Gráficas

comp-420

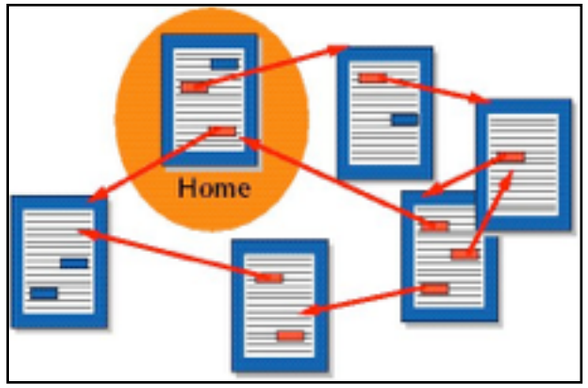
Gráficas

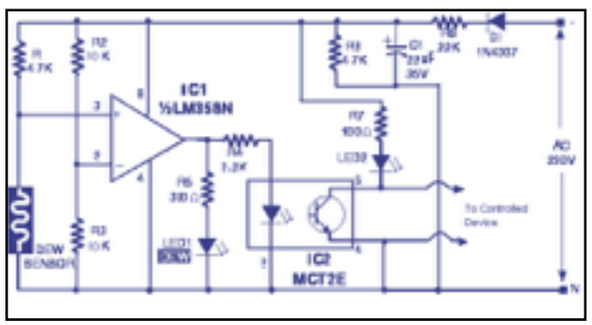
- Muchas aplicaciones computacionales involucran no solo un conjunto de elementos sino que también conexiones entre pares de elementos.

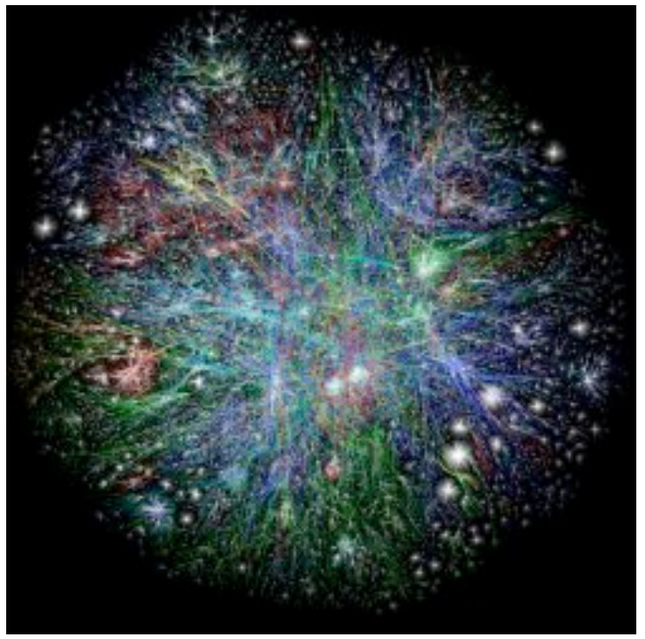
- Las relaciones que resultan de estas conexiones nos llevan a preguntas como:
  - ¿Hay forma de llegar de un elemento a otro siguiendo las conexiones?
  - ¿Cuáles y cuántos elementos se pueden alcanzar a partir de un elemento dado?
  - ¿Cuál es la mejor forma de llegar de un elemento a otro?

- Para modelar situaciones como esta se utilizan las gráficas.

Aplicaciones







- mapas
- hypertexto
- modelación mecánica
- planificación de tareas
- transacciones
- emparejamiento
- redes informáticas, internet
- estructura de programas...

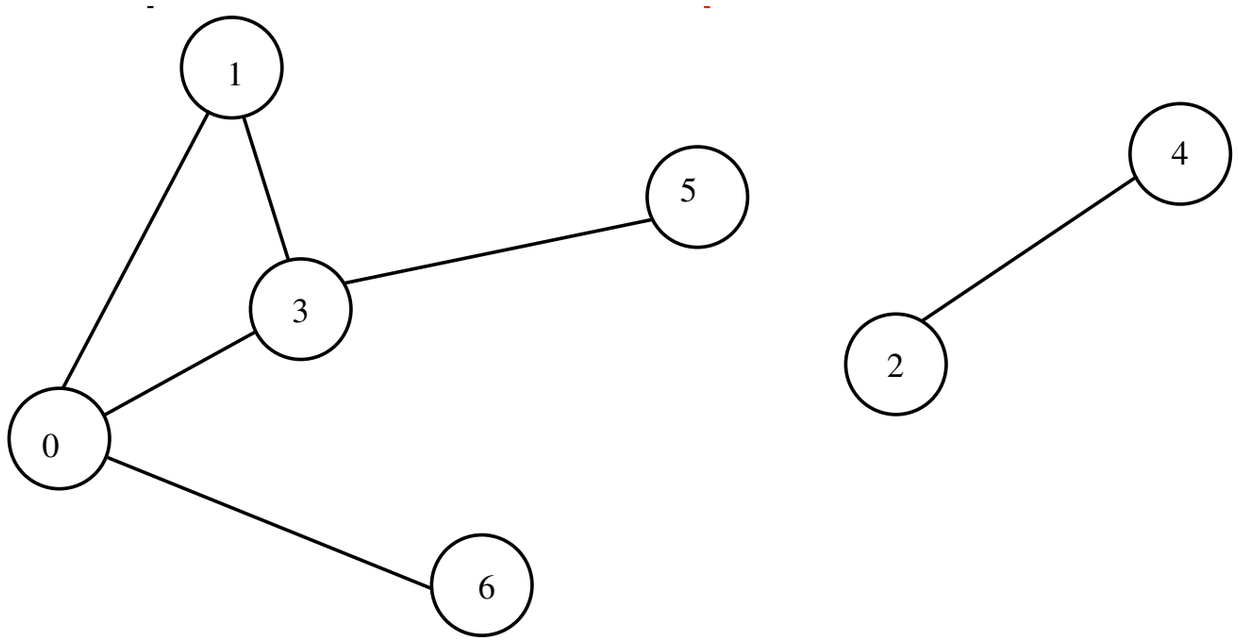
Gráficas

- El estudio del desempeño en algoritmos con gráficas es particularmente retador porque:
  - El costo de un algoritmo depende no solo de las propiedades del conjunto de elementos sino también de las propiedades del conjunto de conexiones.
  - Es difícil hacer modelos precisos de problemas por gráficas.

Gráficas: definiciones y propiedades

- Una gráfica *G* es un par de conjuntos $G = (V, E)$.
- *V* es un conjunto de *n* objetos arbitrarios $u, v, w \in V$ llamados vértices o nodos.
- *E* es un conjunto de aristas, ejes o arcos *m*.
- Las aristas son típicamente pares de vértices, $\{u, v\} \in E$ definiendo una relación entre el conjunto *V* con si mismo: $E \subseteq V \times V$.
- En una gráfica no dirigida los ejes son pares no ordenados o solo conjuntos que contienen dos vértices.
- En una gráfica dirigida o digráfica los ejes son pares ordenados de vértices (están dirigidos).

Gráficas: definiciones y propiedades



- Vértices: 0,1,2,3,4,5,6
- Aristas:

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

Gráficas: definiciones y propiedades

- Un bucle es una arista reflexiva, donde coinciden el vértice de origen y el vértice de destino: $\{v, v\}$ o $< v, v >$.
- En el caso general, puede ser que haya más de una arista por un par de vértices. En tal caso, la gráfica se llama multigráfica.
- Nos ocuparemos de gráficas simples, donde no hay ejes de un vértice hacia si mismo y hay a lo más un eje de un vértice a cualquier otro.
- Si se asignan pesos o costos a las aristas, la gráfica es ponderada.
- Si se asigna identidad a los vértices o a las aristas, la gráfica es etiquetada.

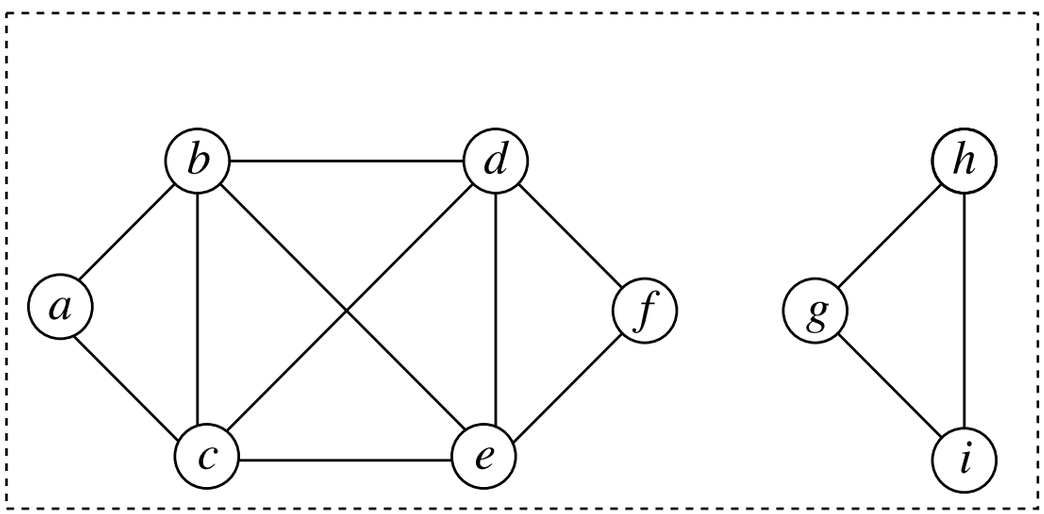
Gráficas: definiciones y propiedades

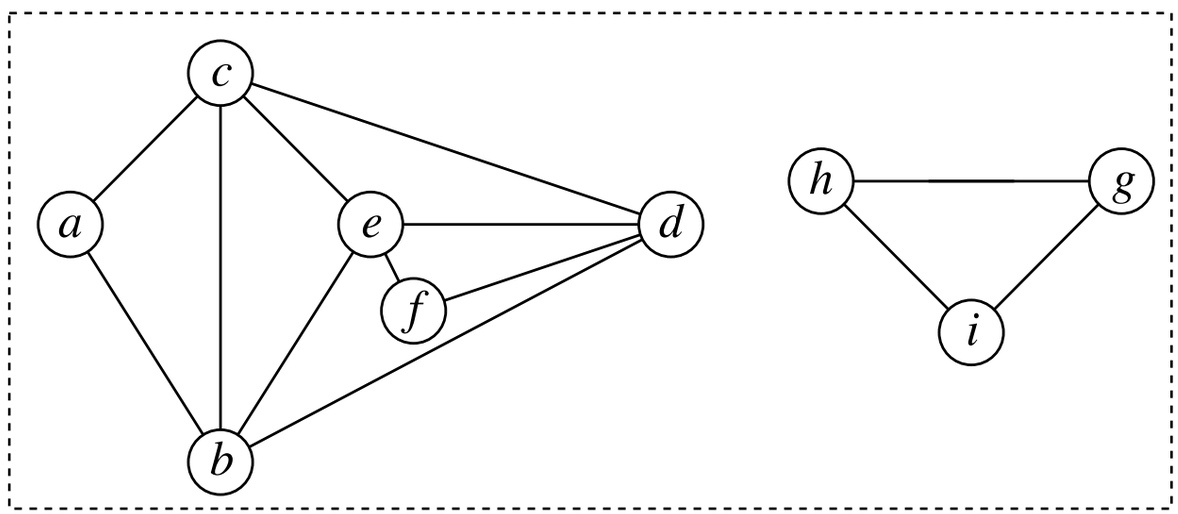
- Usaremos *V* para denotar el número de vértices en una gráfica y *E* para denotar el número de ejes.
- En una gráfica no-dirigida tenemos: $0 \leq E \leq \binom{V}{2}$.
- En una gráfica dirigida tenemos: $0 \leq E \leq V(V-1)$.
- Podemos visualizar las gráficas al mirar una inmersión. La inmersión de una gráfica transforma cada vértice a un punto en el plano y cada eje a una curva o un segmento de recta entre dos vértices.

Gráficas: definiciones y propiedades

- Una gráfica es plana si se puede dibujar en dos dimensiones de tal manera que ninguna arista cruce a otra arista.

- La misma gráfica puede tener varias inmersiones (dibujos) por lo que es importante no confundir la inmersión con la gráfica misma. En particular, una gráfica plana puede tener inmersiones no planas.





Gráficas: definiciones y propiedades

- Dos aristas $\{v_1, v_2\}$ y $\{w_1, w_2\}$ son adyacentes si tienen un vértice común.
- Una arista es incidente a un vértice si ésta lo une al vértice.
- Dos vértices v y w son adyacentes si una arista los une: $\{v, w\} \in E$.
- El grado $\delta(v)$ de un nodo $v \in V$ es el número de aristas de E que inciden en v (el número de vecinos).
- El grado $\Delta(G)$ de una gráfica es el máximo de los grados de sus vértices.

Caminos en gráficas

- Un camino de tamaño n en una gráfica $G=(V,E,\Phi)$ es una secuencia a $n+1$ vértices u_i, para $0 \leq i \leq n$, tales que para todo $1 \leq i \leq n$, existe $e \in E$ tal que $\Phi(e) = (u_{i-1},u_i)$ o $\Phi(e)=(u_i,u_{i-1})$.

- Dicho informalmente, todos los vértices después del primero son adyacentes a su predecesor.

- Camino simple: todos los vértices y las aristas son distintos.

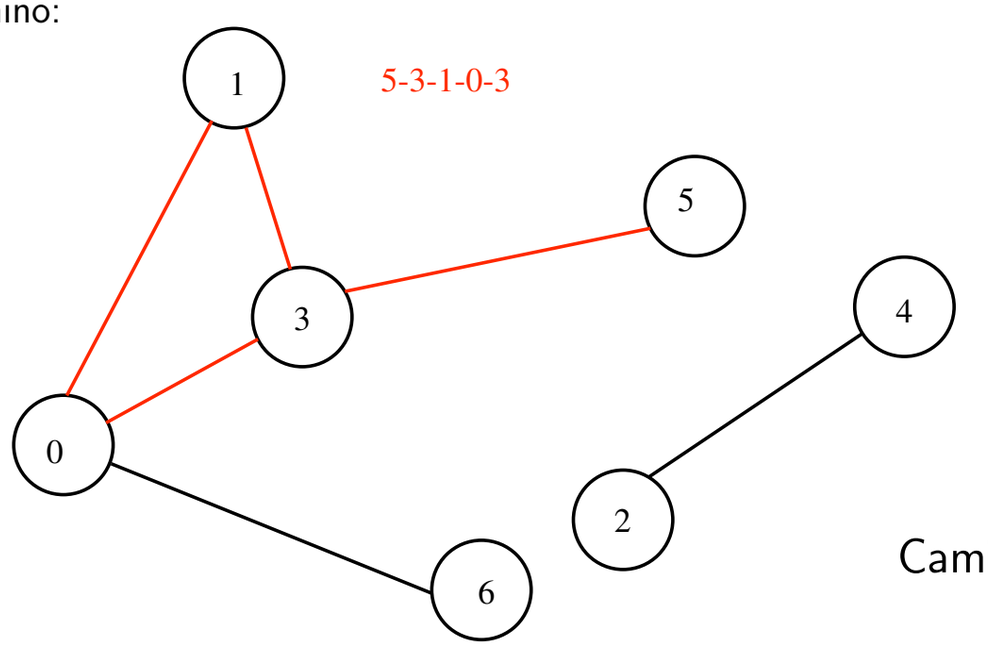
- Cíclo: un camino simple excepto que el primer y el último vértice son iguales.

- Camino cíclico: un camino tal que el primer y el último vértices son iguales.

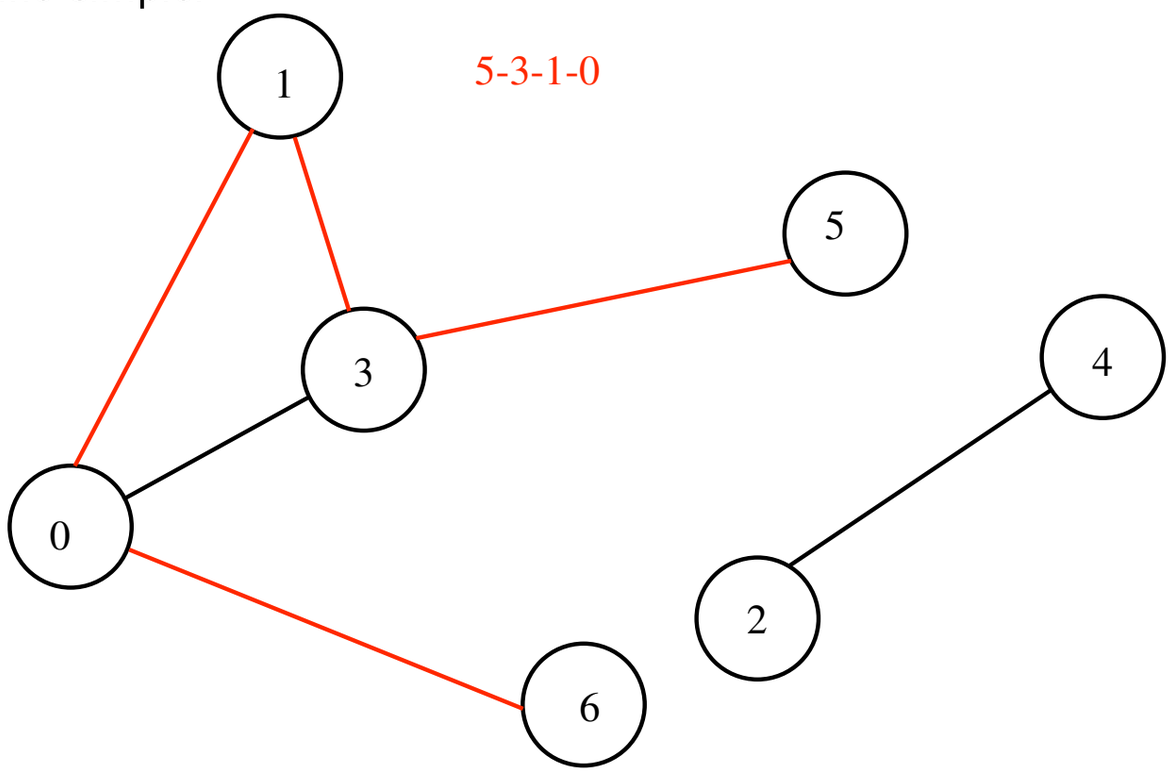
- Tour: camino cíclico que pasa por todos los vértices.

Caminos en gráficas

Camino:

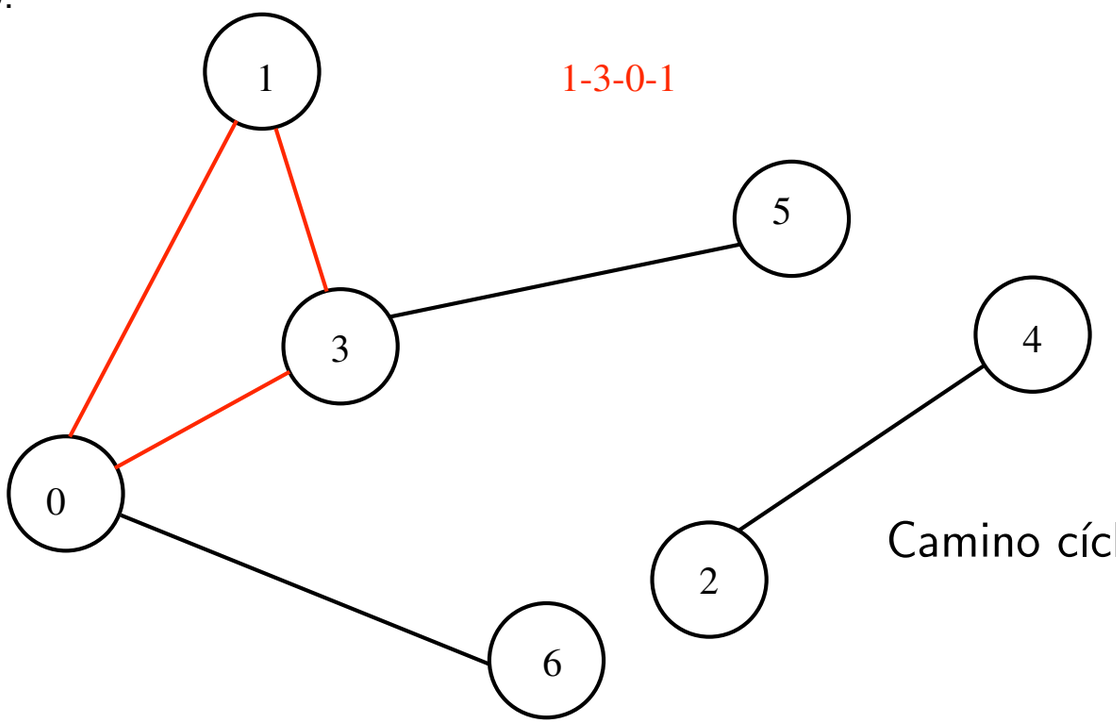


Camino simple:

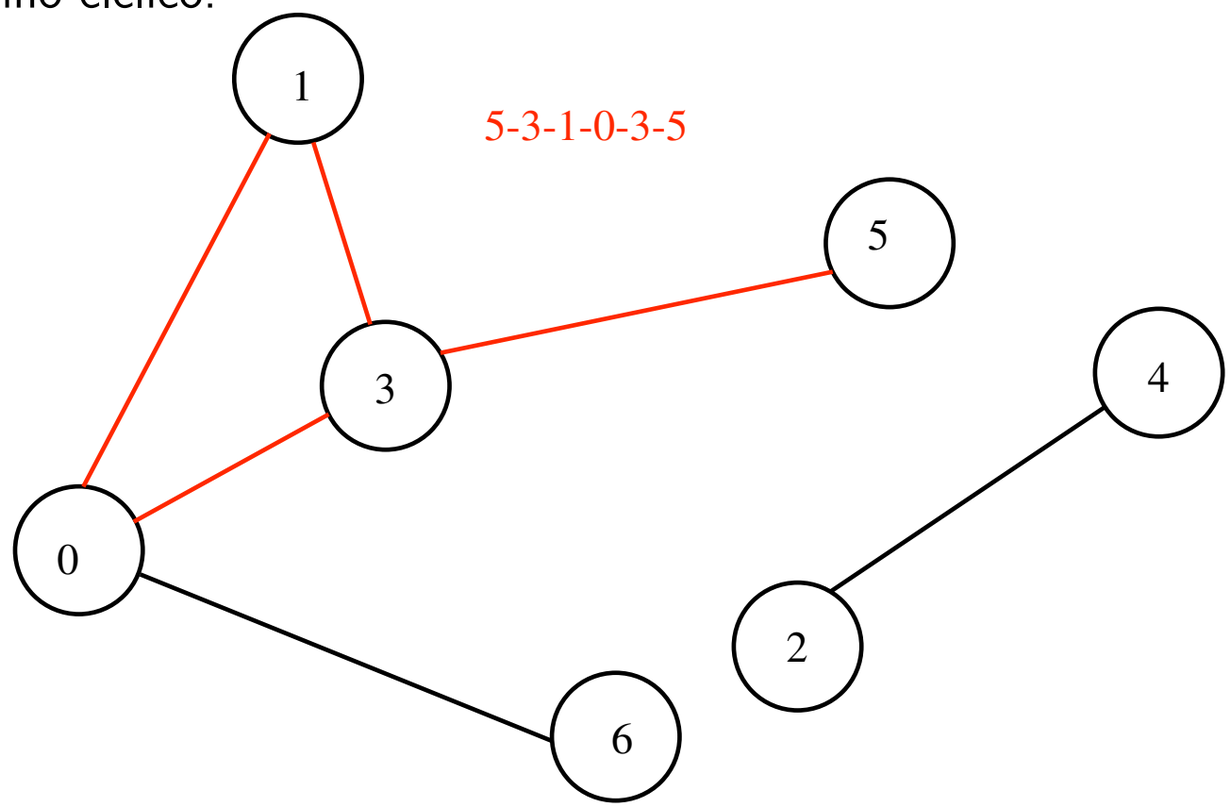


Caminos en gráficas

Ciclo:



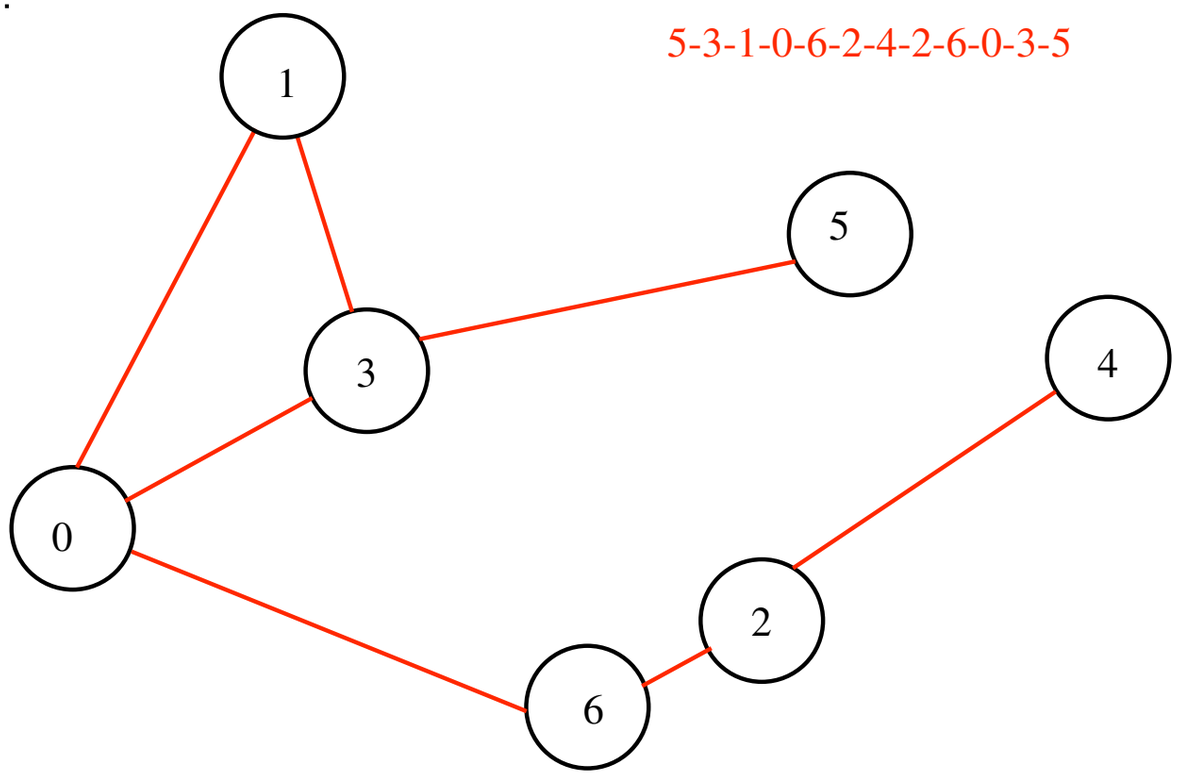
Camino cíclico:



Caminos en gráficas

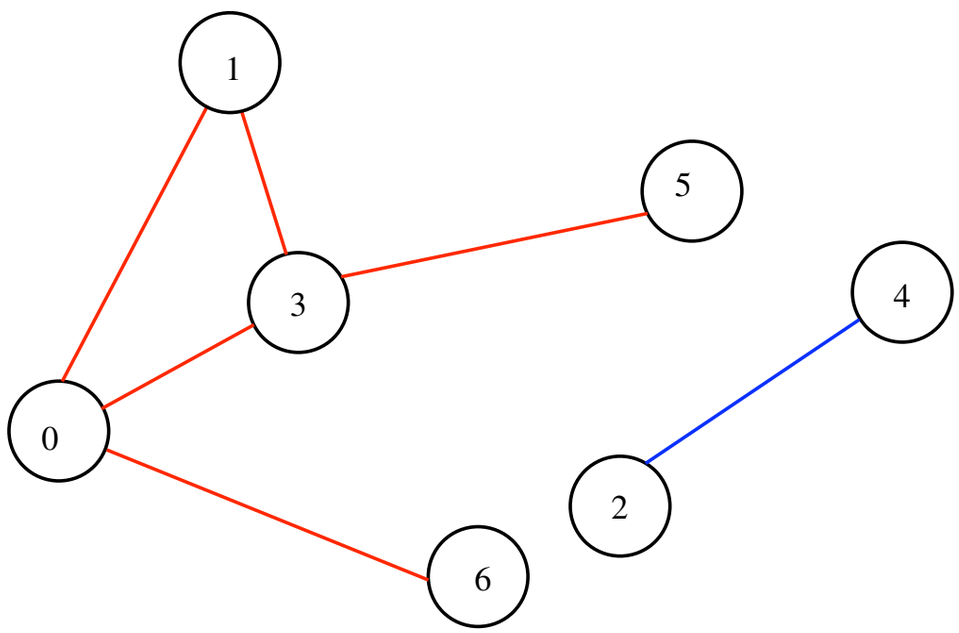
Tour:

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



Conectividad

- Una gráfica está conectada si existe un camino de cualquier vértice de V a cualquier otro vértice de V.
- Una gráfica desconectada consta de varios componentes conectados, que son subgráficas conectadas.
- Dos vértices están en el mismo componente conectado si y solo si existe un camino entre ellos.



Árboles

- Un ciclo es un camino que empieza y termina en el mismo vértice y tiene al menos un eje.
- Una gráfica es acíclica o un bosque si ningúna subgráfica es un ciclo.
- Los árboles son gráficas especiales que pueden definirse de diferentes formas equivalentes:
  - Un árbol es una gráfica acíclica conectada.
  - Un árbol es un componente conectado de un bosque.
  - Un árbol es una gráfica conectada con a lo más $V-1$ ejes.
  - Un árbol es una gráfica mínimamente conectada; eliminar cualquier eje desconecta al gráfico.

Árboles

- Un árbol es una gráfica acíclica con al menos V-*1* ejes.
- Un árbol es una gráfica acíclica maxima; añadir un eje entre cualquier par de vértices crea un ciclo.

- Un árbol generador (spanning tree) de una gráfica G es una subgráfica que es un árbol y que contiene cada vértice de G.
- Un bosque generador es un una colección de árboles generadores, uno por cada componente conectado de G.

Representación de gráficas

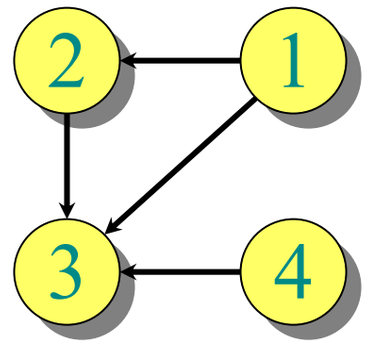
- Aquí se presentan cuatro métodos comunes de representar gráficas.
- Si la gráfica es grande (e.g. infinita), la estructura no se conoce a priori, etc. podemos elegir una representación implícita de la gráfica, donde los ejes y vértices se calculen en línea cuando se necesiten.
- Una representacion explícita es una donde codificamos directamente la estructura de la gráfica en una estructura de datos.

Representación explícita de gráficas

- Hay dos estructuras de datos comunes para la representación explícita de gráficas: las matrices de adyacencia y las listas de adyacencia.
- La matriz de adyacencia de una gráfica $G = (V, E)$ donde $V = \{1, 2, \ldots, n\}$ es la matriz $A[1 \ldots n, 1 \ldots n]$ dada por:

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

A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

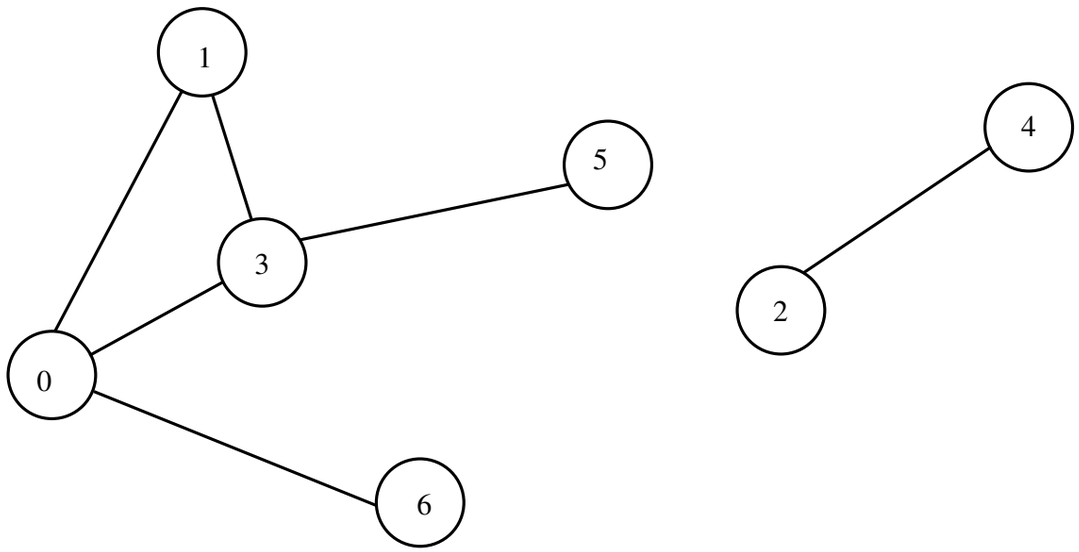


- almacenamiento?

  $\Theta(V^2)$

- representación densa.

Matrices de adyacencia



0	1	0	1	0	0	1
1	0	0	1	0	0	0
0	0	0	0	1	0	0
1	1	0	0	0	1	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
1	0	0	0	0	0	0

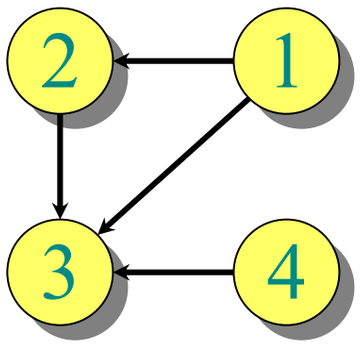
Matrices de adyacencia

$$|V| \times |V|$$

- Un arreglo `Y` tal que `Y[i][j]=1` si `i` y `j` están conectados, cero si no.
  - Matriz de tamaño .
  - En el caso de gráficas no orientados, la matriz Y es simétrica.
  - Representación adecuada para gráficas densas, no adecuada si no (matriz rala)
  - ¿Cuánto tiempo toma verificar si dos vértices están conectados por un eje?
    - ⍬(1) solo verificando la celda indicada de la matriz.
  - ¿Cuánto tiempo toma listar todos los vecinos de un vértice?
    - ⍬(V) para revisar el renglón correspondiente.

Listas de adyacencia

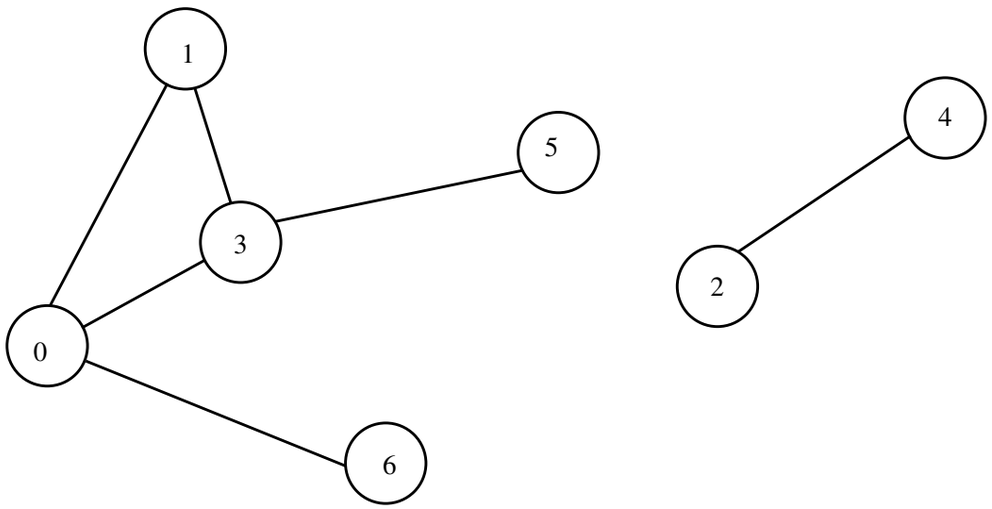
- Una lista de adyacencia de un vértice $v \in V$ es la lista $Adj[v]$ de vértices adyacentes a v.



$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$

- Para gráficas no dirigidas, $|Adj[v]| = degree(v)$.
- Para digráficas, $|Adj[v]| = out\text{-}degree(v)$.

Listas de adyacencia



- 0: $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 6
- 1: $\rightarrow$ 0 $\rightarrow$ 3
- 2: $\rightarrow$ 4
- 3: $\rightarrow$ 0 $\rightarrow$ 1 $\rightarrow$ 5
- 4: $\rightarrow$ 2
- 5: $\rightarrow$ 3
- 6: $\rightarrow$ 0

Listas de adyacencia

- En gráficas no dirigidos, cada eje (u,v) se almacena dos veces, una vez en la lista de vecinos de u y otra en la lista de vecinos de v.
- Para gráficas dirigidas cada eje se almacena una sola vez.
- De cualquier manera el espacio requerido para una lista de adyacencia es O(V+E).
- Listar a los vecinos de un nodo v toma O(1+deg(v)) tiempo.
- ¿A qué otra estructura de datos les recuerda?
  - hashing con encadenamiento.

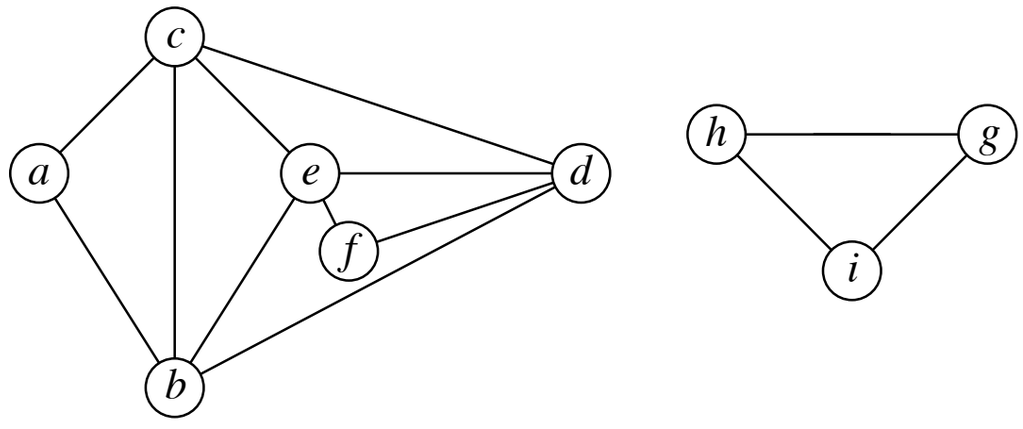
Listas de adyacencia

- Si la estructura de la gráfica es estática (es decir, que no cambia mientras el algoritmo se ejecuta), es común representar las listas de ejes entrantes y salientes como vectores, por eficiencia.
- Para algoritmos más elaborados como gráficas con pesos, se utiliza frecuentemente esta representación.

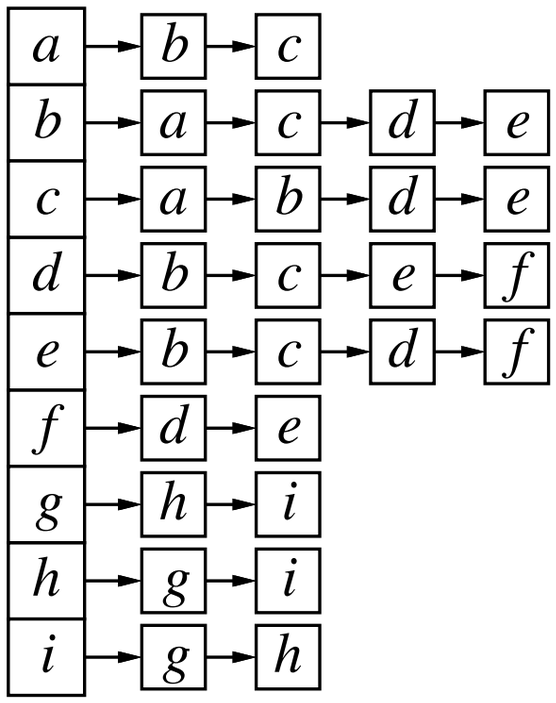
```
public class Edge {
  Vertex x, y;
  double weight;
}

public class Vertex {
  Set<Edge> out;
  Set<Edge> in;
}
```

Representaciones explícitas de grafos



	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



Representación de gráficas

- Dos maneras estándar de representar una gráfica G=(V,E):
  - listas de adyacencia
  - matrices de adyacencia

- Aplicables a gráficas dirigidas y gráficas no dirigidas.

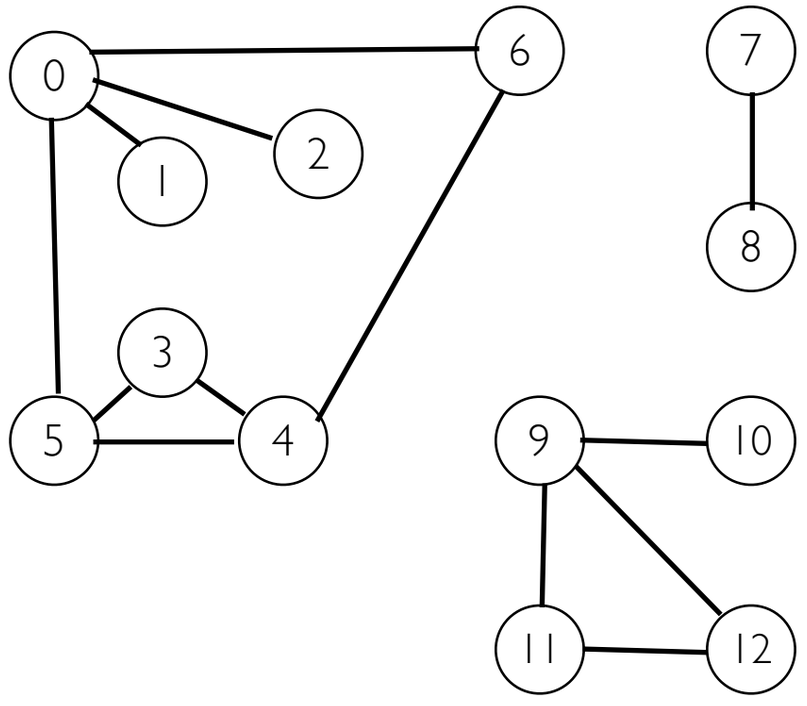
- La representación con listas de adyacencia se prefiere cuando las gráficas son:
  - ralas, es decir...
  - $|E| << |V^2|$.

- La representación con matrices de adyacencia se prefiere cuando las gráficas son:
  - densas, es decir...
  - $|E| \sim |V^2|$ o cuando queremos hacer qué operación?
  - encontrar conectividad entre nodos.

- En general utilizaremos listas de adyacencia a menos que se especifique lo contrario.

Implementación del ADT con matrices de adyacencia

0	0	1	1	0	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

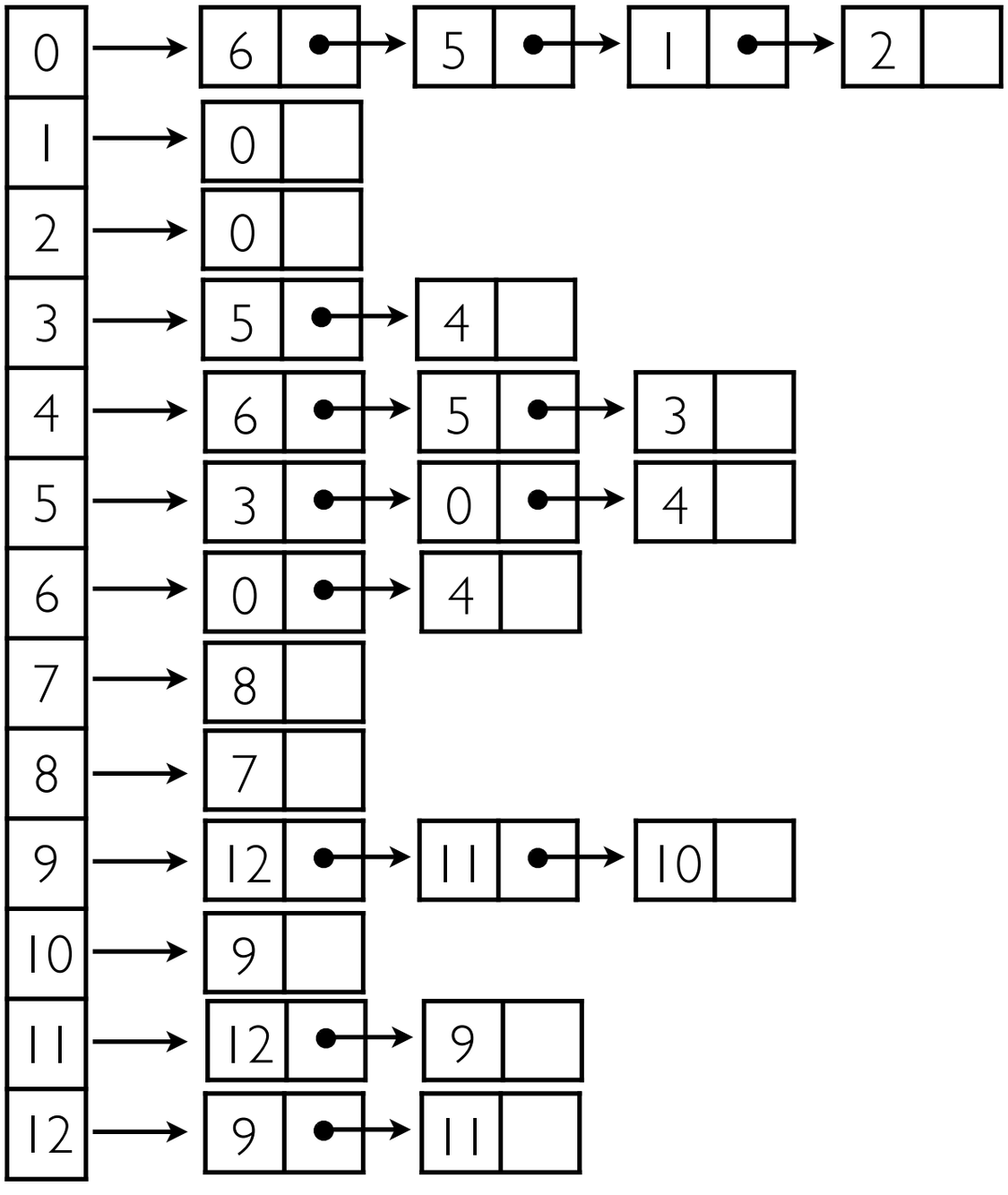


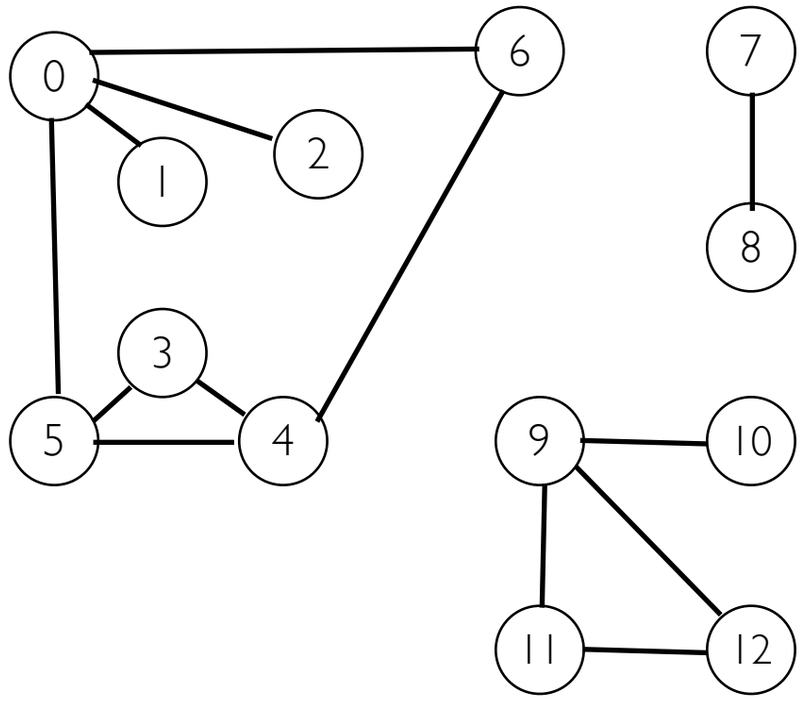
Implementación del ADT con matrices de adyacencia

- Procesar todos los vértices adyacentes a un vértice dado requiere al menos de un tiempo:
  - ➡ proporcional a V.
- Esta interfaz requiere conocer el número de vértices V al inicio.
- Inicializar todos los campos de la matriz al valor false toma un tiempo:
  - ➡ proporcional a V^2.
- Esta representación:
  - ➡ ¿permite aristas paralelas?
    no
  - ➡ ¿permite self-loops?
    si

- La representación con matriz de adyacencia no es satisfactoria para gráficas ralas grandes, necesitamos al menos V^2 bits para almacenamiento y V^2 pasos para construcción.

Implementación del ADT con listas de adyacencia





Implementación del ADT con listas de adyacencia

- Es la representación más utilizada para gráficas que no son densas.
- Una arista se puede agregar en tiempo constante.
- El espacio total es proporcional al número de vértices más el número de aristas.
- Al usar la representación de listas ligadas hay que utilizar el contenedor de la STL o recordar incluir un destructor y un constructor por copia.
- Esta representación ...
  - ¿permite aristas paralelas? si
  - ¿permite self-loops? si
- La principal desventaja es que requiere un tiempo proporcional a V para determinar la existencia de una arista.

Costos en el peor caso para diferentes implementaciones

	arreglo de aristas	matriz de adyacencia	listas de adyacencia
espacio	E	V^2	$V+E$
inicialización	1	V^2	V
copia	E	V^2	$E+V$
destrucción	1	V	E
insertar arista	1	1	1
eliminar arista	E	1	V
¿nodo aislado?	E	V	1
¿camino de u a v?	$E \lg V$	V^2	$V+E$

Recorridos de Gráficas

comp-420

Algoritmos de búsqueda en gráficas

- Una de las operaciones más frecuentes en una gráfica es la de visitar sus vértices uno por uno o en un orden deseado.

- Este procedimiento se llama búsqueda o recorrido de un grafo.

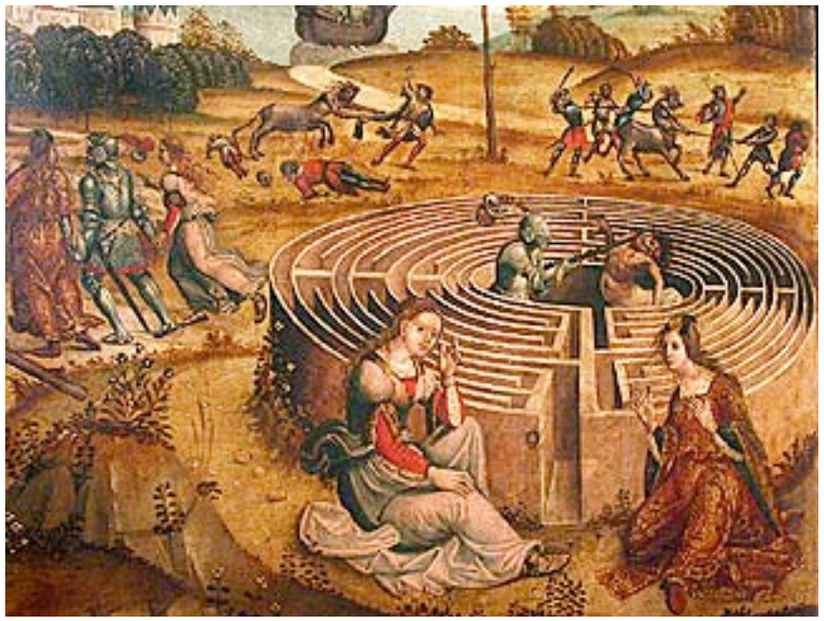
- Los algoritmos clásicos para búsqueda o recorridos en gráficas son:
  - depth-first search
  - breath-first search

- Por ahora solo vamos a "visitar el nodo" pero más tarde veremos acciones específicas a realizar en un nodo.

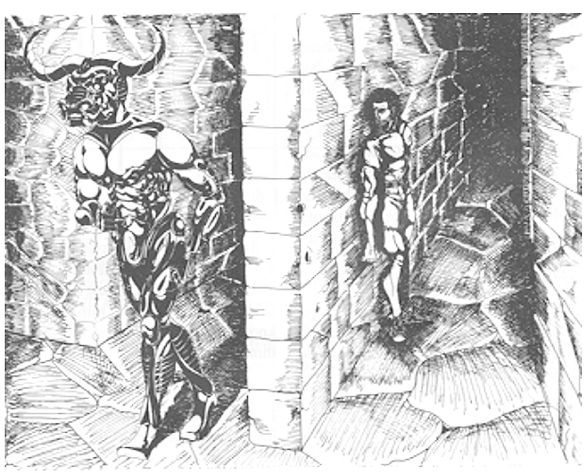
- Se puede elegir visitar el vértice la primera vez que se ve (preorden) o la última vez que se ve (postorden).

Explorando un laberinto

- Queremos encontrar la salida en un laberinto que consiste en pasajes conectados por intersecciones y también revisar todo el laberinto.
- Además del hilo podemos suponer luces, inicialmente apagadas y puertas, inicialmente cerradas en las intersecciones.



- Seguimos la exploración de Trémaux:
- Desenrollar un hilo detrás de nosotros.
- Marcar cada intersección visitada prendiendo una luz.
- Marcar cada pasaje visitado abriendo una puerta.

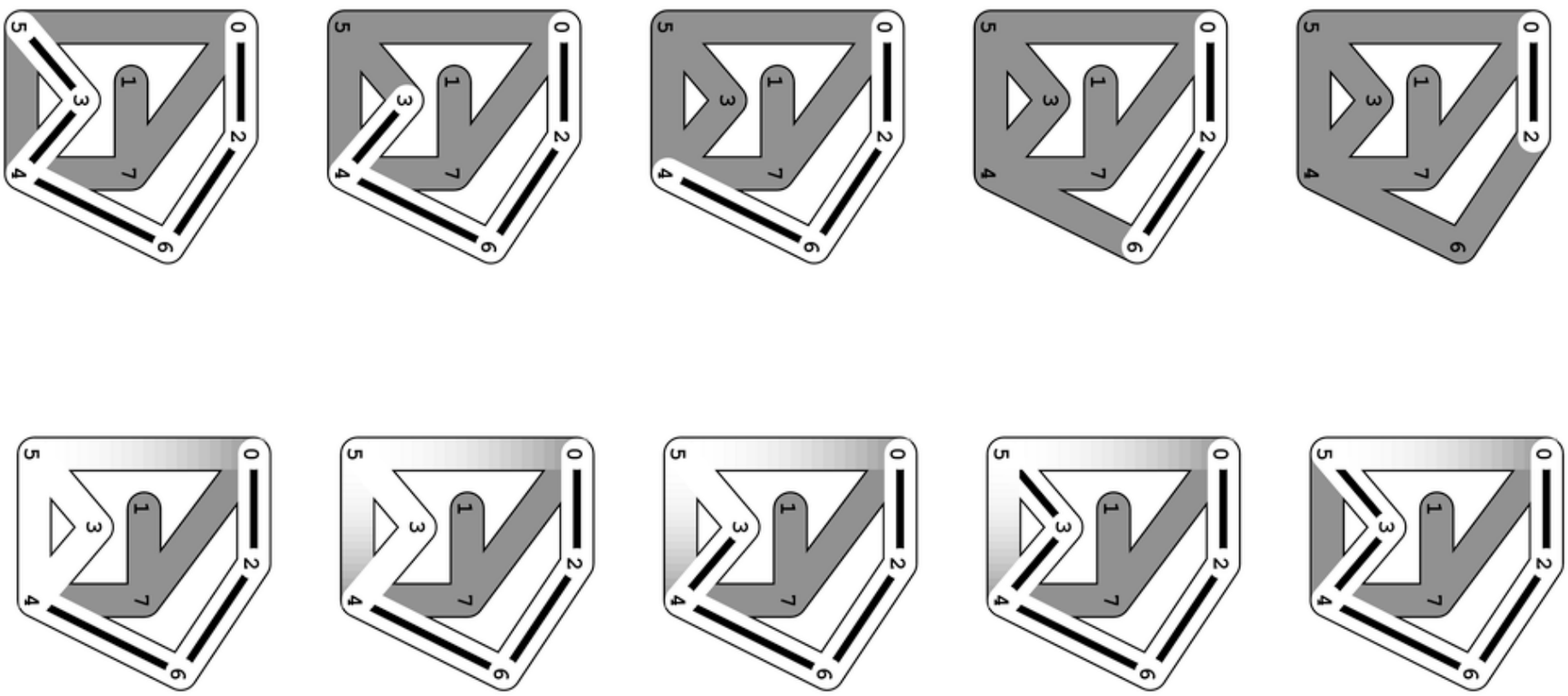


Explorando un laberinto

(I) Si no hay puertas cerradas en la intersección actual, ir al paso (III). Si no, abrir cualquier puerta cerrada para salir a un pasaje y dejar la puerta abierta.

(II) Si se puede ver una intersección al otro lado del pasaje que ya este encendido, probar otra puerta en la intersección actual (del paso 1). Si no (la intersección está apagada), seguir el pasaje hasta la intersección desenrollando el hilo, prender la luz y regresar al paso (I).

(III) Si todas las puertas en la intersección actual están abiertas, verificar si se está en el punto inicial. Si es el caso, parar. Si no, usar el hilo hasta llegar a la intersección que nos llevó allí buscando otra puerta

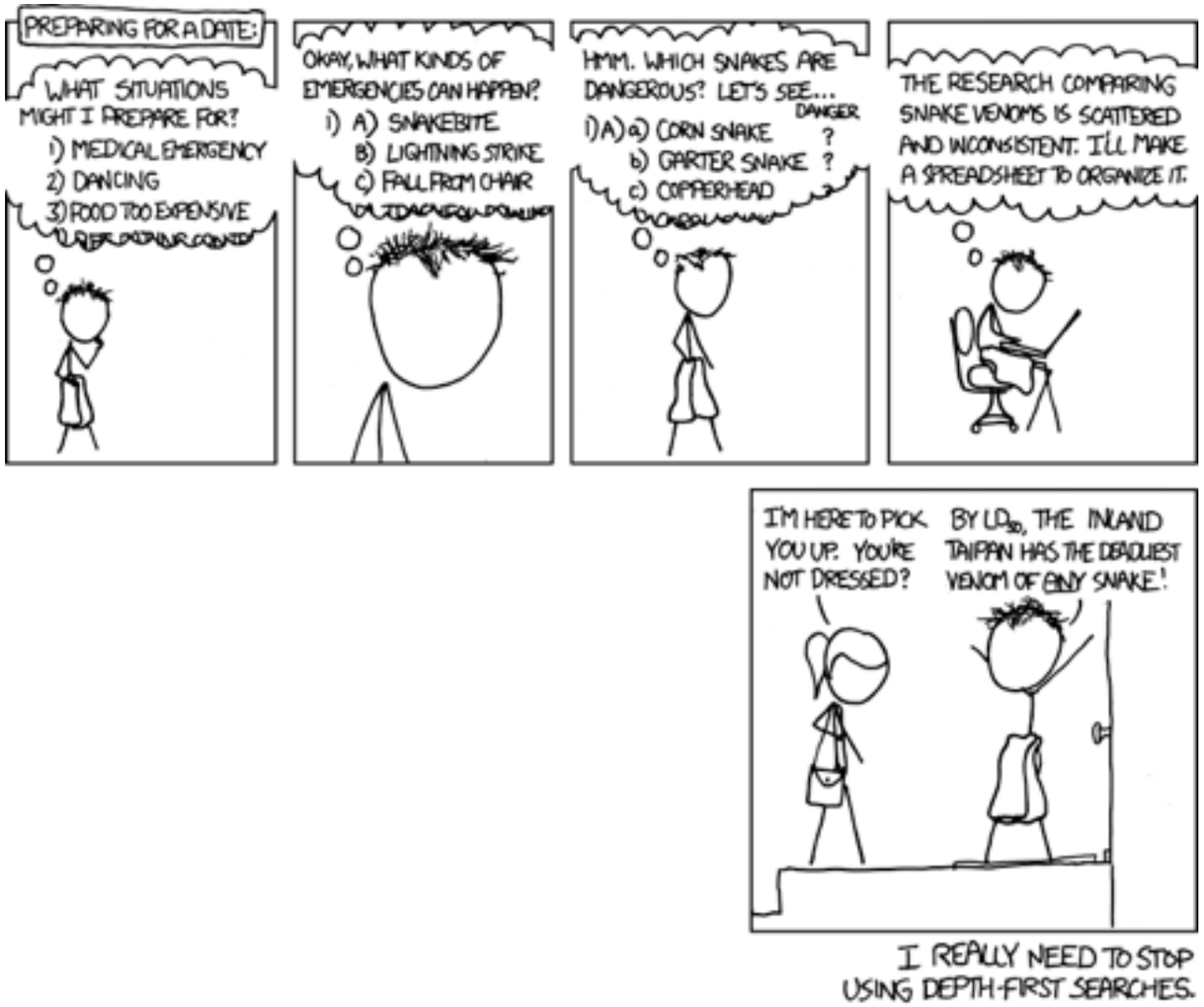

0
2
1
7
3
6
5
4

Explorando un laberinto

- Al usar el algoritmo de exploración de Trémaux, encendemos todas las luces, abrimos todas las puertas y regresamos al punto inicial.
- Hay 4 posibles situaciones:
  - El pasaje está apagado, entonces lo seguimos.
  - El pasaje es el que usamos para entrar (tiene nuestro hilo), lo usamos para salir.
  - La puerta del otro lado esta cerrada (pero la intersección está prendida), no recorremos el pasaje.
  - La puerta al otro lado del pasaje está abierta y la intersección prendida, nos saltamos el pasaje.

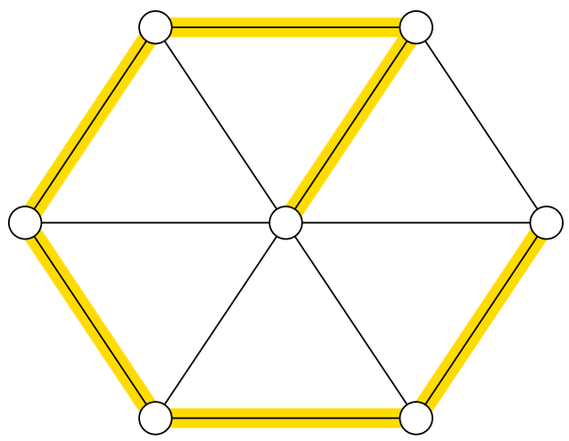
Depth-first search

- La técnica de exploración de Trémaux nos lleva a la función recursiva clásica para recorrer gráficas:
  - para visitar un vértice, lo marcamos como visitado y (recursivamente) visitamos todos los vértices adyacentes a este que no han sido marcados. (DFS)



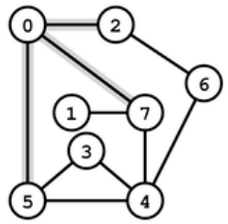
Depth-first search

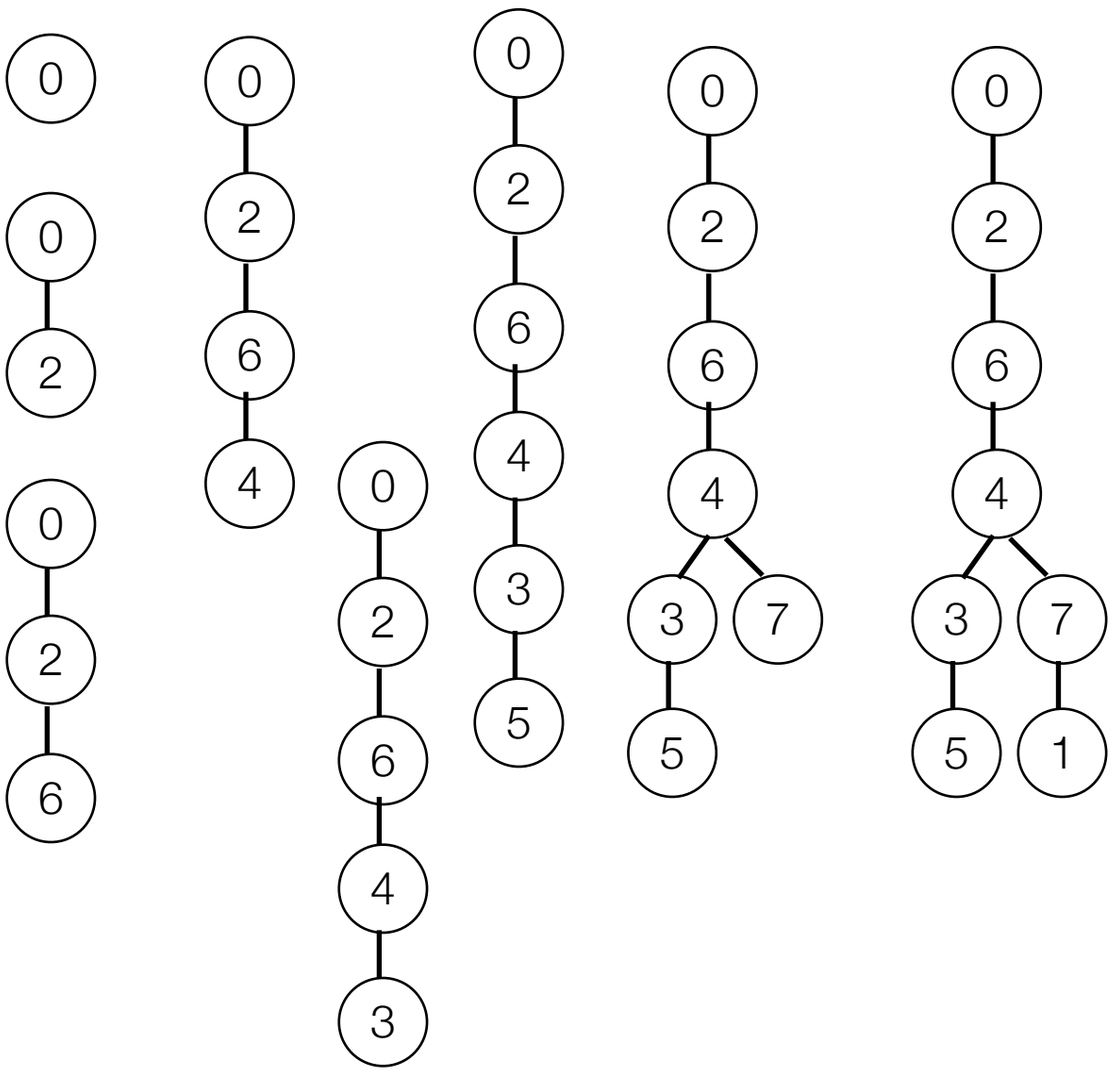
- En depth-first (profundidad primero) search:
  - exploramos en un solo camino en la gráfica hasta lo más lejos que podamos llegar (no se encuentren más vértices).
  - regresamos hasta un punto donde haya vértices que explorar y continuamos (como explorar un laberinto).
- Ejemplo de depth-first search empezando en el centro de la gráfica:



Depth-first search

- Tiempo de ejecución:
  - O(E) ya que cada arista es examinada a lo más dos veces.





Depth-first search

- El algoritmo puede escribirse de manera recursiva o iterativa.
- Ambas versiones toman un vértice fuente (source) *s*.

RECURSIVE_DFS(v):

1. if ***v*** is unmarked
2. mark ***v***
3. for each edge ***(v,w)***
4. RECURSIVE_DFS(***w***)

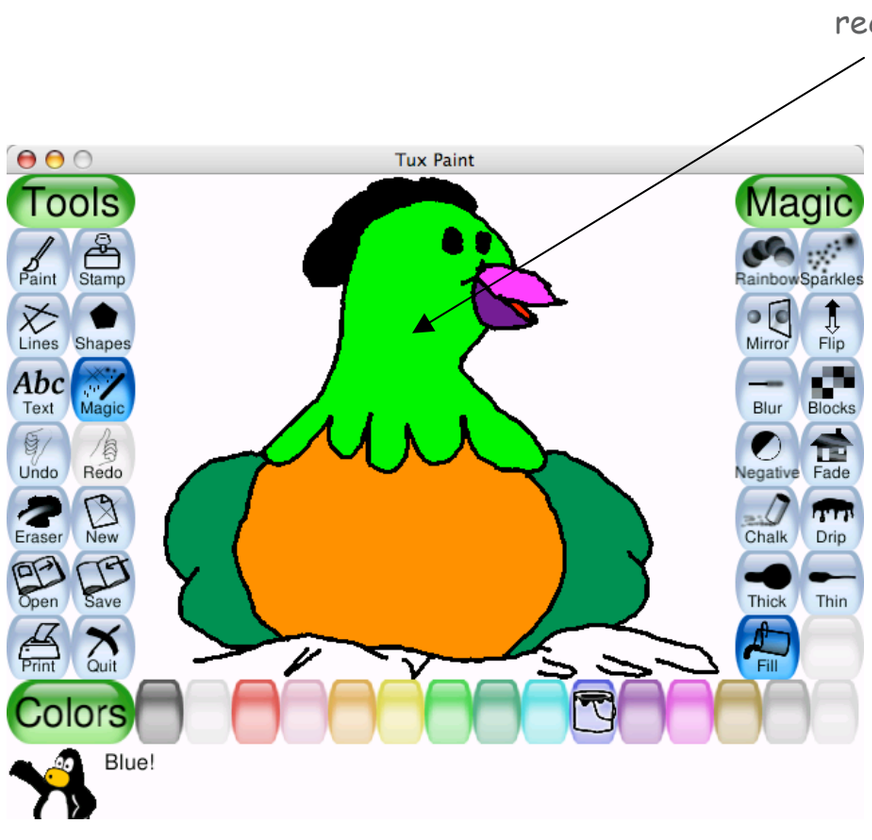
ITERATIVE_DFS(s):

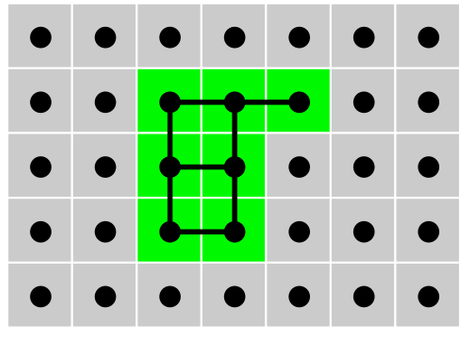
1. ***PUSH(s)***
2. while stack not empty
2. ***v*** ← ***POP***
3. if ***v*** is unmarked
4. mark ***v***
5. for each edge ***(v,w)***
6. ***PUSH(w)***

- Es exactamente el mismo algoritmo con la única diferencia que en la versión iterativa se puede "ver" la pila de la recursión.

Llenado por inundación

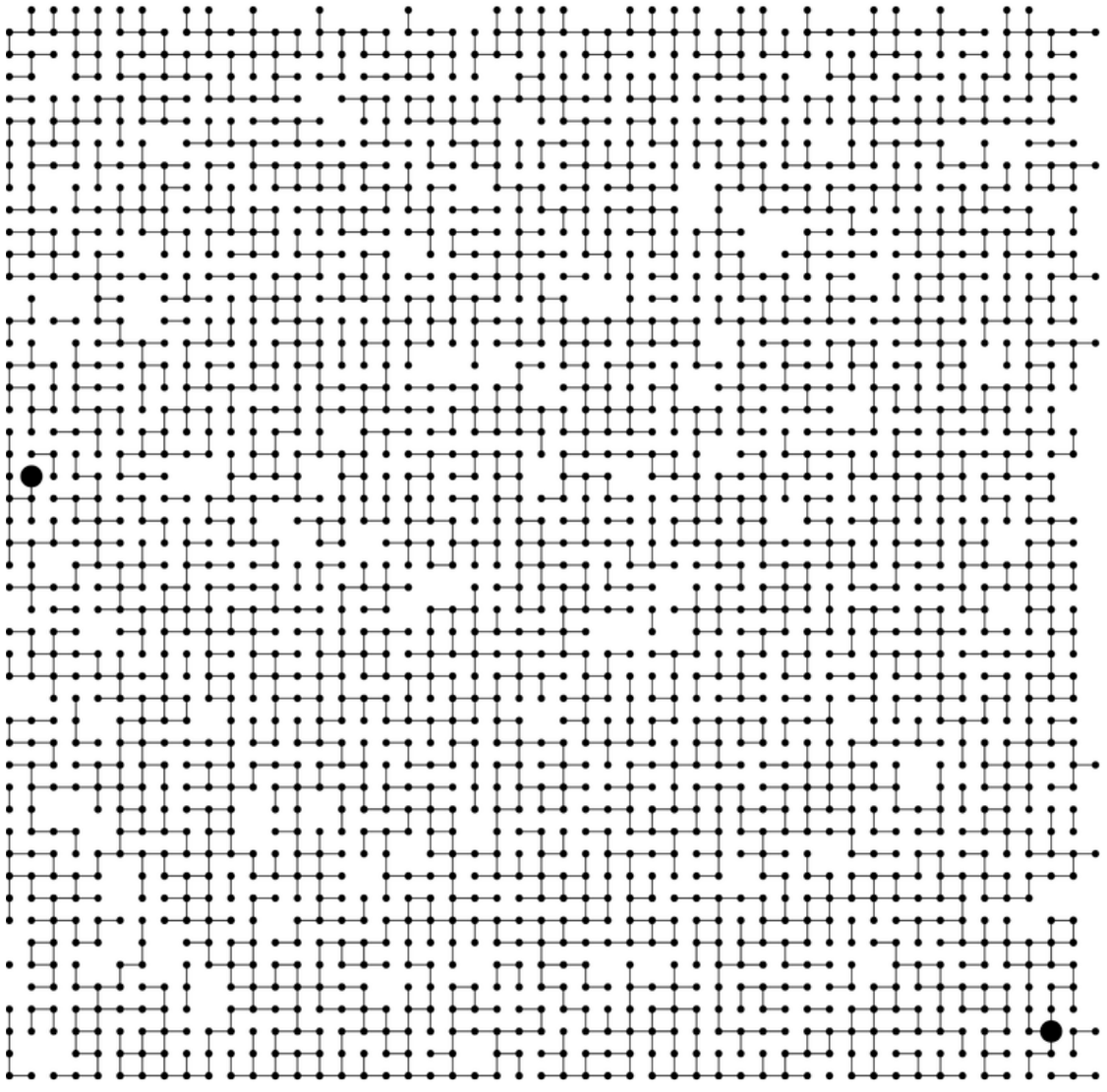
- Dado un pixel de color verde limón en la imagen, cambia el color de todos sus vecinos del mismo color a azul.
- Vértice: pixel
- Arista: pixeles vecinos del mismo color.
- Blob: todos los pixeles alcanzables a partir del pixel inicial.





Encontrar un camino

- Hay un camino de s a t? si lo hay, encuentralo.



Recorridos genéricos en gráficas

- DFS es una (tal vez la más común) instancia de la familia general de algoritmos de recorrido de gráficas.
- El algoritmo de recorrido genérico almacena un conjunto de aristas candidatas en alguna estructura de datos que llamaremos "bolsa".
- Las únicas propiedades importantes de la bolsa es que se puedan poner cosas dentro (como un contenedor en C++) y que se puedan sacar cuando sea necesario.

```
TRAVERSE(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag        (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge (v, w)        (†)
              put (v, w) into the bag      (⋆⋆)
```

Recorridos genéricos en gráficas

```
TRAVERSE(s):
    put (∅, s) in bag
    while the bag is not empty
        take (p, v) from the bag          (⋆)
        if v is unmarked
            mark v
            parent(v) ← p
            for each edge (v, w)          (†)
                put (v, w) into the bag   (⋆⋆)
```

- Notese que estamos guardando las aristas en la bolsa en lugar de los vértices. Esto es porque queremos recordar, cuando visitemos un vértice **v** por primera vez, cuál vértice previamente visitado **p** puso a **v** en la bolsa.
- Llamamos al vértice **p** padre de **v**.

Recorridos genéricos en gráficas

```
TRAVERSE(s):
    put (∅, s) in bag
    while the bag is not empty
        take (p, v) from the bag          (⋆)
        if v is unmarked
            mark v
            parent(v) ← p
            for each edge (v, w)          (†)
                put (v, w) into the bag   (⋆⋆)
```

- TRAVERSE(s) marca cada vértice en una gráfica conectada, exactamente una vez, y el conjunto de aristas *(v,parent(v))* con *parent(v)* $\neq \varnothing$ forma un árbol generador (*spanning tree*) del grafo.
- Cada vértice no se marca más de una vez.
- El conjunto de aristas forman un árbol generador.

```
TRAVERSE(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag        (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge (v, w)        (†)
              put (v, w) into the bag      (⋆⋆)
```

- El tiempo exacto de ejecución de un algoritmo de recorrido de gráficas depende de la representación de la gráfica y de la estructura usada como bolsa. Sin embargo se pueden hacer observaciones generales:

- Ya que cada vértice se visita a lo más una vez, entonces el cíclo $(\dagger)$ se ejecuta a lo más ... V veces .

- Cada arista se pone en la bolsa exactamente dos veces, una como (v,u) y otra como (u,v), entonces la línea $(\star\star)$ se ejecuta a lo más ... $2E$ veces .

- Finalmente, como no podemos sacar más cosas de la bolsa de las que metimos , la línea $(\star)$ se ejecuta a lo más ... $2E + 1$ veces .

Ejemplos de recorridos en gráficas

- Suponiendo una representación con listas de adyacencia, si implementamos la bolsa con una pila (stack), ¿qué algoritmo tenemos?
  - depth-first search.
- Cada ejecución de $(\star)$ o $(\star\ \star)$ toma
  - tiempo constante.
- El tiempo de ejecución total es $O(V+E)$.
- Ya que la gráfica está conectada, $V \leq E+1$, por lo que podemos simplificar el tiempo de ejecución a $O(E)$.
- El árbol generador producido se llama depth-first spanning tree.
- La forma de este árbol depende del orden en que se recorren los nodos adyacentes pero en general serán alargados.

Ejemplos de recorridos en gráficas

- Suponiendo una representación con listas de adyacencia, si implementamos la bolsa con una cola (queue), ¿qué algoritmo tenemos?
  - breath-first search.
- Cada ejecución de $(\star)$ o $(\star\star)$ toma
  - tiempo constante.
- El tiempo de ejecución total es $O(V+E)$.
- Ya que la gráfica está conectada, $V \leq E+1$, por lo que podemos simplificar el tiempo de ejecución a $O(E)$.
- El árbol generador producido contiene los caminos más cortos desde el inicio hasta el vértice actual.
- La forma de este árbol depende del orden en que se recorren los nodos adyacentes pero en general serán anchos y cortos.

Recorridos en gráficas desconectadas

- Si la gráfica está desconectado, entonces TRAVERSE(s) solo visita los nodos en el componente conectado del vértice inicial s.
- Si queremos visitar todos los nodos podemos utilizar el siguiente algoritmo, que calcula el bósque generador de la gráfica.

```
TraverseAll(s):
    for all vertices v
        if v is unmarked
            Traverse(v)
```