Centro de
Investigación en
Computación

Graphs

Grafos

Tema 3

Course

Analysis and design of algorithms

Instructor

Acosta Bermejo Raúl et al.

Lecture notes

Instituto
Politécnico
Nacional

Table of contents (outline)

Tabla de contenido

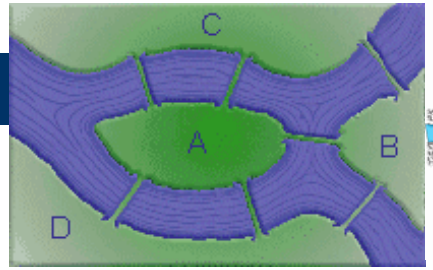
Introduction

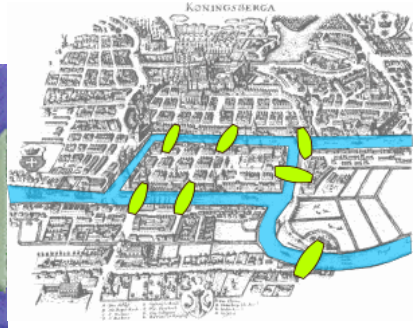
Introducción

Orígenes de la teoría de grafos

Motivation

Historia y motivación





La isla Kueiphof en Koenigsberg (Pomerania, en Rusia) el río que la rodea se divide en dos brazos.

Sobre los brazos estaban construidos *siete puentes* y para los habitantes era motivo de distracción descubrir un itinerario de manera que pudieran regresar al punto de partida, después de haber cruzado por los siete puentes pero pasando sólo una vez por cada uno de ellos.

Leonardo Euler estudió el asunto en 1736 y representó las distintas zonas A, B, C y D por medio de puntos, mientras que los puentes estaban representados por líneas que unían estos puntos. A la figura la llamó grafo, a los puntos los llamó vértices y a las líneas las denominó aristas.

Estudió si una figura lineal se podía dibujar con un solo trazo, sin levantar el lápiz del papel y sin pasar dos veces por el mismo sitio.

Este estudio de Euler ***dio origen a la teoría de los grafos***, que se emplean en el estudio de los circuitos eléctricos, en problemas de transporte, programación con ordenador, etc.

Leonhard Euler

Leonardo Euler

Nació en Suiza y vivió de 1707 a 1783.
Matemático y físico.

- Vivió en Rusia y Alemania la mayor parte de su vida y realizó importantes descubrimientos en áreas tan diversas como el cálculo o la teoría de grafos.
- Introdujo gran parte de la moderna terminología y notación matemática, particularmente para el área del análisis matemático, como por ejemplo la noción de función matemática.
- Asimismo se le conoce por sus trabajos en los campos de la mecánica, óptica y astronomía.

- Definió la constante matemática conocida como número **e** como aquel número real tal que el valor de la derivada (la pendiente de la línea tangente) de la función $f(x)=e^x$ en el punto x=0 es exactamente 1.
- Con su obra Introducción al cálculo infinitesimal, de 1748, popularizó el uso de Pi Π.
- El interés de Euler en la teoría de números procede de la influencia de Christian Goldbach, amigo suyo durante su estancia en la Academia de San Petersburgo. Gran parte de los primeros trabajos de Euler en teoría de números se basan en los trabajos de Pierre de Fermat. Euler desarrolló algunas de las ideas de este matemático francés pero descartó también algunas de sus conjeturas.
- También demostró las identidades de Newton (1642-1727), el pequeño teorema de Fermat, el teorema de Fermat sobre la suma de dos cuadrados e hizo importantes contribuciones al teorema de los cuatro cuadrados de Joseph-Louis de Lagrange (1736-1813).
- Euler y su amigo Daniel Bernoulli (1700-1782) se oponían al monismo de Leibniz (1646-1716) y a la corriente filosófica representada por Christian Wolff.

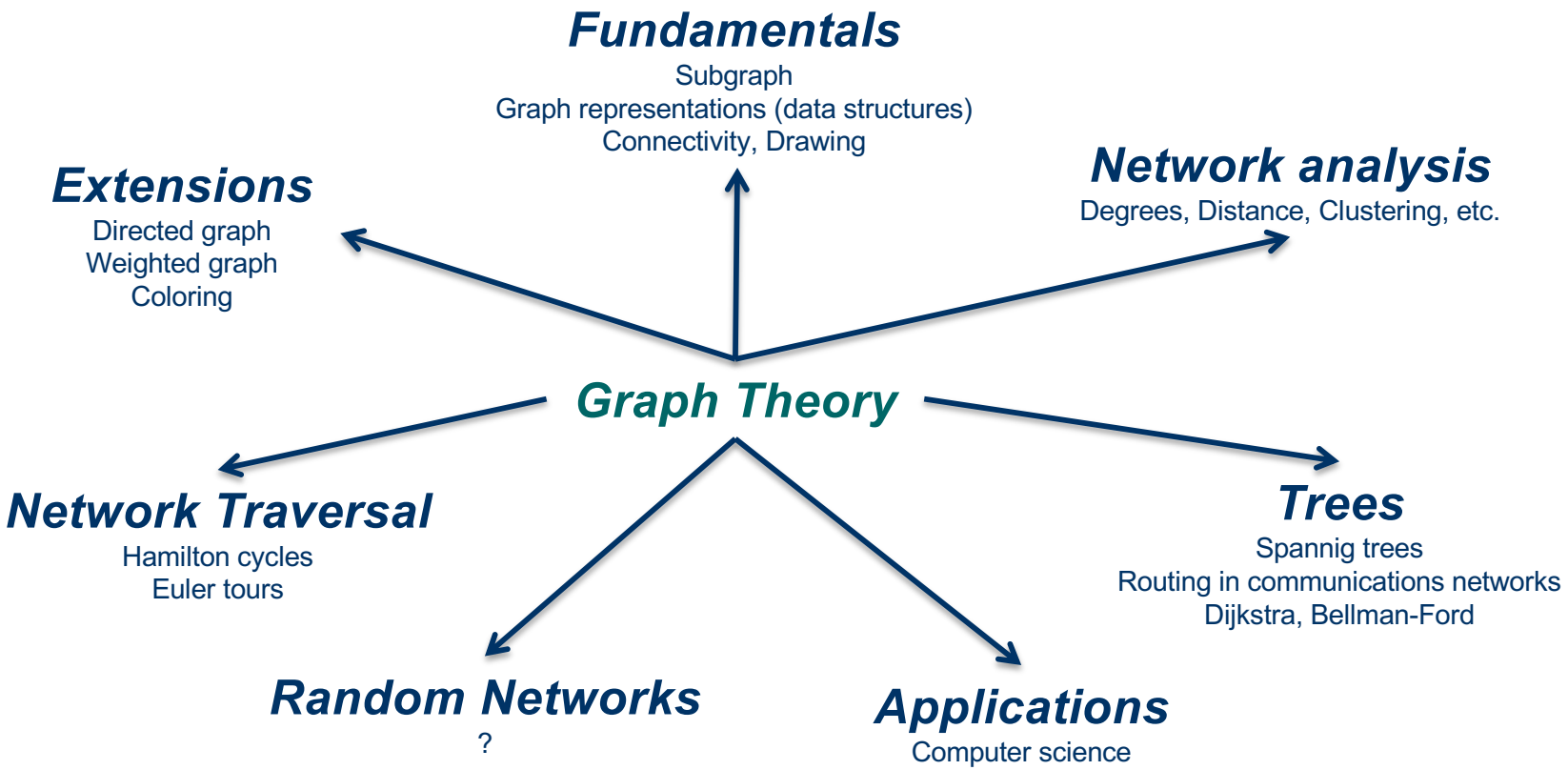
Topics

En la wikipedia

Adjacency Matrix, Adjacency Relation, ***Algorithmic Graph Theory****, Articulation Vertex, Blue-Empty Coloring, Bridge, Chromatic Number, Chromatic Polynomial, Circuit Rank, Cyclomatic Number, Degree,* ***Dijkstra's Algorithm****, Eccentricity, Edge Coloring, Edge Connectivity, Eulerian Circuit, Eulerian Trail, Floyd's Algorithm, Girth, Graph Crossing Number, Graph Cycle, Graph Diameter, Graph Factor, Graph Join, Graph Radius, Graph Two-Coloring,* ***Group Theory****,* ***Hamiltonian Circuit****, Hasse Diagram, Hub, Indegree, Integral Drawing, Isthmus, Local Degree, Monochromatic Forced Triangle, Outdegree, Party Problem, Pólya Enumeration Theorem, Pólya Polynomial, Ramsey Number, Re-Entrant Circuit, Separating Edge, Tait Coloring, Tait Cycle,* ***Traveling Salesman Problem****,* ***Tree****, Tutte's Theorem, Unicursal Circuit,* ***Vertex Coloring****, Vertex Degree, Walk,* ***Convex Hull****, etc .*

Mind map

Mapa mental/conceptual - Campo semántico



3.1 Basic definitions and applications

Definiciones básicas y aplicaciones

Basic definitions and applications

Definiciones básicas y aplicaciones

Vertice (vertices / Nodo, punto)

A set of vértices V, $v \in V$

Represented by a circle O (filled or not).

Geometry
Vertex.- An angle point

Edge (arista)

A set of edges E, $e \in E$

Represented by a line (with/without arrow)

Graph (grafo)

Denoted G, Definied by G=(V,E) is the tuple of 2 sets. $v \in V(G)$

Represented by drawing V and E

undirected G, directed G

Acyclic, cyclic G

Basic definitions and applications

Definiciones básicas y aplicaciones

- *If e joins u, v ∈ V, we write e = ⟨u, v⟩. Vertex u and v are said to be* ***adjacent****.*
- *Edge e is said to be* ***incident*** *with vertices u and v, respectively.*
- *The number of edges incident with a vertex v is called the* ***degree*** *of v, denoted as δ(v)* or deg(v). *Loops are counted twice.*
- ***Teorema****: For all graphs G, the sum of the vertex degrees is twice the number of edges, that is,*

$$\Sigma_{v \in V(G)} \delta(v) = 2\,|E(G)|$$

odd = impar
even = par

- **Corollary:** *For any graph, the number of vertices with odd degree is even*.
- If every vertex has the same degree, the graph is called **regular**. In a **k-regular** graph each vertex has degree *k*. As a special case, 3-regular graphs are also called **cubic graphs**.

Basic definitions and applications

Definiciones básicas y aplicaciones

Neighbor $N(v) = \{ w \in V(G) / v \neq w, \exists e \in E(G): e=(v,w) \}$

Paths (caminos, trayectorias, rutas)

In a graph G=(V,E) is a sequence P of nodes v_1, v_2, … v_{k-1}, v_k with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G.

Cycles (ciclos)

We call a previous sequence (path) of nodes a cycle if $v_1 = v_k$.

We call ***simple*** path if all its vértices are distinct, or ***simple cycle*** if …

Connected (conexo)

If for every pair of nodes u and v, there is a path from u to v.

A **tree** is a connected graph with no cycles.

Basic definitions and applications

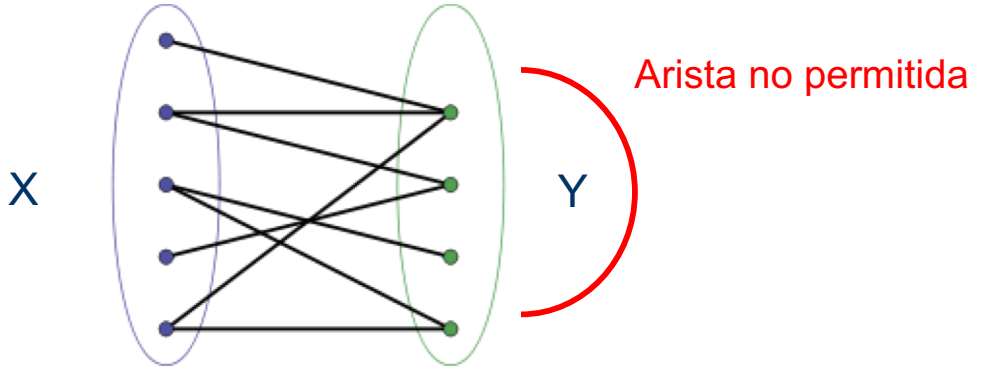
Definiciones básicas y aplicaciones

Bipartie graphs

A graph G=(V,E) undirected is bipartite

if its nodes set V can be partitioned into sets X and Y (disjoint, V=X∪Y)

in such a way that every edge has one end in X and the other end in Y.



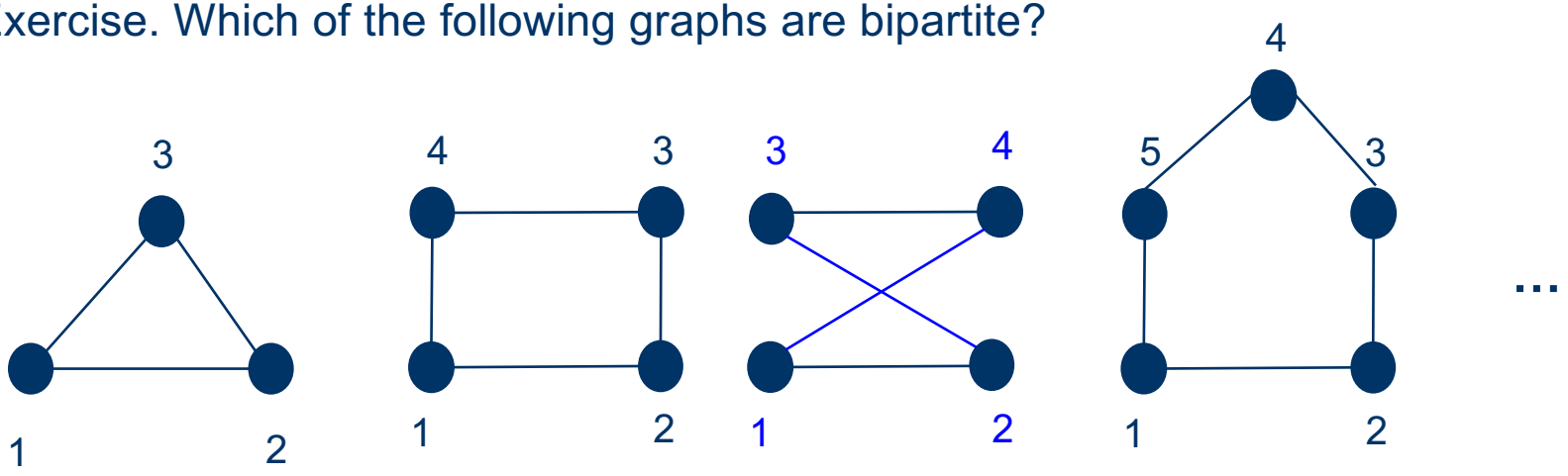
- If |X|=|Y|, that is, if the two subsets have equal cardinality, then G is called a ***balanced bipartite graph***.
- If all vertices on the same side of the bipartition have the same degree, then G is called ***biregular***.

Basic definitions and applications

Definiciones básicas y aplicaciones

Bipartie graphs

- **Theorem** (König, 1936) A graph is bipartite ↔ (syss, iff) it does not have an odd length cycle.
- Exercise. Which of the following graphs are bipartite?



Ciclo impar / odd

Ciclo par / even
Bipartita

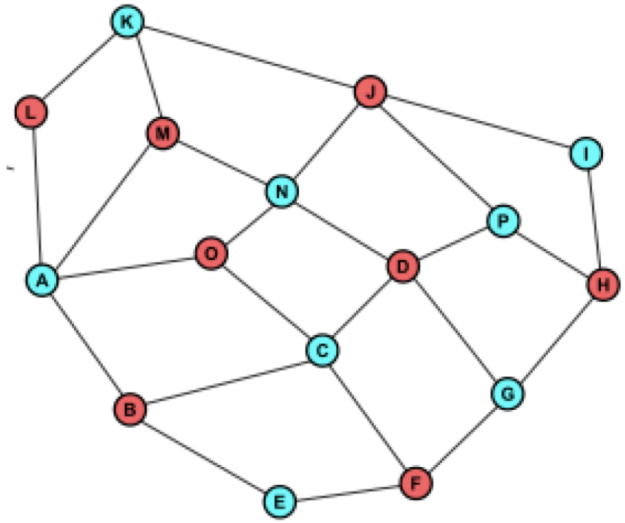
Ciclo impar / odd

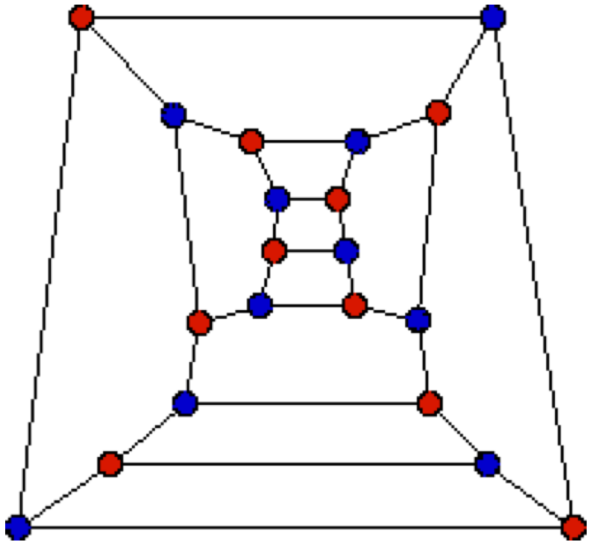
Basic definitions and applications

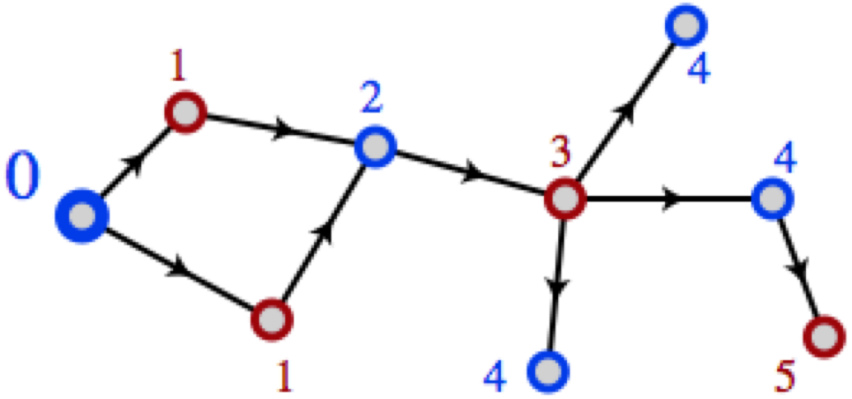
Definiciones básicas y aplicaciones

Bipartie graphs

Examples







Basic definitions and applications

Definiciones básicas y aplicaciones

Representing graphs

1. *Adjacency matrices*

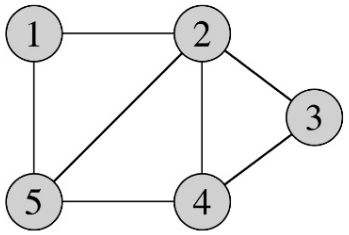
   A two-dimensional array A, with n rows and n columns, where n=IVI. A[i,j] is equal to 1 if there is an edge joining i and j, and it is equal to 0 otherwise.

   If G is undirected then A is symmetric: A[i,j] = A[j,i].

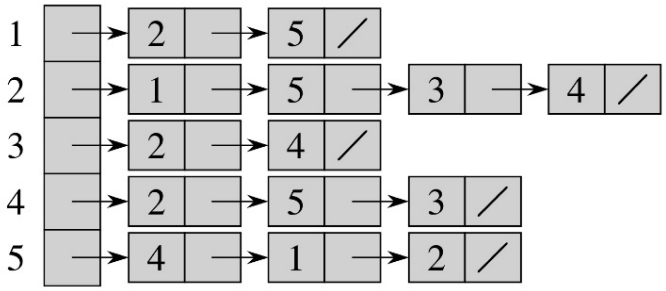
   Properties: adjacent(i,j) is O(1) good, size is $\Theta(n^2)$ bad enormous.

2. *Adjacency lists*

   An n-element array V, where V[i] represents node i and points to a double linked list L_i that contains an entry for each edge e=(i,j) incident to i.



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Basic definitions and applications

Definiciones básicas y aplicaciones

Representing graphs

3. *List of edges*

   Each element in the list has 2 vertex

   For convenince you have the number of vertex and edges.

Software

Software para usar grafos

Gratuito (Open Source)

1. SageMath (http://www.sagemath.org/)
2. Graphviz (http://www.graphviz.org/)
3. GraphTea (http://www.graphtheorysoftware.com/)

Comercial

Mathematica (http://www.wolfram.com/mathematica/)

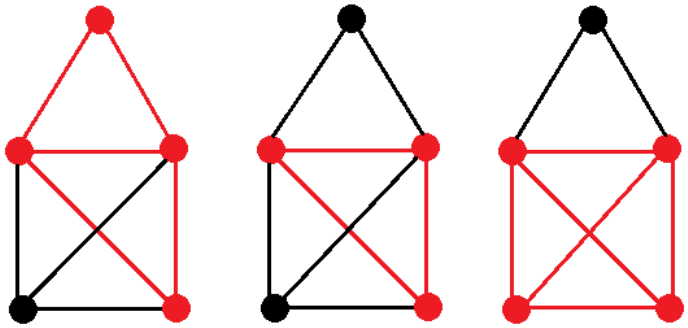
Otros sistemas algebraicos computacionales: Maple, Scilab, Matlab, etc.

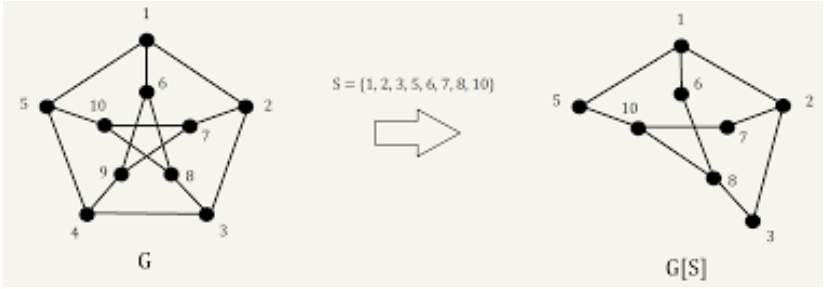
Basic definitions and applications

Definiciones básicas y aplicaciones

Subgraphs

- A graph H is a subgraph of G if H consists of a subset of the edges and vertices of G, such that the end points of edges in G are also contained in H.
- ***Definition***: A graph H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ such that for all $e \in E(H)$ with $e = (u, v)$, we have that $u, v \in V(H)$. When H is a subgraph of G, we write $H \subseteq G$.





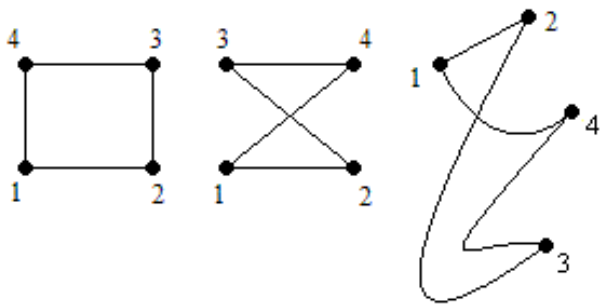
- We also say that G is a supergraph of H.

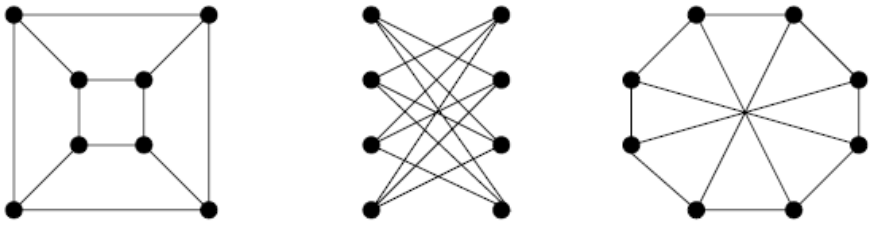
Basic definitions and applications

Definiciones básicas y aplicaciones

Isomorphic graphs

- Consider two graphs G = (V, E) and G* = (V*, E*). G and G* are isomorphic if there exists a one-to-one mapping ϕ: V $\rightarrow$ V* such that for every edge e $\in$ E with e = (u, v), there is a unique edge e* $\in$ E* with e* = (ϕ(u), ϕ(v)).
- Stated differently, two graphs G and G* are isomorphic if we can uniquely map the vertices and edges of G to those of G* such that if two vertices were joined in G by a number of edges, their counterparts in G* will be joined by the same number of edges.





G_1 G_2 G_3

Basic definitions and applications

Definiciones básicas y aplicaciones

Isomorphic graphs

Theorem

If two graphs G and G* are isomorphic, then their respective ordered degree sequences should be the same.

Give an algorithm that test if 2 graphs are isomorphic.

What about its O()?.

Basic definitions and applications

Definiciones básicas y aplicaciones

Isomorphic graphs

- The problem is neither known to be solvable in polynomial time nor NP-complete, and therefore may be in the computational complexity class NP-intermediate.
- It is known that the **graph isomorphism problem** is in the low hierarchy of class NP, which implies that it is not NP-complete unless the polynomial time hierarchy collapses to its second level.
- At the same time, isomorphism for many special classes of graphs can be solved in polynomial time, and in practice graph isomorphism can often be solved efficiently.

Basic definitions and applications

Definiciones básicas y aplicaciones

Isomorphic algorithms

- The best current theoretical algorithm is due to Babai & Luks (1983):
  - It is based on the earlier work by Luks (1982) combined with a subfactorial algorithm due to Zemlyachenko, Korneenko & Tyshkevich (1985). The algorithm relies on the classification of finite simple groups.
  - Without CFSG, a slightly weaker bound to $2^{O(\sqrt{n}\log^2 n)}$ was obtained first for strongly regular graphs by László Babai (1980), and then extended to general graphs by Babai & Luks (1983).
- There are several competing practical algorithms for graph isomorphism, due to McKay (1981), Schmidt & Druffel (1976), Ullman (1976), etc.
- While they seem to perform well on random graphs, a major drawback of these algorithms is their **exponential** time performance in the worst case.
- In 2015, Babai announced a quasipolynomial time algorithm.

Basic definitions and applications

Definiciones básicas y aplicaciones

Ullman's Subgraph Isomorphism Algorithm

- The subgraph isomorphism problem asks whether a graph G has a subgraph G' subset G that is isomorphmic to a graph P.

So basically you have the picture on the box of a puzzle (G) and want to know where a particular piece (P) fits, if at all.

It is NP-complete because Hamiltonian cycle is a special case.

El ciclo hamiltoniano es el contorno de la pieza

- In 1976 Ullman proposed a backtracking algorithm for this problem. The writeup in the original paper is hard to follow because the pseudo-code doesn't use functions or even loops.
- La libreria VFLIB se especializa en isomorfismos:
  https://github.com/chebee7i/vflib

3.2 Connectivity and graph traversal

Conectividad y recorridos de grafos

Connectivity and graph traversal

Conectividad y recorridos de grafos

Connectivity

Let G=(V,E) be a graph and two nodes *s* and *t*,

Is there a path from *s* to *t* in G?

Which is the most efficient algorithm?

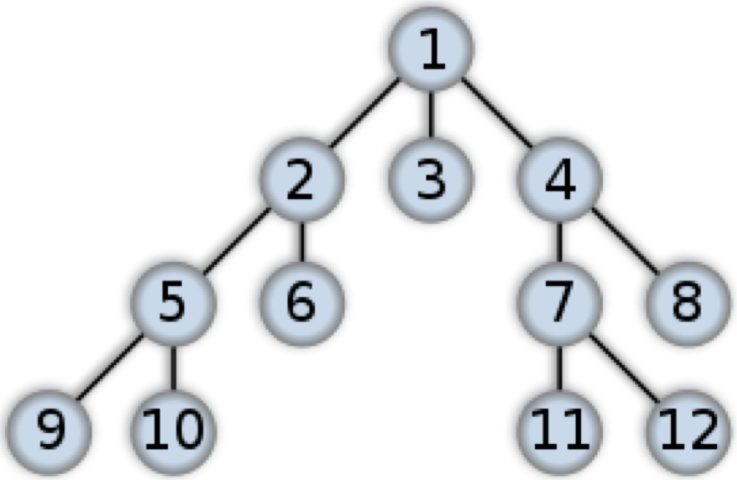
We are going to see two algorithms BFS y DFS to answer.

Connectivity and graph traversal

Conectividad y recorridos de grafos

Breadth-First Search (BFS), búsqueda a lo ancho.

It explores G by considering all nearby nodes first (**layer**), rather than exploring deeply in any particular direction.



Nearby: cercano, adyacente, vecino.

Order in which the nodes are expanded.

BFS was invented in the late 1950s by E. F. Moore, who used it to find the shortest path out of a maze, and discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).

Connectivity and graph traversal

Conectividad y recorridos de grafos

Algoritm **Ver 1**

BFS(s)
Mark s as "Visited".
Initialize R = {s}.
Define layer L_0 = {s}.
while L_i is not empty
  for each node u ∈ L_i
    Consider each edge (u, v) incident to v
    if v is not marked "Visited" then
      Mark v "Visited"
      Add v to the set R and to layer L_{i+1}
    endif
  endfor
endwhile

Comentarios

- Pseudo-codigo: No se describe el detalle, por ejemplo, Initialize R, o Consider
- Las capas (layer) son implementadas como colas (queues).
- Implementación iterativa (vs recursiva).

Complejidad

- Se iteran 2 ciclos.
- Costo de los TADs.

Connectivity and graph traversal

Conectividad y recorridos de grafos

Algoritm **Ver 2**

```
BFS(G, v)
  for each node n in G:
     n.distance = INFINITY
     n.parent = NIL

  create empty queue Q
  v.distance = 0
  Q.enqueue(v)

  while Q is not empty:
    u = Q.dequeue()
    for each node n that is adjacent to u:
      if n.distance == INFINITY:
         n.distance = u.distance + 1
         n.parent = u
         Q.enqueue(n)
```

Comentarios

- Menos pseudo-código.
- Implementacion: iterativa.

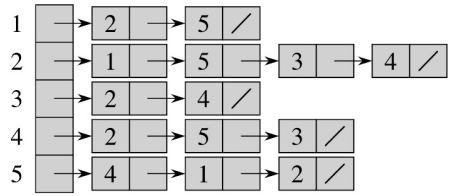
O(V) ya que Q se construye con los V.

O(E) para cada nodo.

- Para cada nodo no se recorren las aristas a las que no están conectados.
- Esto determina * o +

Connectivity and graph traversal

Conectividad y recorridos de grafos



BFS Time and space complexity

V+E

5 vértices + 14 aristas

Complejidad temporal

O(IVI + IEI)

Note that O(|E|) may vary between O(1) and O($|V|^2$), depending on how sparse the input graph is.

escaso, esparcido / antónimo denso

Complejidad espacial

If the graph is represented by an:

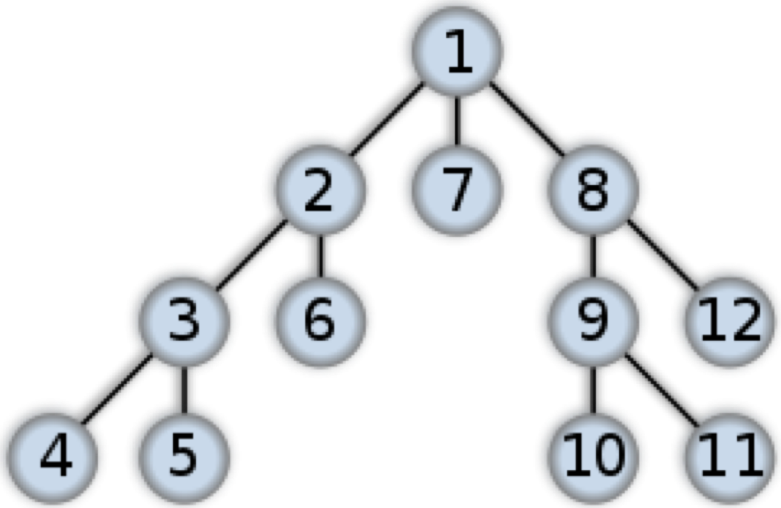
- Adjacency list it occupies Θ (|V|+|E|) space in memory.
- Adjacency matrix representation occupies Θ ($|V|^2$).

Connectivity and graph traversal

Conectividad y recorridos de grafos

Depth-First Search (DFS), búsqueda en profundidad.

It explores G by going as deeply as possible, and only retreating when necessary.



Investigar que es un Trémaux tree

- It is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.
- **DFS** was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

Connectivity and graph traversal

Conectividad y recorridos de grafos

Algoritm **Ver 1** Recursive implementation

```
DFS-r(G,v)
   label v as discovered
   for all edges w in G.adjacentEdges(v) do
      if vertex w is not labeled as discovered then
         recursively call DFS-r(G,w)
```

Algoritm **Ver 2** Iterative implementation

```
DFS-i(G,v)
      let S be a stack
      S.push(v)
      while S is not empty
            v = S.pop()
            if v is not labeled as discovered:
               label v as discovered
               for all edges from v to w in G.adjacentEdges(v) do
                  S.push(w)
```

```
DFS-r2(u)
Mark u as "Visited" and add u to R.
for each edge (u, v) incident to u
   if v is not marked "Visited" then
      Add v to R.
      Recursively invoke DFS-r2(v).
   endif
endfor
```

Cual seria el algoritmo cuando se usa una implementacion de matriz?

Connectivity and graph traversal

Conectividad y recorridos de grafos

DFS Time and space complexity

The time and space analysis of DFS differs according to its application area, but in general:

Complejidad temporal

O(IVI + IEI)

Complejidad espacial

O(IVI) if entire graph is traversed without repetition.

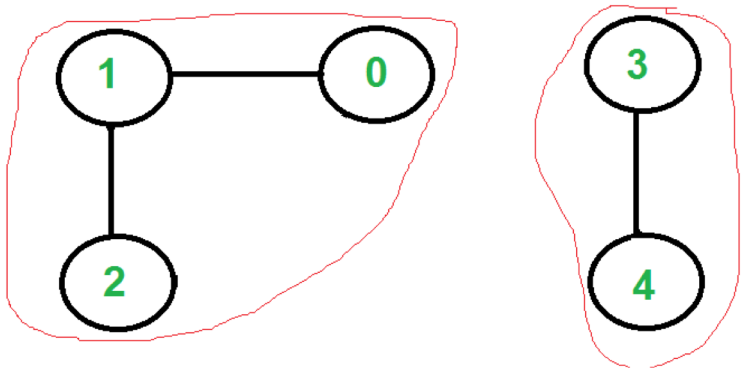
Connectivity and graph traversal

Conectividad y recorridos de grafos

Problems resolved with BFS or DFS

1.- Connected Components (CC)

- **F**inding all
- We don't want to determine a path between a specific pair of nodes.
- We want to produce ALL the connected components of G.
- One option is to determine if G is a connected graph.



Connectivity and graph traversal

Conectividad y recorridos de grafos

Algorithm for CC We can use BFS or DFS.

- Nodes are labeled 1, 2, . . . , n, and there is an array B that stores the "Visited/Unvisited" status of each node.
- We first grow the component R1 containing 1 (in time proportional to the number of nodes and edges in R1).
- We then walk through the entries of the array B in sequence.
  - Either we reach the end and discover that all nodes have been visited, or
  - We come to the first node *i* that is not yet visited.
- The node *i* must belong to a different connected component, so we proceed to grow the component R_i containing it in time proportional to the number of nodes and edges in R_i.
- We then return to scanning the array B for unvisited nodes, starting from the node *i*, and continue in this way.

Connectivity and graph traversal

Conectividad y recorridos de grafos

2.- Finding Cut-Points in a Graph

- Given a connected graph G = (V, E), we say that u $\in$ V is a cut-point if deleting u disconnects G; in other words, if G-{u} is not connected.
- We can think of the cut-points as the “weak points” of G; the destruction of a single cut-point separates the graph into multiple pieces.
- We use de **earliest** definition of u:

It is the earliest node x such that some node in the sub-tree rooted at u is joined by a non-tree edge to x.

Let T be a DFS tree of G, with root r.

1. A node u ≠ r is a cut-point if and only if there is a child v of u for which u $\preccurlyeq$ earliest(v).
2. The root r is cut-point if and only if it has more than one child.

Connectivity and graph traversal

Conectividad y recorridos de grafos

Algorithm for Finding Cut-Points

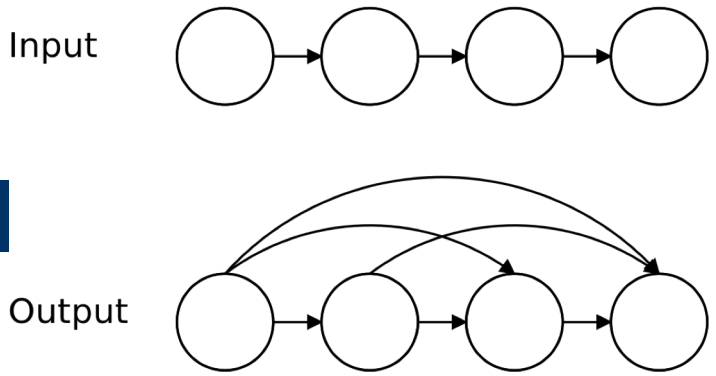
Pseudo-code
For each node u
To process a DFS tree from leaves upward using **earliest**.

Earliest(u)

Compute a DFS tree T of G rooted at r.
Now process the nodes in T from the leaves upward,
so that a node u is only processed after all its children:
To process node u:
if u is a leaf, then earliest(u) is just the earliest node
to which u is joined by a non-tree edge.
We define earliest(u) = u if u has no incident non-tree edges.
else (u is not a leaf) Consider the set
S = {u} ∪{w : (u, w) is a non-tree edge} ∪ {earliest(v) : v is a child of u}
Define earliest(u) to be the earliest node in S
endif

Transitive closure

Cerradura transitiva



Definitions

In mathematics:

The TC of a binary relation R on a set X is the transitive relation R+ on set X such that R+ contains R and R+ is minimal

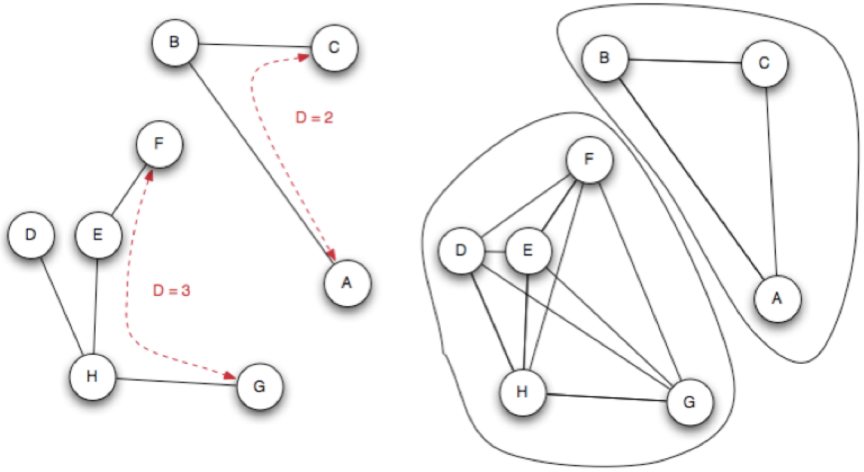
In graphs:

The TC of a digraph G is another digraph with the same set of vertices, but with an edge from v to w in the transitive closure if and only if w is reachable from v in G.

- By convention, every vertex is reachable from itself, so the transitive closure has V self-loops.
- Generally, the transitive closure of a digraph has many more edges than the digraph itself, and it is not at all unusual for a sparse graph to have a dense transitive closure.
- Since transitive closures are typically dense, we normally represent them with a matrix of boolean values, where the entry in row v and column w is true if and only if w is reachable from v.

Transitive closure

Cerradura transitiva



Efficient algorithms

for computing the TC of a graph

- Found in Nuutila (1995).
- The fastest worst-case methods, which are not practical, reduce the problem to matrix multiplication.
- The problem can also be solved by the Floyd–Warshall algorithm, or by repeated breadth-first search or depth-first search starting from each node of the graph.
- More recent research has explored efficient ways of computing transitive closure on distributed systems based on the MapReduce paradigm (Hadoop).

Transitive closure

Cerradura transitiva

Floyd–Warshall

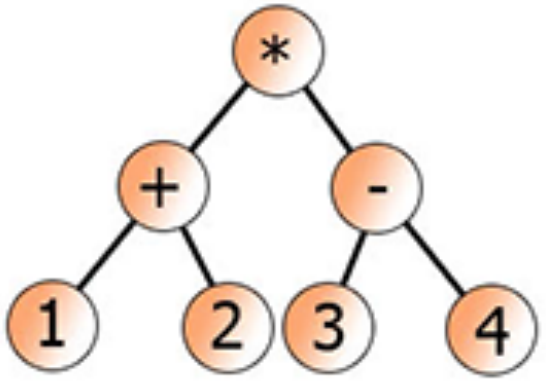
Algorithm *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Complejidad $O(n^3)$.

Clasic topics

Temas clásicos



((1+2)*(3-4))

Notaciones aritméticas

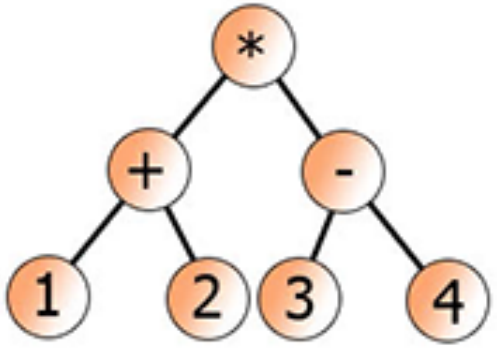
- Notación prefija
  - Operador Operando-1 Operadondo-2 = + A B
- Notación infija
  - Operando-1 Operador Operadondo-2 = A + B
- Notación postfija
  - Operando-1 Operadondo-2 Operador = A B +

- Si se implementan las expresiones con un árbol binario las notaciones se obtienen de un recorrido en profundidad DFS.
- Las notaciones posfija y prefija no necesitan paréntesis para indicar el orden de las operaciones, mientras la aridad del operador sea fija.

Implementación más eficiente (memoria)

Clasic topics

Temas clásicos



Recursive algorithm

```
string DFS(node n)
{
 string l, r;

     If( n==null) return “”;
     l= DFS(n->left);
     r= DFS(n->right);
     return strcat(strcat(l,r), n->value);
}
```

Si se tiene un árbol n-ario se usa un ciclo y una sola invocación (lista)

Variable	Value
node n	
string l	
string r	
uname vars	return
IP register	l= DFS...

Repaso de SO

Conceptos básicos

Proceso: es un programa (sólo el código) en ejecución.

- Significa que está cargado en RAM.
- Hay ciertos valores en los registros del procesador, y
- Las funciones se implementan con la pila.

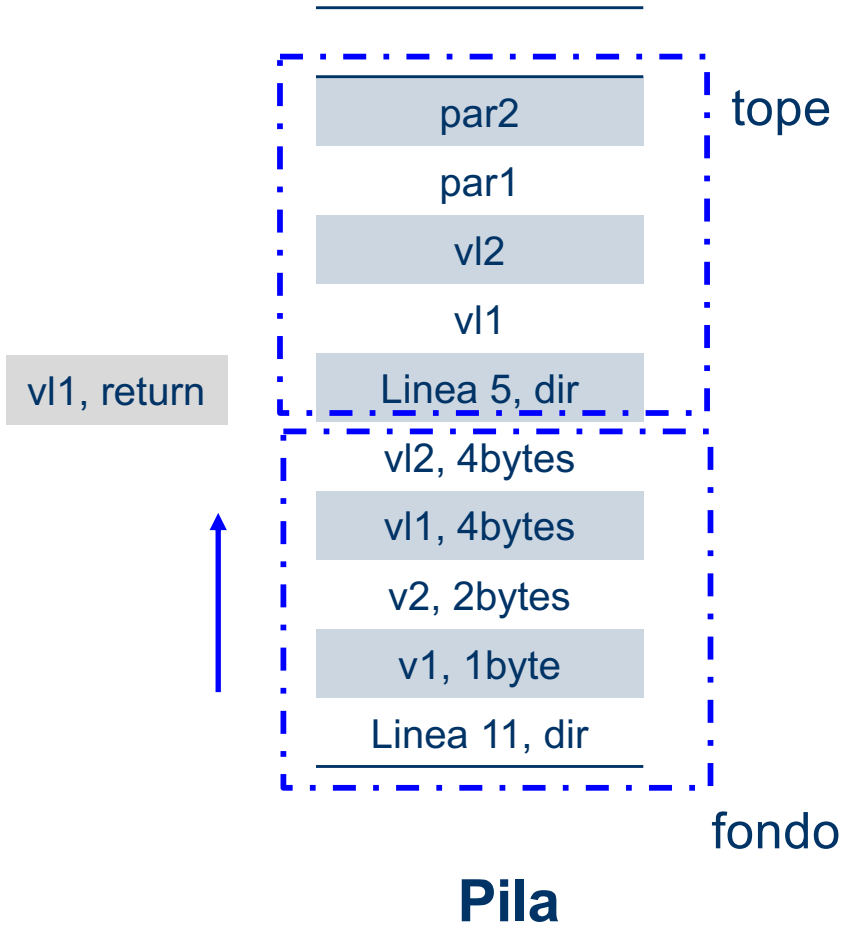
Segmentos Data Pointer (DP)	Datos (Data) Variables globales Heap malloc, new
Instruction Pointer (IP)	Código (Text) Funciones
Stack Pointer (SP)	Pila (Stack) Variables locales Parámetros

Base Pointer
Specific Pointer

Llamadas recursivas

Implementación

```
1   tipo func(char par1, int par2)
2   {
3    double vl1, vl2;
4         if( cond ) salir de la rec.
5         vl1= func(vl1, vl2);
6         return vl1;
7   }
8   main()
9   {
10     tipo var;
11        var = func(v1, v2);
12  }
```



Pila

En lenguaje C
NO se deben regresar variables locales (apuntadores).

Práctica

Modelo de función

Práctica optativa

- Verificar la forma en que se implementan las funciones con una pila:
  - Corroborar los datos de la pila y su orden.
  - Alterar dichos valores de manera indirecta.
  - Cambiar el tamaño máximo de la pila (según el compilador).
  - Etc.

Clasic topics

Temas clásicos

Notaciones polaca inversa (Reverse Polish Notation, RPN)

- Su nombre viene por analogía con la relacionada notación polaca, una notación de prefijo introducida en 1920 por el matemático polaco Jan Łukasiewicz.
- Es un método algebraico alternativo de introducción de datos donde cada operador está antes de sus operandos. En la notación polaca inversa es al revés.
- El esquema polaco inverso fue propuesto en 1954 por Burks, Warren y Wright y reinventado independientemente por Friedrich L. Bauer y Edsger Dijkstra a principios de los años 1960, para reducir el acceso de la memoria de computadora y para usar el stack para evaluar expresiones.
- La notación y los algoritmos fueron extendidos por el filósofo y científico de la computación australiano Charles Leonard Hamblin a mediados de los años 1960.
- Posteriormente, Hewlett-Packard lo aplicó por primera vez en la calculadora de sobremesa HP-9100A en 1968 y luego en la primera calculadora científica de bolsillo, la HP-35.
- Durante los años 1970 y 1980, el RPN tenía cierto valor incluso entre el público general, pues fue ampliamente usado en las calculadoras de escritorio del tiempo - por ejemplo, las calculadoras de la serie HP-10C.

Advanced topics

Sethi–Ullman algorithm (also known as Sethi–Ullman numbering)

- When generating code for arithmetic expressions, the compiler has to decide which is the best way to translate the expression in terms of number of instructions used as well as number of registers needed to evaluate a certain subtree.
  - Especially in the case that free registers are scarce, the order of evaluation can be important to the length of the generated code, because different orderings may lead to larger or smaller numbers of intermediate values being spilled to memory and then restored.
  - The Sethi–Ullman algorithm fulfills the property of producing code which needs the **fewest** instructions possible as well as the **fewest** storage references.
- NOTE
  - The algorithm succeeds as well if neither commutativity nor associativity hold for the expressions used, and therefore arithmetic transformations can not be applied.
  - The algorithm also does not take advantage of common subexpressions or apply directly to expressions represented as general directed acyclic graphs rather than trees.

Clasic topics

Temas clásicos

Práctica optativa

- Implementar un árbol con una tabla:
  - Opción 1: Tabla con nodos y apuntadores.
  - Opción 2: Tabla con hijo izquierdo y derecho.

3.3 Testing bipartiteness

Probando la bipartividad

Testing bipartiteness

Probando la bipartividad

Algorithm to test bipartiteness

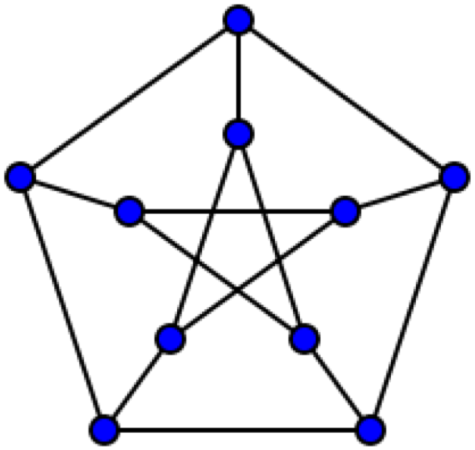
Ciclos impares no es bipartita.
Como adaptar BFS?

Using BFS

- Starting the search at any vertex and giving alternating labels to the vertices visited during the search.
  - That is, give label 0 (red) to the starting vertex, 1 (blue) to all its neighbors, 0 to those neighbors' neighbors, and so on.
- If at any step a vertex has (visited) neighbors with the same label as itself, then the graph is not bipartite.
- If the search ends without such a situation occurring, then the graph is bipartite.

Testing bipartiteness

Grafo de Petersen

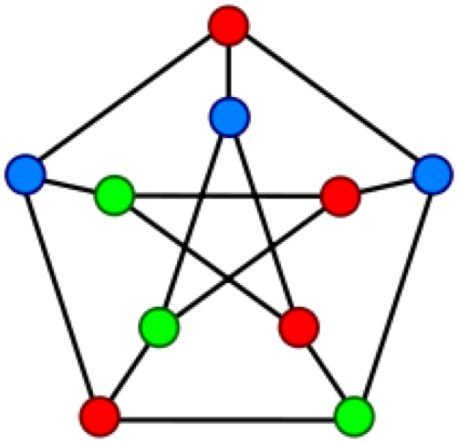


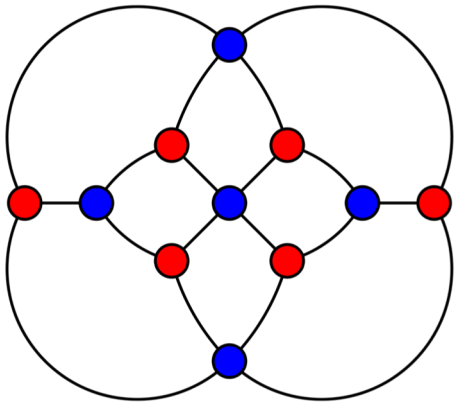
- In the mathematical field of graph theory, the Petersen graph is an undirected graph with 10 vertices and 15 edges.
- It is a small graph that serves as a useful example and counterexample for many problems in graph theory.
- The Petersen graph is named after Julius Petersen, who in 1898 constructed it to be the smallest bridgeless cubic graph with no three-edge-coloring.
- Although the graph is generally credited to Petersen, it had in fact first appeared 12 years earlier, in a paper by A. B. Kempe (1886).
- The Petersen graph is nonplanar. If planar it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.
- The Petersen graph is strongly regular and it is also symmetric, meaning that it is edge transitive and vertex transitive.

J. Petersen
Danish Mathematician
1839 - 1910

Testing bipartiteness

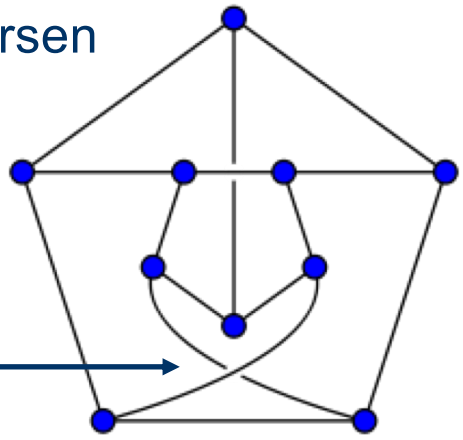
Ejemplos





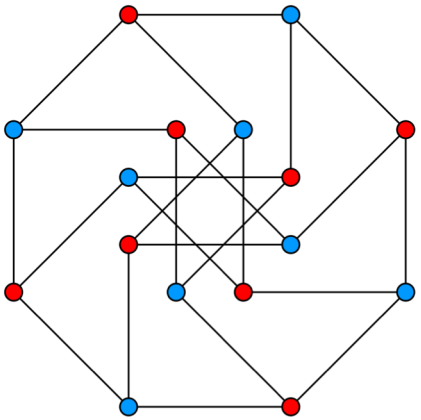
G. Hershel

G. Petersen



No planar

G. Möbius
Kantor



3.4 Connectivity in directed graphs

Conectividad en grafos dirigidos

Connectivity in directed graphs

Conectividad en grafos dirigidos

Direct graphs

G(V,E) such that edge (u,v) leaves node u and enters node v.

Graph search

Directed reachability

Given a node s, find **all nodes** reachable from s.

Directed s-t shortest path problem

Given **two nodes** s and t, what is the length of the shortest path from s and t?

BFS extends naturally to direct graphs.

Many aplications: web crawler (find all web pages linked to s).

Connectivity in directed graphs

Conectividad en grafos dirigidos

Strong connectivity

Definitions

- Nodes u and v are mutually reachable if there is a both path from u to v and also a path from v tu u.
- A graph is **strongly connected** if every pair of nodes is mutually reachable.

Lemma

Let s be any node, G is strongly connected iff every node is reachable from s, and s is reacheble from every node.

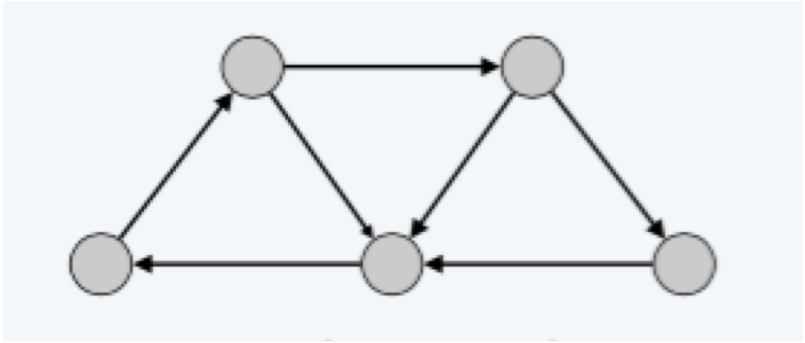
Connectivity in directed graphs

Conectividad en grafos dirigidos

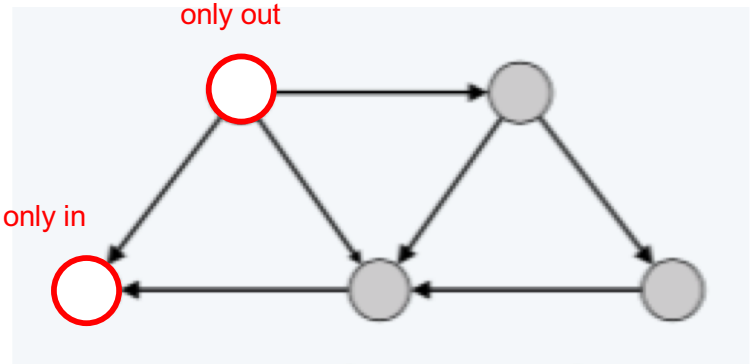
Theorem

One can determine if G es strongly connected in O(m+n) time.

Prove and give the algorithm:



strongly connected



NOT strongly connected

Connectivity in directed graphs

Conectividad en grafos dirigidos

Proof

- Pick any node *s*.
- Run BFS from *s* in *G*.
- Run BFS from *s* in *Greverse*.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

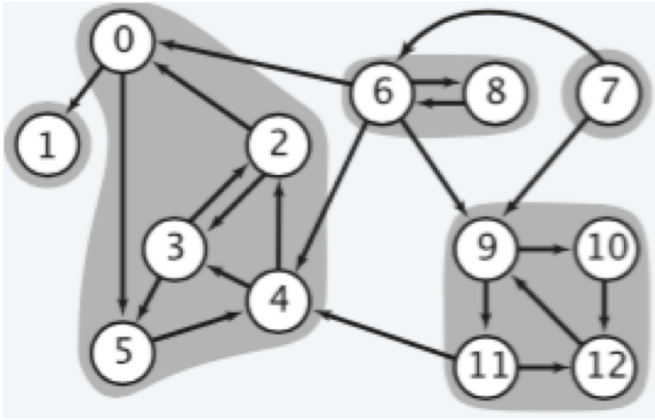
Remember that BFS is O(V+E)

Connectivity in directed graphs

Conectividad en grafos dirigidos

Definition

A **strong component** is a maximal subset of mutually reachable nodes.



- There are two algorithm to find strongly connected components:
  1. **Tarjan** theorem, 1972. You can find all strong components in $O(m + n)$ time.
  2. **Kosaraju**'s algorithm (1978) uses two passes of DFS.
- The **path-based strong component algorithm** uses a DFS but with two stacks (one with vertices not yet assigned, the other with the current path. The first linear time version was published by Edsger W. Dijkstra in 1976.

Connectivity in directed graphs

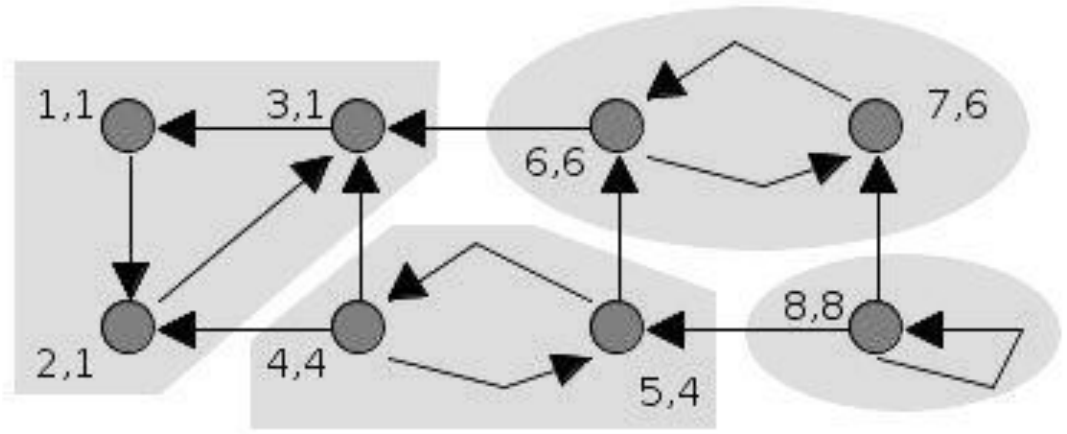
Conectividad en grafos dirigidos

Tarjan algorithm

For finding strong component using DFS. O(V+E)

Animation wikipedia:

Strongly
Connected
Component (**SCC**)



https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

Connectivity in directed graphs

Conectividad en grafos dirigidos

Algorithm tarjan **is**
input: graph $G = (V, E)$
output: set of SCC (sets of vertices)

index := 0
S := empty array

for each *v* **in** *V* **do**
  if (*v*.index is undefined) **then**
    strongconnect(*v*)
  end if
end for

Connectivity in directed graphs

Conectividad en grafos dirigidos

```
Function strongconnect(v) // Set the depth index for v to the smallest unused index
  v.index := index
  v.lowlink := index
  index := index + 1
  S.push(v)
  v.onStack := true

  // Consider successors of v
  for each (v, w) in E do
    if (w.index is undefined) then // Successor w has not yet been visited; recurse on it
      strongconnect(w)
      v.lowlink := min(v.lowlink, w.lowlink)
    else if (w.onStack) then
      // Successor w is in stack S and hence in the current SCC
      v.lowlink := min(v.lowlink, w.index)
    end if
  end for // If v is a root node, pop the stack and generate an SCC
```

Connectivity in directed graphs

Conectividad en grafos dirigidos

Continuation

```
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        w.onStack := false
        add w to current strongly connected component
      while (w != v)
      output the current strongly connected component
    end if
end function
```

3.5 Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

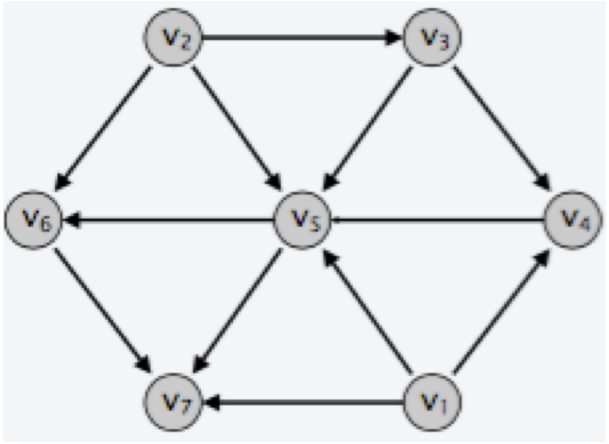
Definition

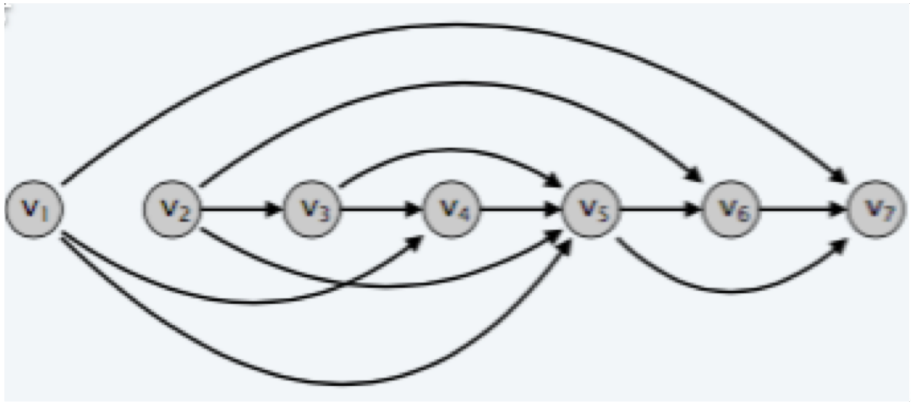
A **Directed Acyclic Graph** (DAG) is a directed graph that contains no directed cycles.

Definition

A **topological order** of a directed graph G = (V, E) is an ordering of its nodes as v_1, v_2, ..., v_n so that for every edge (v_i, v_j) we have i < j.

a DAG





Topological ordering

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

Lemma

If *G* has a topological order, then *G* is a DAG.

Question

- Does every DAG have a topological ordering?
- If so, how do we compute one?

Algorithm finds a topological order in $O(m + n)$ time.

Lemma

If *G* is a DAG, then *G* has a node with no entering edges.

Lemma

If *G* is a DAG, then *G* has a topological ordering.

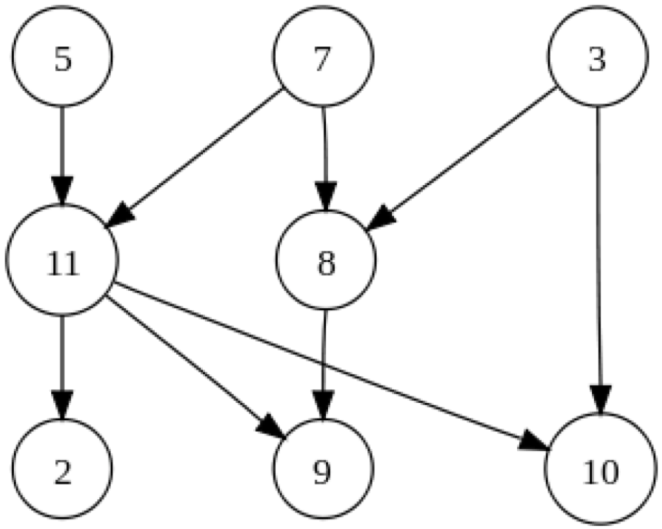
Proofs (4): Compilar prueba (explicación) en la tarea optativa.

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

Examples of topological ordering

The graph shown to the left has many valid topological sorts, including:



- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically O(V+E).

Algorithm 1

- One of first was described by **Kahn in 1962**.
- It works by choosing vertices in the same order as the eventual topological sort.
- First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph.

Algorithm 2

- An alternative algorithm for topological sorting is **based on DFS**.
  - Cormen 2001 y Tarjan 1976.
- The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when:

  it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e. a leaf node).

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

Kahn algorithm

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
While S is non-empty do
    remove a node n from S
    add n to tail of L
    For each node m with an edge e from n to m do
        remove edge e from the graph
        If m has no other incoming edges then
            insert m into S
If graph has edges then
    return error (graph has at least one cycle)
Else
    return L (a topologically sorted order)
```

Directed acyclic graphs and topological ordering

Grafos aciclicos dirigidos

DFS algorithm

L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark **do**
    select an unmarked node *n*
    visit(*n*)

function visit(node *n*)
    if *n* has a permanent mark **then**
      return
    if *n* has a temporary mark **then**
      stop *(not a DAG)*

    mark *n* with a temporary mark
    for each node *m* with an edge from *n* to *m* **do**
        visit(*m*)
    remove temporary mark from *n*
    mark *n* with a permanent mark
    add *n* to head of *L*

This depth-first-search-based algorithm is the one described by

- Cormen et al. (2001);
- it seems to have been first described in print by Tarjan (1976).

Otras implementaciones
https://www.geeksforgeeks.org/topological-sorting/
En C++, Java, Python, C#

Other problems

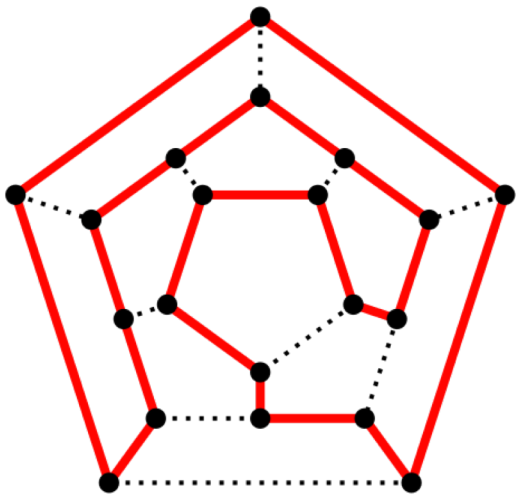
Otros problemas

Hamiltonian path problem

Caminos Hamiltonianos

1805-1865
Irish physicist, astronomer, and mathematician
Contributions to classical mechanics, optics, and algebra.

- A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once.
- A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle.
- Hamiltonian paths and cycle paths are named after William Rowan Hamilton who invented the icosian game.



However, despite being named after Hamilton, Hamiltonian cycles in polyhedra had also been studied a year earlier by Thomas Kirkman, who, in particular, gave an example of a **polyhedron** without Hamiltonian cycles.

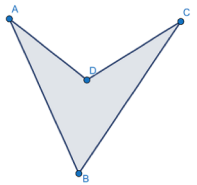
Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Hamiltonian path problem

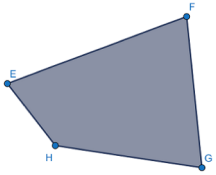
Caminos Hamiltonianos

Rompecabezas

- The game is now also known as Hamilton's **puzzle**, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron.
  - A dodecahedron is any polyhedron with twelve flat faces.
  - A polyhedron is a solid in three dimensions with flat polygonal faces, straight edges and sharp corners or vertices.
  - Cubes and pyramids are examples of polyhedra.
  - A polyhedron is said to be **convex** if its surface (comprising its faces, edges and vertices) does not intersect itself and the line segment joining any two points of the polyhedron is contained in the interior or surface.

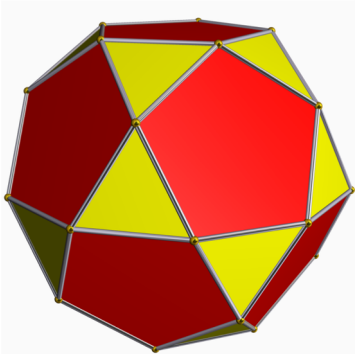


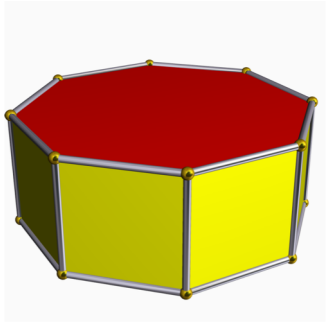
Cóncavo

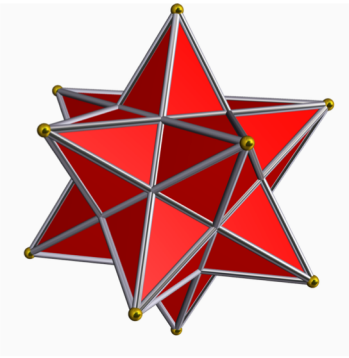


Convexo

Cuadrilátero







No es convexo

Hamiltonian path problem

Caminos Hamiltonianos

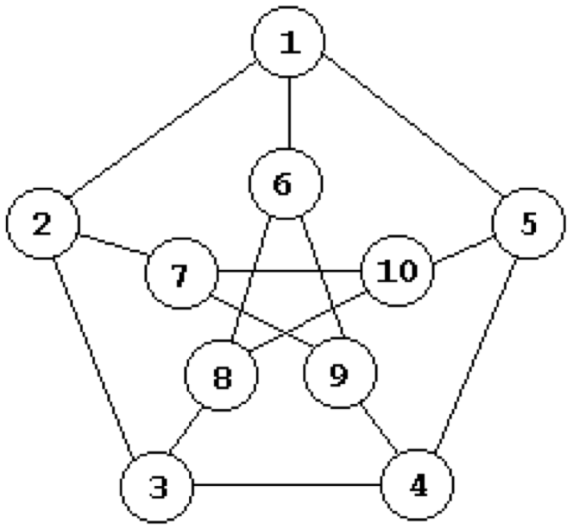
Algorithms

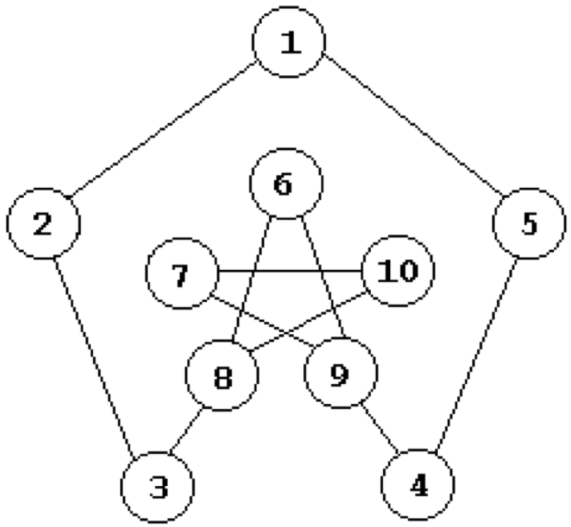
- The first algorithm for finding an Hamiltonian cycle on a directed graph was the enumerative algorithm of Martello.
- There are **n!** different sequences of vertices that might be Hamiltonian paths in a given n-vertex graph (and are, in a complete graph), so a brute force search algorithm that tests all possible sequences would be **very slow**.
- There are several “faster” approaches:
  - A successful algorithm for the undirected Hamiltonian path problem. 1985, Gerald L. Thompson, Sharad Singhal.
  - An algorithm for finding hamiltonian paths and cycles in random graphs, Bollobas, Fenner, y Frieze, 1985. https://www.math.cmu.edu/~af1p/Texfiles/AFFHCIRG.pdf.

Hamiltonian path problem

Caminos Hamiltonianos

- The Petersen graph has a Hamiltonian path but no Hamiltonian cycle.





Euler cycle

Grafos eulerianos

- Un ciclo euleriano o circuito euleriano es aquel camino que recorre todas las aristas de un grafo tan solo una única vez
- Es condición necesaria que regrese al vértice inicial de salida (ciclo = camino en un grafo donde coinciden vértice inicial o de salida y vértice final o meta).
- Se debe tener en cuenta que no importa la repetición de vértices mientras no se repitan aristas.

Algoritmo de Fleuri (1921)

1. Elegir un vértice arbitrario $v_1 \in V$. Definir $P_0 = (v_1)$, $G_0 = G$.
2. Si la ruta $P_k = (v_1, e_1, v_2, \ldots, e_{k-1}, v_k)$ ha sido construido, elegir $e_k \in E(G_{k-1})$ tal que (i) e_k incide con v_k. (ii) a menos que no haya otra elección e_k no es un puente de G_{k-1}. Si no existe tal k alto.
3. Definir $P_k := (P_{k-1}, e_k, v_{k+1})$, donde v_{k+1} es el otro extremo de e_k.
   Definir $G_k := G_{k-1} - e_k$.
4. Devolver $P := P_k$.

Example

El problema chino del cartero

- Propuesto en 1962 por el matemático chino Kuan.
- Cada día laboral, un cartero sale de la oficina postal y recorre todas las calles (aristas) de su zona para entregar el correo y luego regresa (ciclo) a la oficina postal. El cartero desea encontrar la ruta que le permita caminar lo menos posible.

Definición formal

- Sea G un multigrafo conexo y sea c : E(G) → R, con c(e) ≥ 0 para toda e ∈ E(G).
- Hallar un camino cerrado:

$$W = v_1\, e_1\, v_2 \ldots e_r\, v_{r+1}$$

- que pase por todas las aristas de G y tal que:

$$c(W) = \sum_{i=1}^{r} c(e_r)$$

  sea mínima.
- Si G es Euleriano entonces un circuito Euleriano es una solución óptima para el problema, en otro caso algunas aristas se recorren más de una vez.

Several topics

Varios temas

Notas históricas

Problemas abiertos

Tareas

URLs

A recordar

Open problemas

Problemas sin resolver

Some examples:

1. Método general para encontrar un ciclo hamiltoniano.

Look for another's:

- http://www.openproblemgarden.org/
- http://www.openproblemgarden.org/category/graph_theory

Recommended reading and Web sites

Lecturas recomendadas

Remember that this can change at any time (o estar fuera de linea)

Enciclopedias de matemáticas:

- http://www.encyclopediaofmath.org/index.php/Graph,_bipartite
- http://enciclopedia.us.es/index.php/Teoría_de_grafos
- http://mathworld.wolfram.com/Graph.html

- BFS
  - https://www.cs.usfca.edu/~galles/visualization/BFS.html
  - https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm
  - ***C++ implementation***: http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/

To remember

A recordar

Lo más importante es:

1. TODAS las definiciones de grafos.
2. Los algoritmos básicos de búsqueda (ancho, profundidad) que se usan en alguna variante para resolver otros problemas.
3. La forma en que se hacen pruebas con grafos.

List of Pbms, Alg, etc.

Lista de problemas

Pbm	Algoritmo
Graph isomorphism problem	Alg. Babai (quasipolynomial)
	Varios McKay, Schmidt & Druffel, Ullman
DFS	Iterativo, Recursivo
BFS	Iterativo, Recursivo
Finding Cut-Points	DFS, BFS
Finding all Connected Components	DFS, BFS
Transitive closure	Alg. Floyd-Warshall
Notaciones	DFS

Pbm	Algoritmo
Test bipartiteness	Red-blue / BFS
Strong conectivity	Proof BFS
Strongly Connected Components	Alg. Tarjan
	Alg. Kosaraju
	Alg. Dijkstra
Orden topológico	Alg. Kahn
	Alg. DFS
Camino Hamiltoniano	Martello O(n!)
Thomson, Bolobas	No los vimos
Ciclo Euleriano	Alg. Fleuri
Pbm. chino del cartero	

The end

Contacto

Raúl Acosta Bermejo

http:www.cic.ipn.mx
http://www.ciseg.cic.ipn.mx/

racostab@ipn.mx
racosta@cic.ipn.mx

57-29-60-00
Ext. 56652

Homeworks

Tareas

Resolver los problemas de:

1. Libro de Kleinberg 2003, Capítulo 2.
   5 problemas

   i. Lepidopterists (they study butterflies). Give an algorithm (design) with running time O(m + n) (restriction) that determines whether …
   ii. Connected graph G = (V,E) Computing DFS tree (obtain T) and BFS tree (obtain T). Prove that G = T.
   iii. Cut-points in a graph ... Give an algorithm that does this so that it still runs in O(m + n), and it terminates with stored values for earliest.