**PROTOCOLO:**

**/***

Instituto Tecnológico de León
Ingeniería en Sistemas Computacionales
Estructuras de datos
Pablo Vargas Bermúdez

Exposición
Fecha pactada: 13 de noviembre de 2019
Fecha de entrega: 13 de noviembre de 2019

*/

**DIAPOSITIVAS:**

# Protocolo

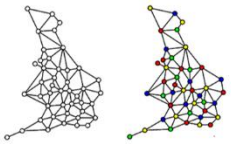
Instituto Tecnológico de León
Ingeniería en Sistemas Computacionales
Estructuras de datos
Pablo Vargas Bermúdez

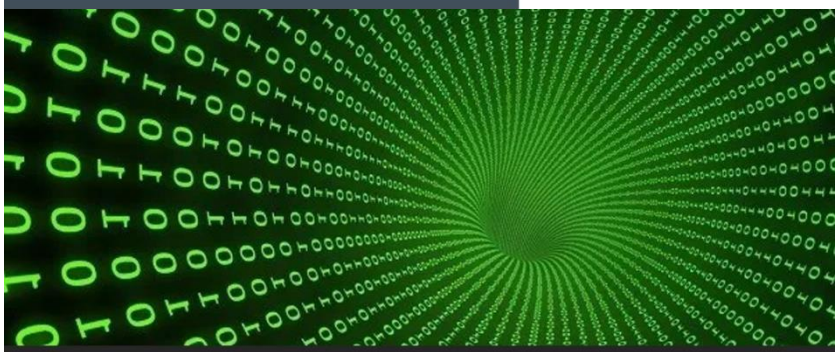
Exposición
Fecha pactada: 13 de noviembre de 2019
Fecha de entrega: 13 de noviembre de 2019

## ¿Qué es?

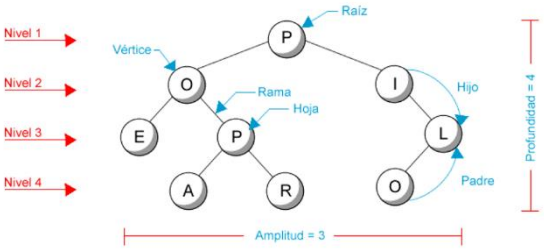
Es un tipo abstracto de datos (TAD), que consiste en un conjunto de nodos (también llamados vértices) y un conjunto de arcos (aristas) que establecen relaciones entre los nodos.

Normalmente, un grafo se define como G=(V,E), siendo V un conjunto cuyos elementos son los vértices del grafo y, E uno cuyos elementos son las aristas (edges en inglés), las cuales son pares (ordenados si el grafo es dirigido) de elementos en V

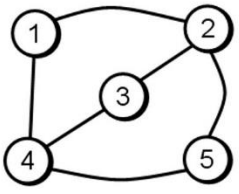




## ¿Cómo se representa?



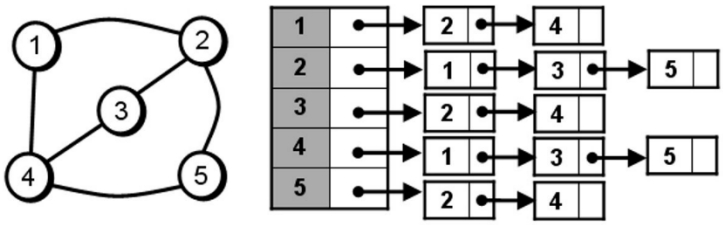

## Conceptos básicos




| M | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 1 | 0 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |

## ¿Cómo se representa?

*Matriz de adyacencias:* se asocia cada fila y cada columna a cada nodo del grafo, siendo los elementos de la matriz la relación entre los mismos, tomando los valores de 1 si existe la arista y 0 en caso contrario.




## Lista de adyacencias

*Lista de adyacencias:* se asocia a cada nodo del grafo una lista que contenga todos aquellos nodos que sean adyacentes a él.

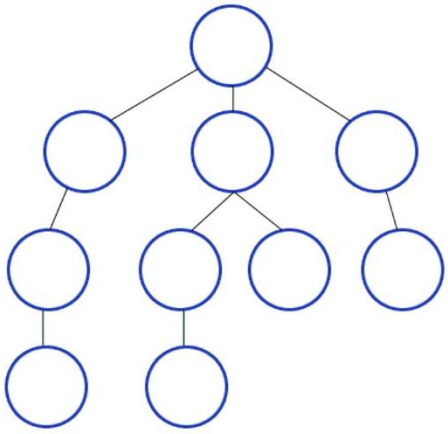

# Depth First Search ( DFS )

**Pseudocódigo:**

Temporal: O( |V| + |E| )
Espacial: O( |V| )

```
procedure DFS(G,v):
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

## Descripción gráfica



# Aplicaciones

Clasificación topológica.

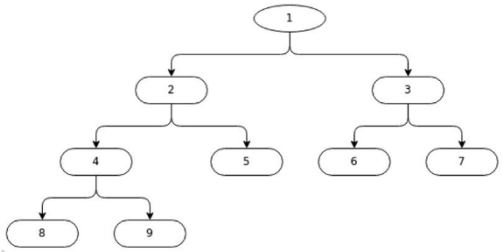
Encontrar los puentes de un gráfico.

Prueba de planaridad.

Resolver acertijos con una sola solución, como laberintos.

La generación de laberintos.

Encontrar biconnectividad en gráficos.

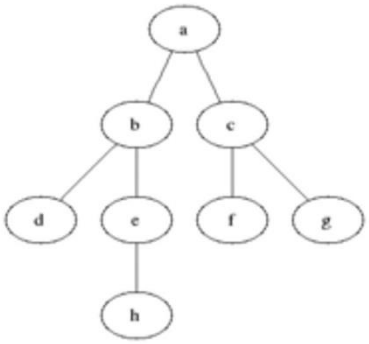



# Ejemplo ( Laberintos )

# Breadth First Search ( BFS )

```
procedure BFS(G,start_v):
    let Q be a queue
    label start_v as discovered
    Q.enqueue(start_v)
    while Q is not empty
        v = Q.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                Q.enqueue(w)
```

Temporal: O( |V| + |E| )
Espacial: O( |V| )



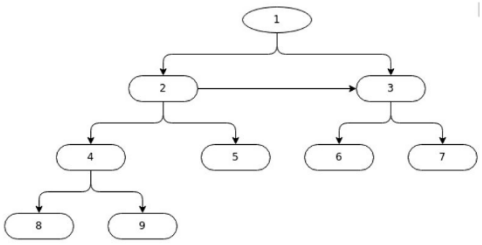

# Descripción gráfica

## Aplicaciones

Encontrar la ruta más corta entre dos nodos u y v, con la longitud de la ruta medida por el número de bordes

La serialización / deserialización de un árbol binario frente a la serialización en orden permite que el árbol se reconstruya de manera eficiente.

Prueba de bipartidismo de un gráfico.




## Ejemplo ( Ruta más corta )

## Bibliografía

Tipo de dato y Representaciones:

https://es.wikipedia.org/wiki/Grafo_(tipo_de_dato_abstracto)

Algoritmos de búsqueda:

http://www.dma.fi.upm.es/personal/gregorio/grafos/web/jagraph/busqueda.html

DFS:

https://en.wikipedia.org/wiki/Depth-first_search#Pseudocode

BFS:

https://en.wikipedia.org/wiki/Breadth-first_search

**EJEMPLOS VISUALES ( GIF'S ):**

**DFSVisual.gif**

**BFSVisual.gif**

**EJEMPLOS:**

**ADJACENCIAS**

```
// Usando listas
first =[
   [0, 1, 0, 1, 0],
   [1, 0, 1, 0, 1],
   [0, 1, 0, 1, 0],
   [1, 0, 1, 0, 1],
   [0, 1, 0, 1, 0]
   ]

// Usando diccionarios
second = {
   1: [2, 4],
   2: [1, 3, 5],
```

```
   3: [2, 4],
   4: [1, 3, 5],
   5: [2, 4]
   }
```
**DFS**
```
"""
1.- Meta:
   Escribir("Encontrar la una ruta del nodo (root) a nodo (leave)")
   Escribir("como en un laberinto")

2.- Datos:
   2.1.- Inicializar:
      // Inicializa el grafo y set de visitados
      G = eval(leerGrafo())
      visitados = set()

      2.1.1.- LeerGrafo:
         // Lee el grafo de un archivo
         Escribir("Archivo: ") file = ?
         abrir(file)
         s = leerTodo(file)
         cerrar(file)
         retornar s

   2.2.- PreguntarNodos:
      Escibir("Nodo inicio: ") inicio = ?
      Escribir("Nodo final: ") final = ?


3.- Calculos:
   // Usa G (grafo), v (inicio) y f (final)
   visitados.add(v)
   res.append(v)
   Si(v == f) entonces
      retornar Verdadero
   terminar
   Para cada w en G[v]:
      Si(w no está en visitados) entonces
         Si(Calculos(G, w, f) entonces
            retornar Verdadero
         terminar
      terminar
   terminar
   res.pop()
   retornar Falso

4.- Resultados:
   Escribir(res)

5.- Navegabilidad:
   No hay
"""
res = list()

def meta():
   s = "Encontrar una ruta del nodo (root) a nodo (leave)"
   s = s + " como en un laberinto"
```

```
    print(s)

def datos():
    def _inicializar():
        global G, visitados
        def __leer_grafo():
            file = input("Archivo: ")
            with open(file) as f:
                return f.read()
        G = eval(__leer_grafo())
        visitados = set()

    def _preguntar_nodos():
        global inicio, final
        inicio = int(input("Nodo inicio: "))
        final  = int(input("Nodo final : "))

    _inicializar()
    _preguntar_nodos()

def calculos(G, v, f):
    visitados.add(v)
    res.append(v)
    if v == f:
        return True
    for w in G[v]:
        if w not in visitados:
            if calculos(G, w, f):
                return True
    res.pop()
    return False

def resultados():
    print(res)

if __name__ == "__main__":
    meta()
    datos()
    calculos(G, inicio, final)
    resultados()
```

**BFS**

```
"""
1.- Meta:
    Escribir("Encontrar la ruta más corta entre dos  nodos")
    Escribir("a partir del número de aristas")

2.- Datos:
    2.1.- Inicalizar:
        visitados = set()
        parent = dict()
        G = eval(leerGrafo())

        2.1.1.- LeerGrafo:
            // Lee el grafo de un archivo
            Escribir("Archivo: ") file = ?
            abrir(file)
            s = leerTodo(file)
```

```
        cerrar(file)
        retornar s

    2.2.- PedirDatos:
      Escribir("Nodo inicio: ") inicio = ?
      Escribir("Nodo final : ") final  = ?

3.- Calculos:
    // Usa G (grafo), inicio (Nodo inicial) y final (Nodo final)
    Q = list()
    visitados.add(inicio)
    Q.append(inicio)
    Mientras(Q no esté vacia) entonces
      v = Q.pop()
      Si(v == final) entonces
        retornar v
      terminar
      Para cada w en G[v] empezar
        Si(w no está en visitados) entonces
          visitados.add(w)
          parent[w] = v
          Q.append(w)
        terminar
      terminar
    terminar

4.- Resultados:
    count = 1
    p = parent[final]
    res.append(final)
    res.append(p)
    Mientras(p != inicio) entonces
      p = parent[p]
      res.append(p)
      count++
    terminar
    Escribir(count)
    Escribir(reverse(res))

5.- Navegabilidad:
    No hay
"""

def meta():
    s = "Encontrar la ruta más corta entre dos nodos"
    s = s + " a partir del número de aristas"
    print(s)

def datos():
    def _inicializar():
      global G, visitados, parent
      def __leer_grafo():
        file = input("Archivo: ")
        with open(file) as f:
          return f.read()
      G = eval(__leer_grafo())
      visitados = set()
```

```
        parent = dict()
    def _pedir_datos():
        global inicio, final
        inicio = int(input("Nodo inicio: "))
        final  = int(input("Nodo final : "))

    _inicializar()
    _pedir_datos()

def calculos(G, inicio, final):
    Q = list()
    visitados.add(inicio)
    Q.append(inicio)
    while Q:
        v = Q.pop()
        if v == final:
            return v
        for w in G[v]:
            if w not in visitados:
                visitados.add(w)
                parent[w] = v
                Q.append(w)

def resultados():
    res = list()
    count = 1
    p = parent[final]
    res.append(final)
    res.append(p)
    while p != inicio:
        p = parent[p]
        res.append(p)
        count = count + 1
    print(res[::-1], count, sep = ': ')

if __name__ == "__main__":
    meta()
    datos()
    calculos(G, inicio, final)
    resultados()
```

**PERSISTENCIA**

**G**

```
{
    1: set([2, 3]),
    2: set([4, 5]),
    3: set([6, 7]),
    4: set([8, 9]),
    5: set(),
    6: set(),
    7: set(),
    8: set(),
    9: set()
}
```

**G2**

```
{
    1: set([2, 3]),
    2: set([3, 4, 5]),
```

```
    3: set([6, 7]),
    4: set([8, 9]),
    5: set(),
    6: set(),
    7: set(),
    8: set(),
    9: set()
}
```