando en los vértices ***a*** y ***d***



Ejercicios de grafos no dirigidos

- TDA GRAFO NO DIRIGIDO
 - Describe en lenguaje natural un método para hallar puntos de articulación en un grafo no dirigido.
 - Escribe en pseudocódigo un algoritmo que implemente el método descrito.
 - Demuestra la ejecución con un ejemplo

Ejercicios de grafos

- Dado el siguiente grafo NO DIRIGIDO, halla el ÁRBOL ABARCADOR DE COSTO MÍNIMO mediante el algoritmo de KRUSKAL, mostrando el orden en que fueron elegidas las aristas, y cómo va quedando el grafo en cada iteración.
- Idem, utilizando el algoritmo de PRIM

