Informatique Fondamentale IMA S8

Cours 4 : graphs, problems and algorithms on graphs, (notions of) NP completeness

Laure Gonnord

http://laure.gonnord.org/pro/teaching/

Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

2012

IV - Graphs, problems and algorithms on graphs, NP-completeness

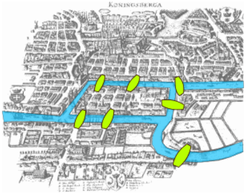
Goal

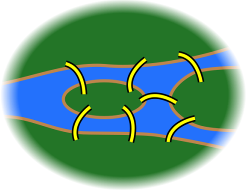
Model :

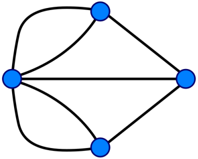
- **relationships between objects** (“I know him”, “This register is in conflict with with one”, “there is a road between these two towns”)
- classical problems of paths (“Do I have a friend that knows a friend ..... who knows the Dalai-Lama ?”, “It is possible to go from this town to this another in less than 600 km ?”, “how to get out of the labyrinth ?”)

Who ?

Euler (1740) formalised the theory and the Konisberg problem :







Problem : compute an **eulerian path**.

Source (fr) : `http://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes`

Other classical problems

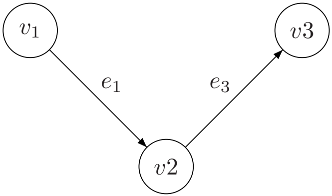
- The travelling salesman problem : TSP
- Task scheduling (on parallel machines or not !), the PERT method.
- Coloring maps
- Covering (telecom) trees
- Minimal paths ...

▶ Graphs are everywhere.

General definition

Graph

- A finite set of vertices (a vertex) : V
- A finite set of edges. $E \subseteq V \times V$

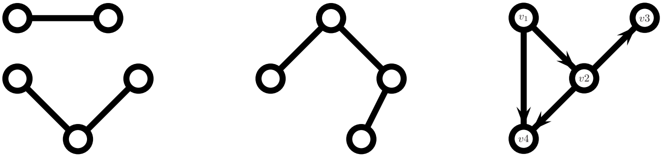


▶ A simplier model than the previous ones !

Other definitions

- Non Oriented Graph iff
  $\forall(v1, v2), (v1, v2) \in E \Rightarrow (v2, v1) \in E$.
- Tree (non oriented acyclic connex) - Forest .
- Directed Acyclic Graph : DAG . Oriented Graph with no cycle.

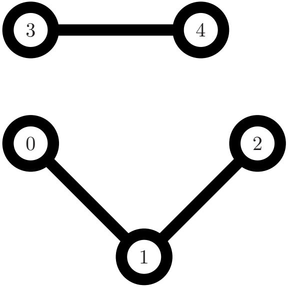
▶ trees are dags !



Sources

Picture and Java codes from
`http://pauillac.inria.fr/~levy/courses/X/IF/a1/a1.html`

Adjacency matrix



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- If the graph is oriented, then the matrix is NOT symmetric
- Memory complexity : $O(V^2)$

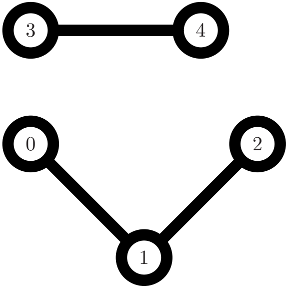
Adjacency matrix - Java

```
boolean[ ][ ] m;

Graph (int n) {
  m = new boolean[n][n];
   for (int i = 0; i < n; ++i)
     for (int j = 0; j < n; ++j)
       m[x][y] = false;
}

static void addEdge (Graph g, int x, int y)
         { g.m[x][y] = true;
           g.m[y][x] = true //only if non oriented
```

Array of lists



- Memory complexity : $O(V + E)$

Array of lists - Java

```
   Liste[ ] succ;

   Graph (int n) { succ = new Liste[n]; }

   static void addEdge (Graph g, int x, int y) {
     g.succ[x] = new Liste (y, g.succ[x]);
     // if non oriented also modify g.succ[y] !
   }
}
```

Search Algorithms

! Translation problem ! (en) Search Algorithms - (fr) Parcours de Graphes

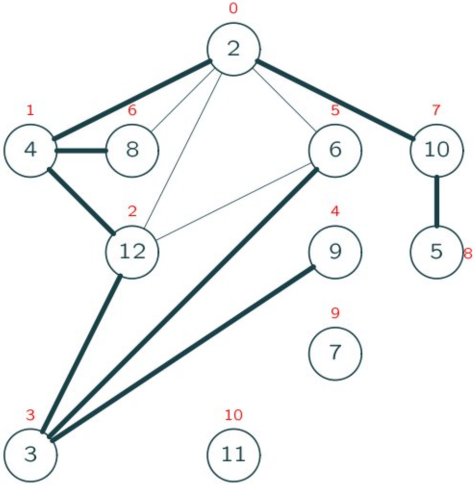
Reminder : search/do algorithm on a **binary tree** :

1. If the tree is not empty, do $f(root)$.
2. recursively apply the function on the left sub-tree
3. recursively apply the function on the right sub-tree

- Deep-First Search

Deep First Search on Graphs

On graphs, there can be cycles !



Deep First Search on Graphs - Java

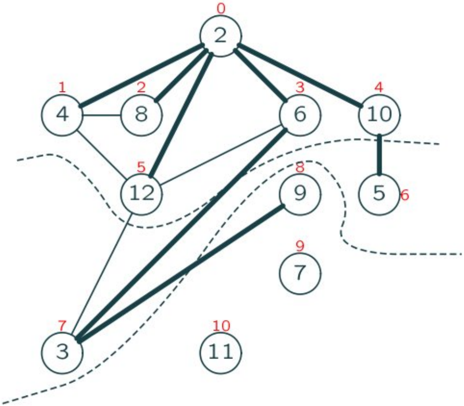
```
static int[ ] num; // numbering the edges according to the visiting or

static void visiter (Graphe g) {
  int n = g.succ.length;
  num = new int[n]; numOrdre = −1;
  for (int x = 0; x < n; ++x)  num[x] = −1; // −1 = ''not seen''
  for (int x = 0; x < n; ++x)
    if (num[x] == −1)
      dfs(g, x);
}

static void dfs (Graphe g, int x) {
  num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (num[y] == −1)
      dfs(g, y);
  }
}
```

- Complexity = $O(V + E)$. Tarjan [1972]

Breadth First Search on Graphs



Conventions

BLANC = not (yet) processed NOIR = has been processed
GRIS = being processed

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;
static int[ ] couleur;
```

Breadth First Search on Graphs - Java

```
 int n = g.succ.length; couleur = new int[n];
  FIFO f = new FIFO (n);
  for (int x = 0; x < n; ++x)  couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC) {
      FIFO.ajouter(f, x); couleur[x] = GRIS;
      bfs (g, f);
   }

static void bfs (Graphe g, FIFO f) {
  while ( !f.vide() ) {
    int x = FIFO.supprimer (f);
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (couleur[y] == BLANC) {
        FIFO.ajouter(f, y); couleur[y] = GRIS;
      }
      couleur[x] = NOIR;
} } }
```

- Complexity ?

Application : Labyrinth

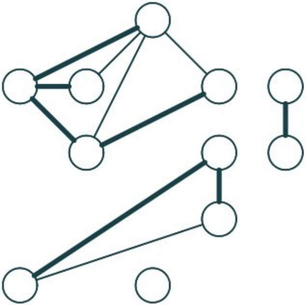
Finds exit , ie a path from d to s ?

```
  couleur[d] = GRIS;
  if (d == s)
    return new Liste (d, null);
  for (Liste ls = g.succ[d];
          ls != null; ls = ls.suivant) {
    int x = ls.val;
    if (num[x] == BLANC) {
      Liste r = chemin (g, x, s);
      if (r != null)
        return new Liste (d, r);
    }
  }
  return null;
}
```

Definition - Connex Component

Connex component

A **connex component** is a maximal set of connected vertices.



- Exercise : Algorithm for computing CCs.

Definition - Connex Graph

Connex graph

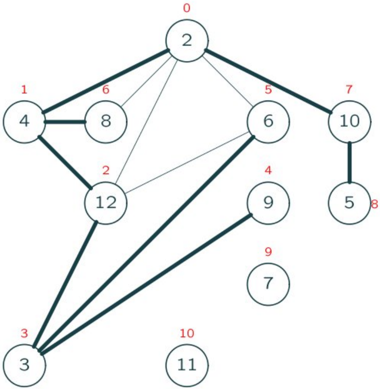
A graph is **connex** if it has only one connex component.

- Exercise : Algorithm that tests the connexity of a graph.

Other Algorithms

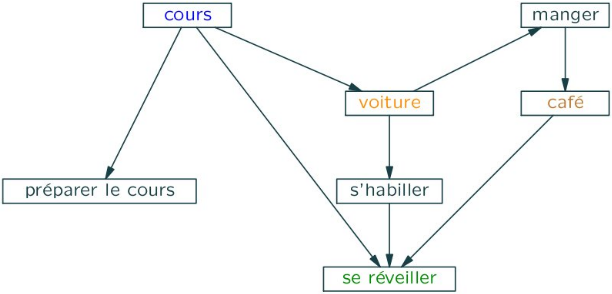
There exists variants of DFS algo to :

- Compute Cover Forets (in bold in the picture) :



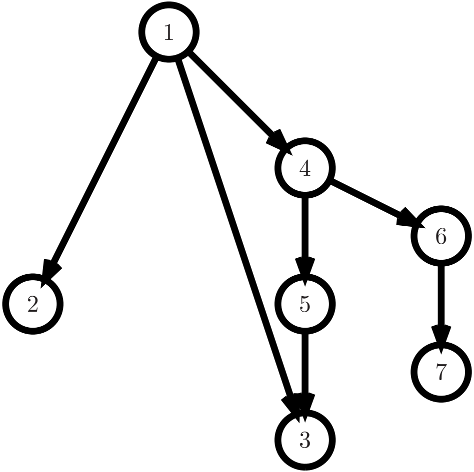
- Detect Cycles.

Topological sorting of a DAG



- Give a list of vertices in an order compatible with the DAG order : $v_i < v_j \Rightarrow v_j \rightarrow v_i$ (total order from a partial header).

Topological sorting of a DAG - Example



- **If** we know an entrant node, a **dfs** gives the result.

Topological sorting of a DAG - implementation

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;

static Liste aFaire (Graphe g, int x) {
  int n = g.succ.length;
  int[ ] couleur = new int[n];
  for (int x = 0; x < n; ++x)  couleur[x] = BLANC;
  return dfs(g, x, null, couleur);
}

static Liste dfs(Graphe g, int x, Liste r, int[ ] couleur) {
  couleur[x] = GRIS;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      r = dfs(g, y, r, couleur);
  }
  return new Liste (x, r);
}
```

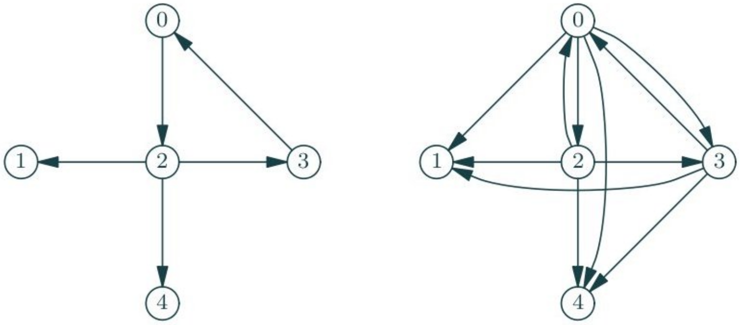
- **dfs** here returns a list.

Topological sorting of a DAG - Applications

- instruction scheduling
- cell evaluation in spreadsheets
- order of compilation tasks (Makefiles)
- symbol dependencies in linkers
- sheduling in project managements (PERT, 1960)

`http://en.wikipedia.org/wiki/Topological_sorting`

Transitive closure



The graph $G^+ = (V, E^+)$ of the transitive closure of $G = (V, E)$ is such that E^+ is the smallest set verifying :

- $E \subseteq E^+$
- $(x, y) \in E^+$ and $(y, z) \in E^+ \Leftrightarrow (x, z) \in E^+$

Algorithm for Transitive closure

If E is the adjacency matrix : $E^{+} = E + E^{2} + \ldots E^{n-1}$.

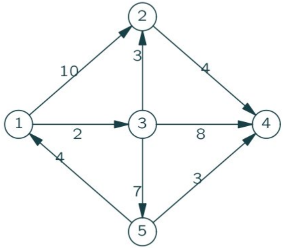
- An algorithm in $O(n^4)$.

- Is it possible to have a better complexity ? Yes (Warshall, in $O(n^3)$, see later).

Definition of oriented valuated graph

Valuated graph

It's a graph with a function $d : V \times V \to \mathbb{N}$ called **distance**.



- G is represented by an adjacency matrix with integer values, or $+\infty$ if there is no edge.

The shortest path problem

Given G, find a path whose sum of distances is minimal !

more variants on `http://en.wikipedia.org/wiki/Shortest_path_problem`

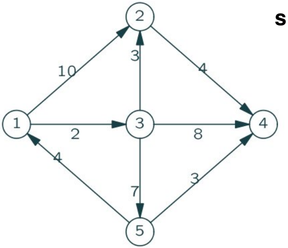
Multiple Source Shortest Path with dynamic programming -1

Between all the vertices ! Floyd-Warshall algorithm
Principle : Let $d^k_{x,y}$ be the minimal distance between x and y passing through vertices whose number is $< k$. Then

$$d^0_{x,y} = d_{x,y} \text{ and } d^k_{x,y} = min\{d^{k-1}_{x,y}, d^{k-1}_{x,k} + d^{k-1}_{k,y}\}$$

- Store all these temporary distances.

Multiple Source Shortest Path with dynamic programming -2




```
static Graphe plusCourtsChemins (Graphe g) {
  int n = g.d.length;
  Graphe gplus = copieGraphe (g);
  for (int k = 0; k < n; ++k)
   for (int x = 0; x < n; ++x)
    for (int y = 0; y < n; ++y)
     gplus.d[x][y] = Math.min (gplus.d[x][y],
                  gplus.d[x][k] + gplus.d[k][y]);
  return gplus;}
```

- Complexity $O(n^3)$, space $O(n^2)$.

Simple Source Shortest Path

- All d are > 0 : Dijkstra, 1959
- Negative distances : Bellman Ford $\simeq$ 1960

▶ Routing algorithms (Bellman uses only local information).

Eulerian and Hamiltonian paths

Eulerian/Hamiltonian Path

- An Eulerian path : each edge is seen only once.
- An Hamiltonien path : each node ...

Hamiltonian Tour, one solution

- Extend a path until not possible
- Backtrack one time, and try with another neighbour
- And so on.

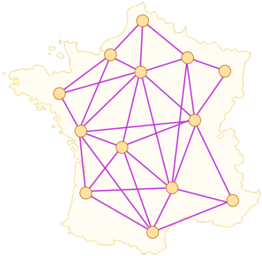
► Worst case complexity time : exponential in n

Application 1 : Knight's Tour

Is it possible to make a Knight's tour ? « The knight is placed on the empty board and, moving according to the rules of chess, must visit each square exactly once.»

▶ A variant of the Hamiltonian tour that can be solved in polynomial time.

The Travelling Salesman Problem



with valuated (non oriented) graph.

- Find an hamiltonian cycle of minimum weight

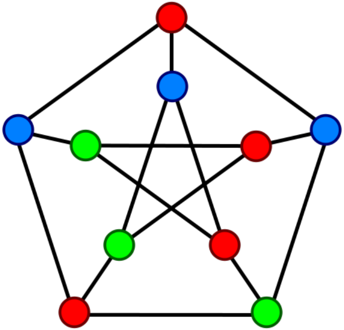
Travelling Salesman Problem - 2

A (non-polynomial) algorithm for this problem :

- Transform into a linear programming problem with integers
- Solve it with the simplex !

▶ This algorithm is **exponential in the size of the graph**

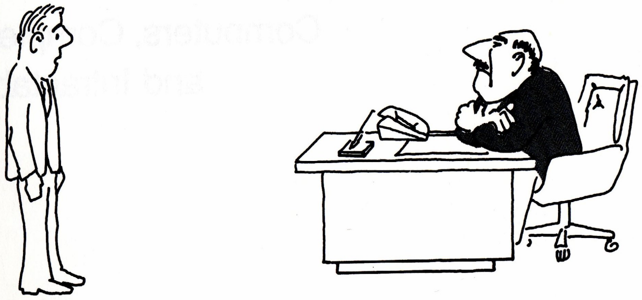
Graph Coloring Problem



- Application to the **register allocation** in compilers.
- The sudoku problem (9-coloring of a 81-vertices graph)

Graph Coloring Problem - 2

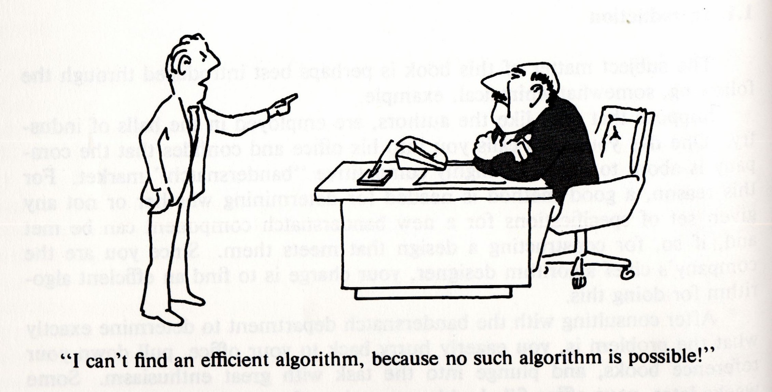
Are **you** able to design a polynomial algorithm ?



"I can't find an efficient algorithm, I guess I'm just too dumb."

Graph Coloring Problem - 3

We do not know any polynomial algorithm for this problem.



Complexity

The complexity of an implementation / an algorithm is linked to Turing machines :

- The number of steps in the execution of a TM gives the **complexity in time**,
- The number of seen squares in the execution of a TM gives the **complexity in space**.

We can replace the TM by “a C implementation”.

P Problems

P

A **problem** (or a language) belongs to P if there exists a **deterministic** Turing Machine that gives the result in **polynomial time**.

- there is a polynom p such that for all entry x, the Turing machine stops (with the good result) in less than $p(|x|)$ steps.

Example : $L = a^n b^n c^n$, the TM of the previous course checks any $a^i b^i c^i$ in $O(n)$ steps

P Problems

- Integer multiplication
- Eulerian tour (each edge once)
- Linear programming (recall that polynomial diophantine equations are **undecidible**.)
- Primality test

http://www.cs.uu.nl/groups/AD/compl-diaz.pdf

NP Problems

NP

A **problem** (or a language) belongs to NP if there exists a **deterministic** Turing Machine that checks the result in **polynomial time**.

Example : there exists a TM that is able to check if a given path is an hamiltonien tour, in polynomial time.

NP Problems - Examples

- Factoring
- 3-SAT (formulae in CNF with 3 litterals on each term)
- Hamiltonian tour
- 3-colorability

`http://www.cs.uu.nl/groups/AD/compl-diaz.pdf`

NP vs P - some facts

- If a problem belongs to NP, then there exists an exponential brute-force deterministic algorithm to solve it (easy to prove)
- $P \subseteq NP$ (trivial)

▶ P = NP ? P $\neq$ NP ? Open question ! Problem of the millenium !
Is finding more difficult that verifying ?

NP-complete problems - 1

Given a **problem** :

- if we find a polynomial algorithm ▶ P
- if we find a polynomial time checking algorithm ▶ NP, but we want to know more :

"it it really hard to solve it ?"
"is it worth looking for a better algorithm ?"

NP-complete problems - 2

The key notion is the notion of polynomial reduction

Polynomial reduction of problems

Given two problems P_1 and P_2, P_1 reduces polynomially to P_2 if there exists an f such that :

- f is polynomial
- x_1 solution of P_1 iff $f(x_1)$ solution of P_2.

Example : Hamiltonian circuit is polynomially reductible to TSP.

NP-complete problems - 3

The most difficult problems in NP. All problems in NPC are **computationally equivalent** in the sense that, if one problem is easy, all the problems in the class are easy. Therefore, if one problem in NPC turns out to be in P, then

P=NP

NP-complete problems - 4

Reminder : we want to characterise the most difficult problems in NP.

NP-complete

A problem is said to be NP-complete if :

- It belongs to NP
- All other problems in NP **reduce polynomially** to it.

In other words, if we solve any NP-complete in polynomial time, we have proved $P = NP$ (and we become rich)

NP-complete problems - 4

Problem : how to prove that a problem is NP-complete ?

- The very first one is hard to prove
- Then, we use the following result :

> If two problems / languages L_1 and L_2 belong to NP and L_1 is NP-complete, and L_1 reduces polynomially to L_2, then L_2 is NP-complete.

▶ Pick one NP complete problem and try to reduce it to **your** problem.

NP-complete problems - The historical example

- SATISFIABILITY is NP-complete (Cook, 1971)
- SAT reduces polynomially to 3SAT
- 3SAT reduces polynomially to Vertex Cover
- Vertex Cover reduces polynomially to Hamiltonian circuit

► “So”Hamiltonian circuit, then TSP are also **NP-complete** problems

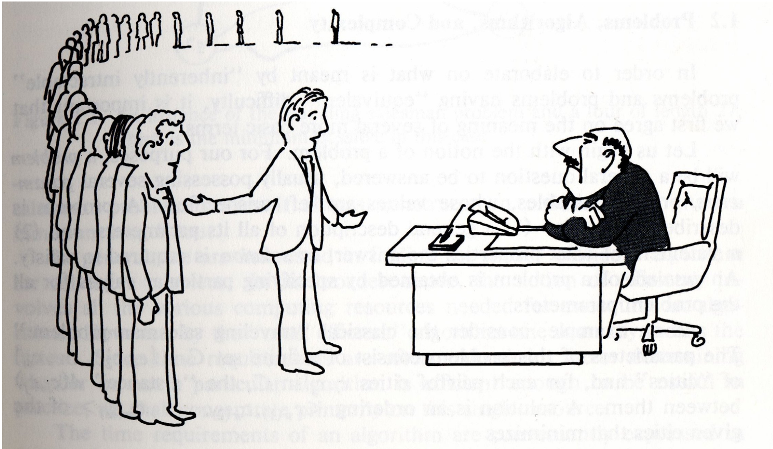
Some NP-complete (common) problems

Some classical examples :

- Hamiltonian cycle and TSP
- Graph Coloring Problem
- Vertex cover
- Knapsack (integer)
- Flow shop problem
- Sudoku

The book : Computers and Intractability : A Guide to the Theory of NP-Completeness. A short list on the french webpage : http://fr.wikipedia.org/wiki/Liste_de_probl%C3%A8mes_NP-complets

Conclusion - 1



"I can't find an efficient algorithm, but neither can all these famous people."

Conclusion - 2

Knowing that a given problem is NP complete is just the beginning :

- handle less general classes of inputs
- compute less precise solutions

More docs on graphs

- In french : http://laure.gonnord.org/site-ens/mim/graphes/cours/cours_graphes.pdf or type “cours de Théorie des graphes” in Google
- In english : an interactive tutorial here : http://primes.utm.edu/cgi-bin/caldwell/tutor/graph/intro.
- (en) The theorical book “Graph Theory”, R. Diestel (electronic version : http://diestel-graph-theory.com/basic.html)
- (en) e-book for algorithms : http://code.google.com/p/graph-theory-algorithms-book/

More docs on complexity

The Book : computers and intractability, a guide to the theory of NP completeness, by Garey/Johnson