Graphs' decompositions
and
resolutions of combinatorial problems

Stéphane Secouard
Supervised by : Florent Madeleine

Caen University - Computer science

25 october 2016

Table of contents

Contents

Objectives

Objectives (of my project)

- *The first goal of this project is, starting from a graph whose tree representation is known, to solve corresponding combinatorial problems.*
  The k-coloring problem, the max clique problem or the Hamilton path problem can be explored.
- *The second purpose is to implement a graph decomposition calculator.*
- *Finally, it can be considered extensions by working on the efficiency of implementations on large-size structures, or improving the shape of displayed results.*

Contents

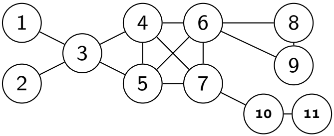
Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

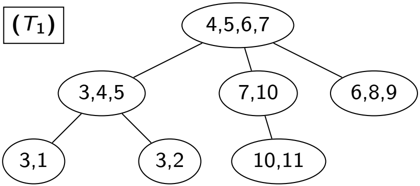
1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :

(G)



(T_1)

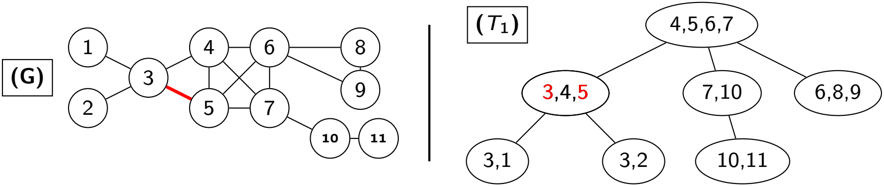


Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :

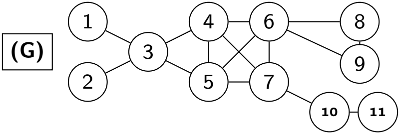


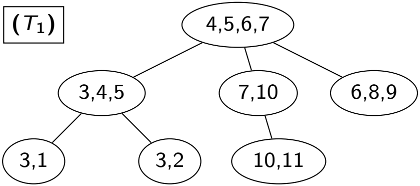
Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :





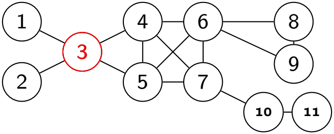
Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

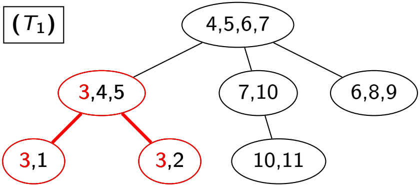
1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :

(G)



(T_1)

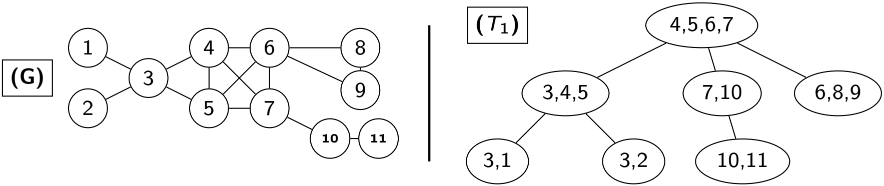


Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :

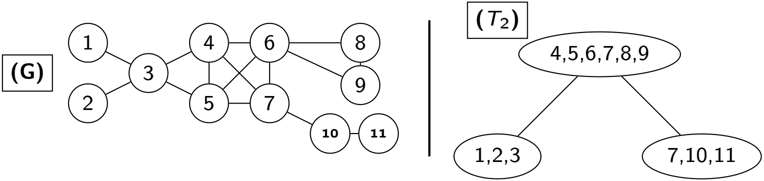


Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :



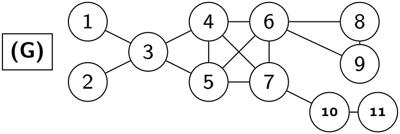
Tree decomposition of a graph

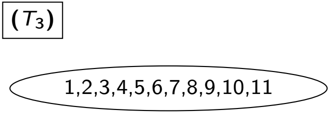
Definition (Tree decomposition of a graph)

A tree T is a tree decomposition of a graph G where its nodes are arranged satisfying the following properties :

1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

example :





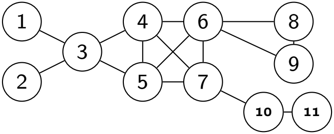
Treewidth

Definition (treewidth)

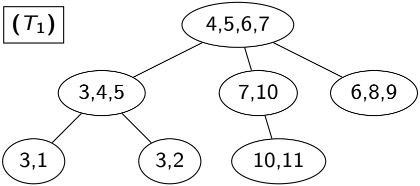
- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

(G)



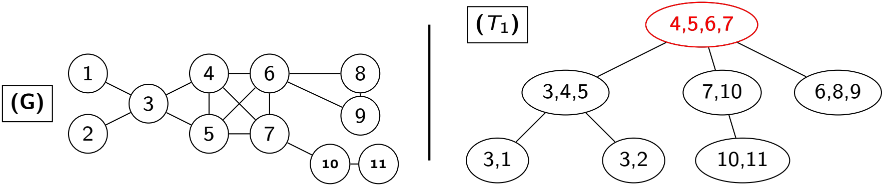
(T_1)



Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

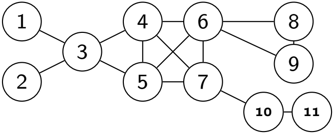


Definition (treewidth)

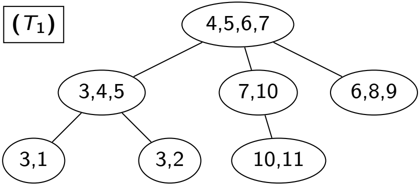
- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

(G)



(T_1)

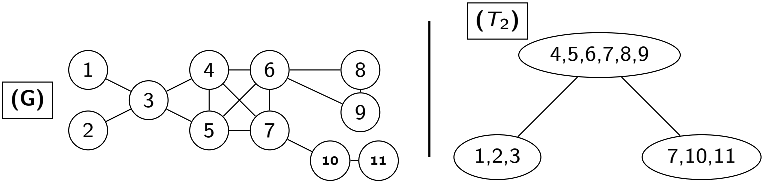


width(T_1)=4-1=3

Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

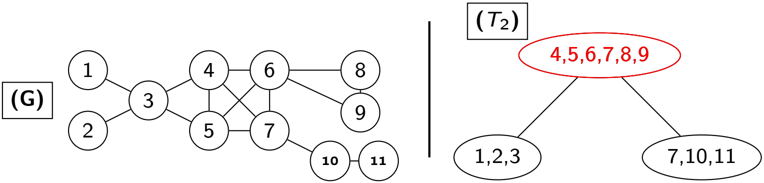
example :



Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

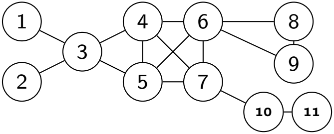


Definition (treewidth)

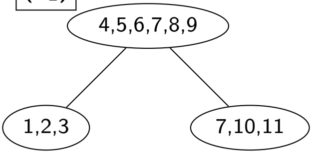
- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

(G)



(T_2)

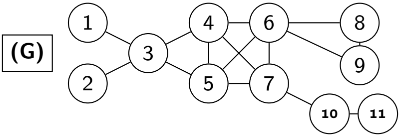


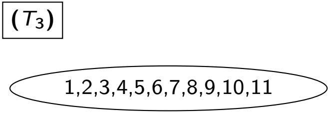
width(T_2)=6-1=5

Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :



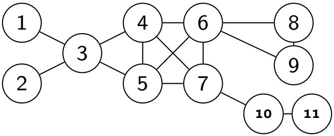


Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :

(G)



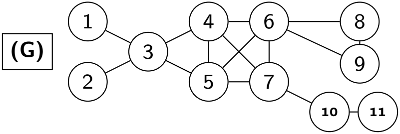
(T_3)

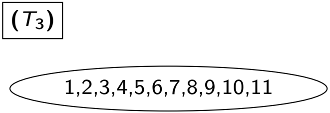
Treewidth

Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions

example :



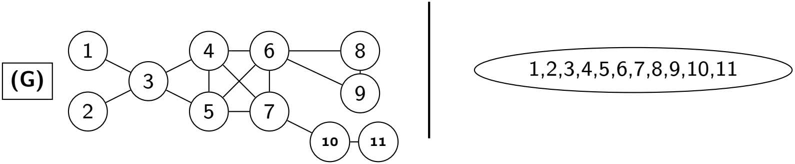


width(T_3)=11-1=10

Definition (treewidth)

- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions of this graph.

example :

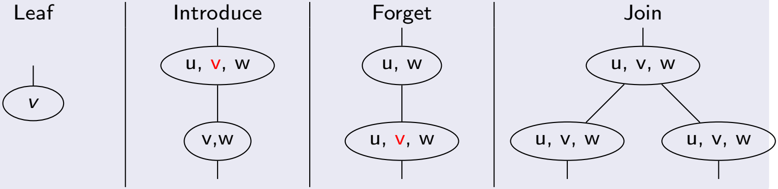


$\boxed{\text{TreewidthG=Min(width(}T_k\text{)=3}}$

Nice tree

Definition (nice tree)

A tree decomposition is *nice* if every node is one of the following 4 types :

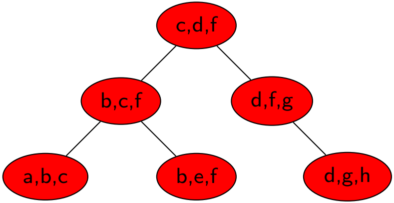


Remark

- *A tree decomposition can be turned into a nice tree decomposition*
- *A nice tree can be used to simplify a proof, or to find an easy program to solve a problem (as we will see later on).*

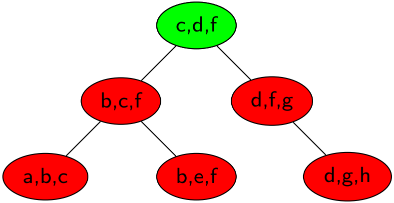
How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.



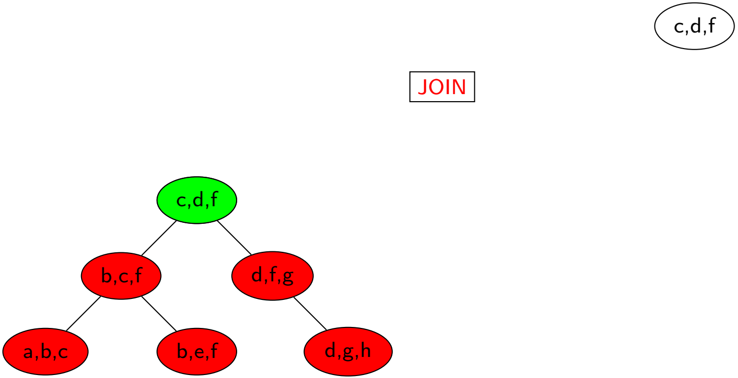
How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.



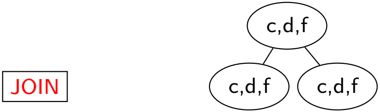
How to obtain a nice tree ?

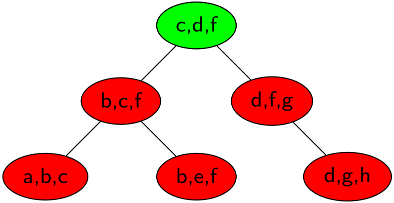
This is a tree decomposition of the graph seen previously.



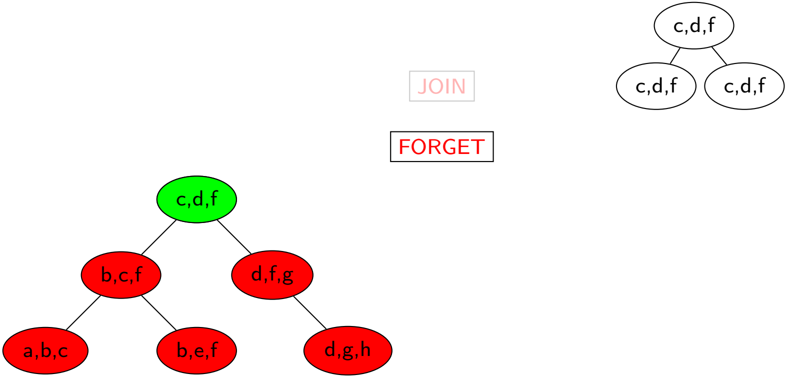
How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.

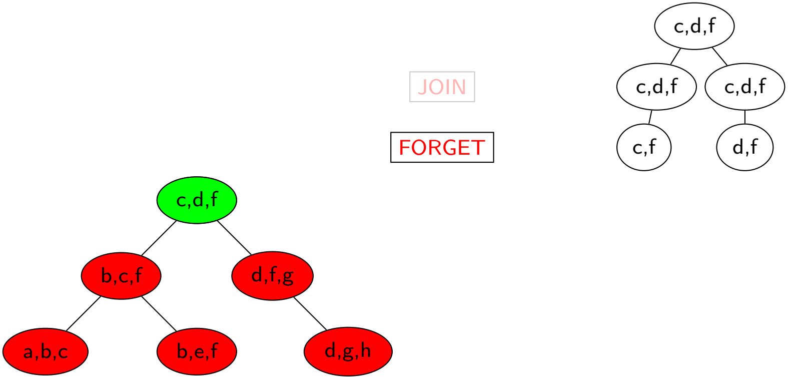




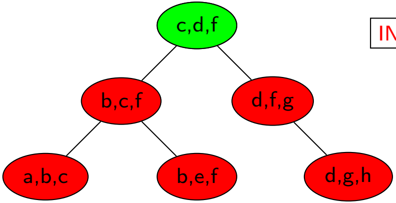
This is a tree decomposition of the graph seen previously.



This is a tree decomposition of the graph seen previously.



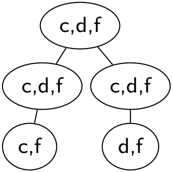
This is a tree decomposition of the graph seen previously.



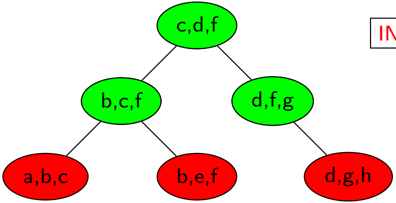
JOIN

FORGET

INTRODUCE



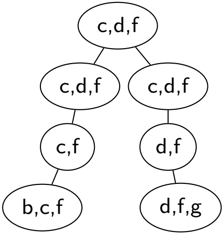
This is a tree decomposition of the graph seen previously.



JOIN

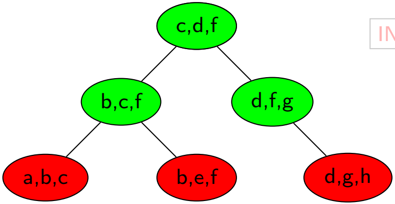
FORGET

INTRODUCE



How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.

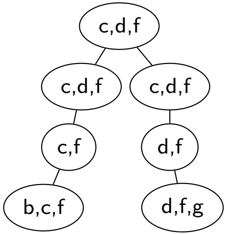


JOIN

FORGET

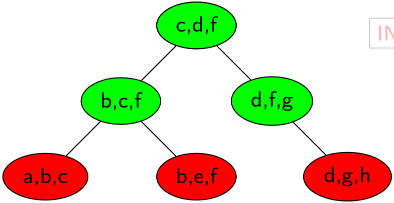
INTRODUCE

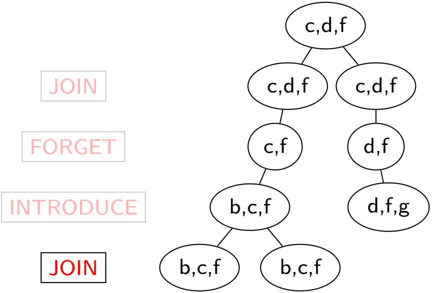
JOIN



How to obtain a nice tree?

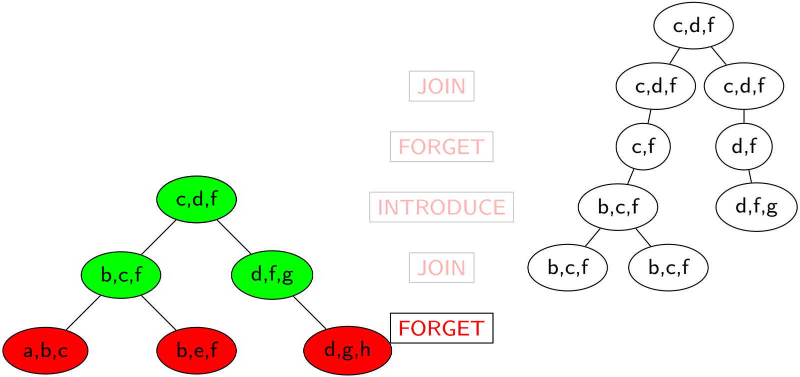
This is a tree decomposition of the graph seen previously.





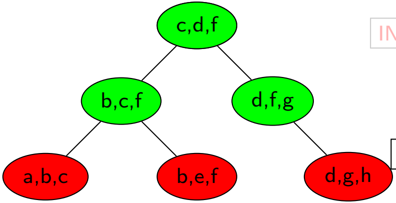
How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.



How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.



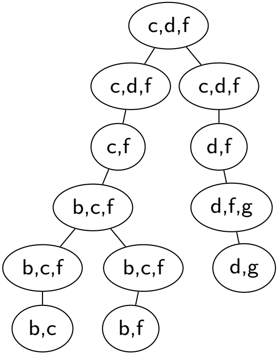
JOIN

FORGET

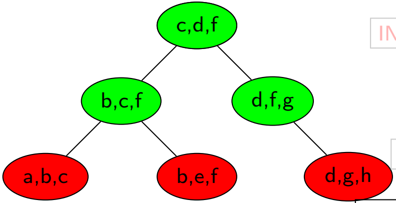
INTRODUCE

JOIN

FORGET



This is a tree decomposition of the graph seen previously.



JOIN

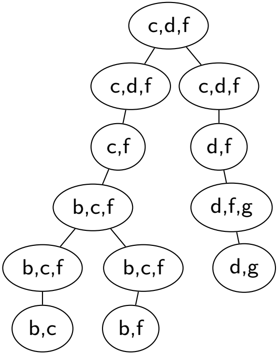
FORGET

INTRODUCE

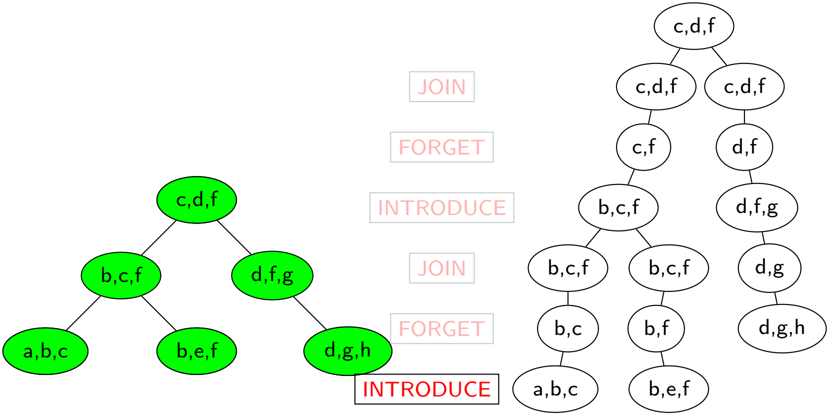
JOIN

FORGET

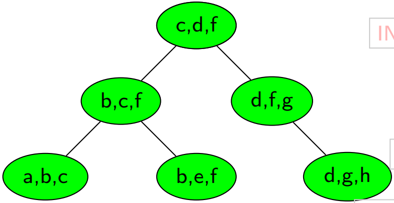
INTRODUCE

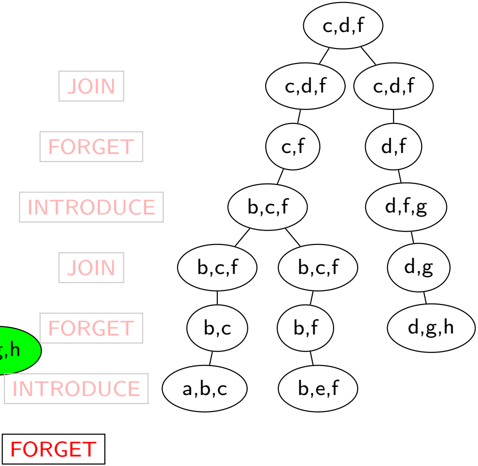


This is a tree decomposition of the graph seen previously.

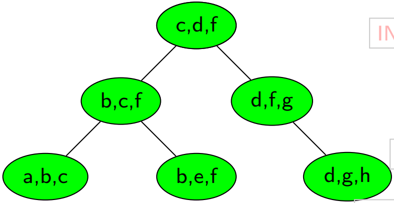


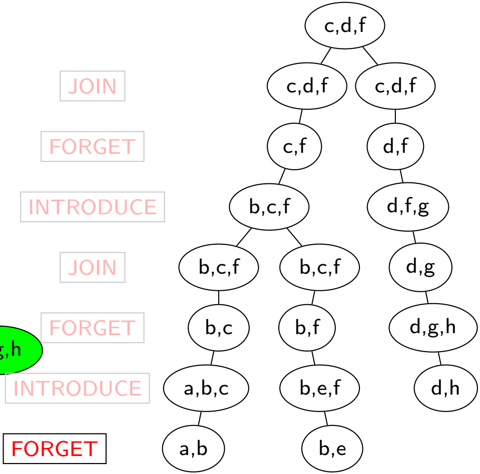
This is a tree decomposition of the graph seen previously.





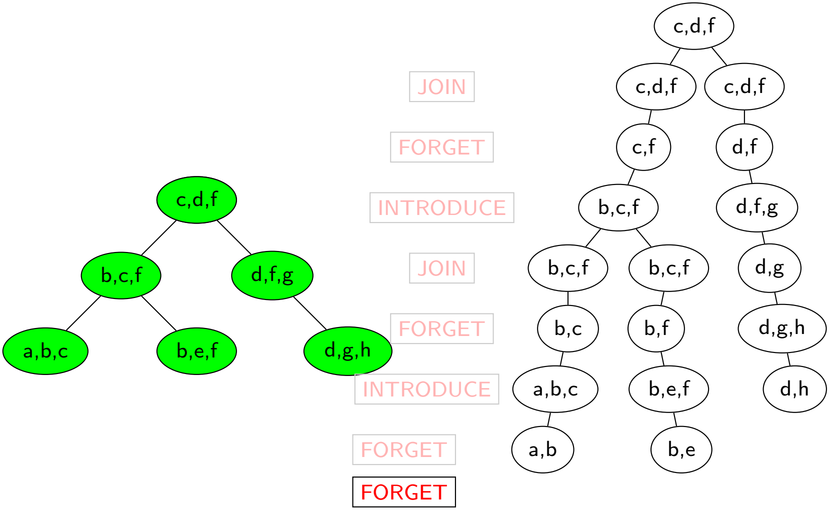
This is a tree decomposition of the graph seen previously.





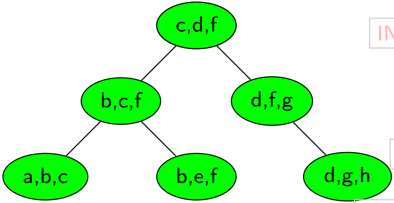
How to obtain a nice tree ?

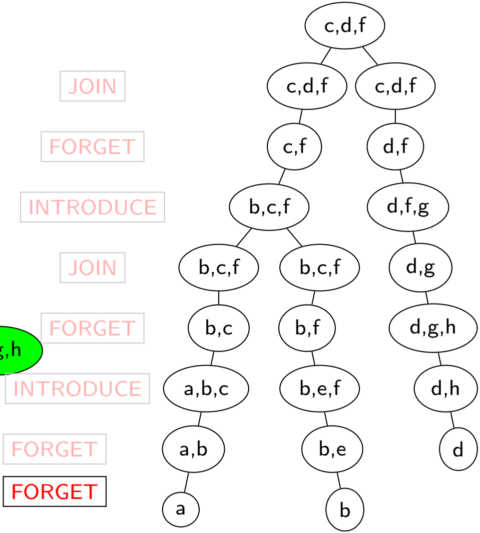
This is a tree decomposition of the graph seen previously.



How to obtain a nice tree ?

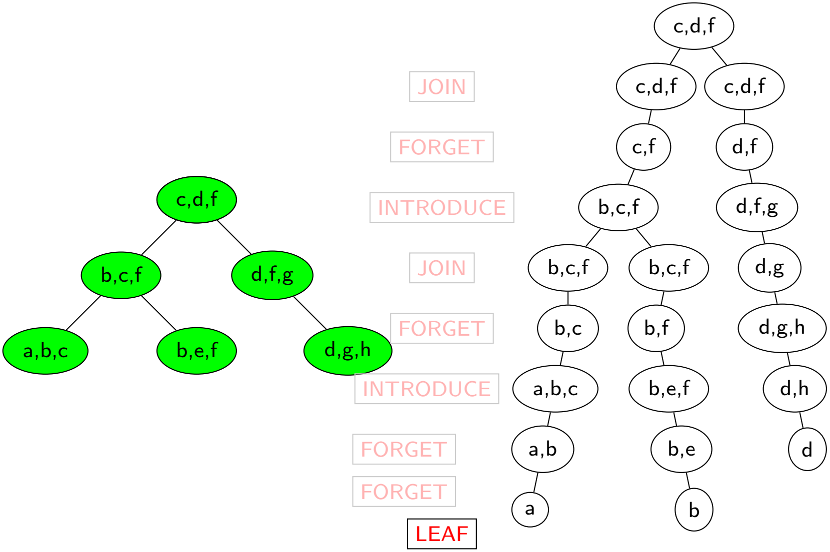
This is a tree decomposition of the graph seen previously.





How to obtain a nice tree ?

This is a tree decomposition of the graph seen previously.



Contents

the problem

Problem (k-color)

Let (G) be a graph and k an integer. We want to know if it is possible to draw each vertice of the graph so that two neighbors have never the same color and with only k colors.
This problem is a problem of decisions problem which is NP-complet.

Solution

tree-width : *k-color is possible for a graph (G) if and only if $k >$ treewidth(G).*

nice tree : *a nice tree of (G) gives a way to find a k-coloration of (G) (if $k >$ treewidth(G)).*

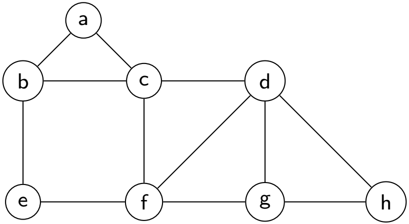
Problem (k-color)

Let (G) be a graph and k an integer. We want to know if it is possible to draw each vertice of the graph so that two neighbors have never the same color and with only k colors.
This problem is a problem of decisions problem which is NP-complet.

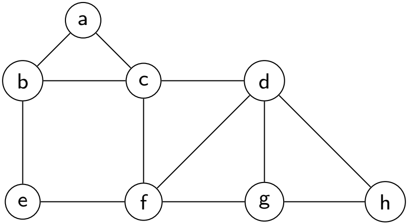
Solution

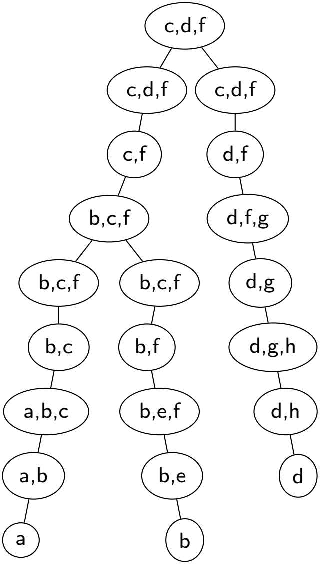
- tree-width : *k-color is possible for a graph (G) if and only if* $k > treewidth(G)$.
- nice tree : *a nice tree of (G) gives a way to find a k-coloration of (G) (if* $k > treewidth(G)$*).*

Illustration : we will use the previously trees to solve the problem with this graph :

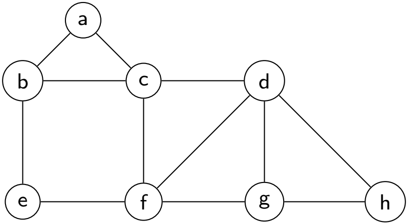


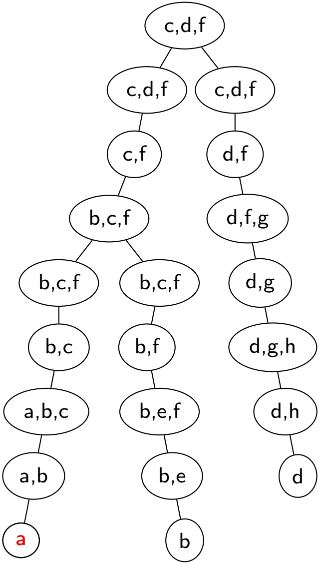
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



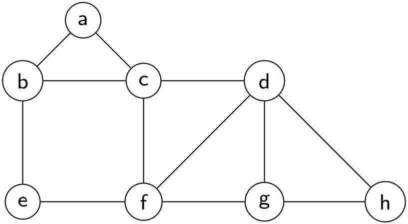


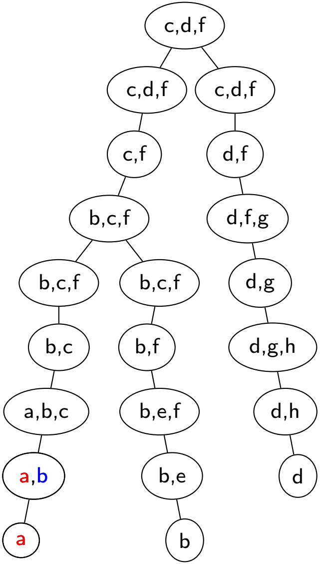
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



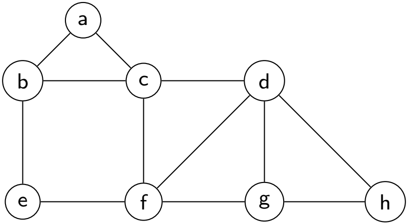


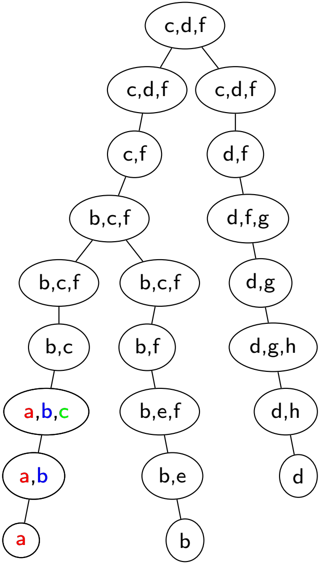
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





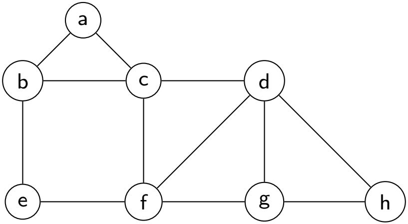
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

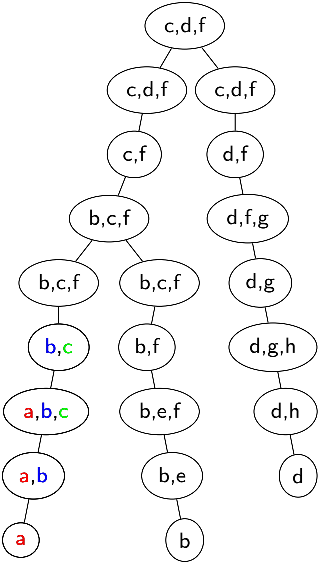




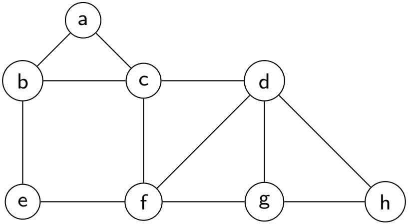
nice tree of the graph

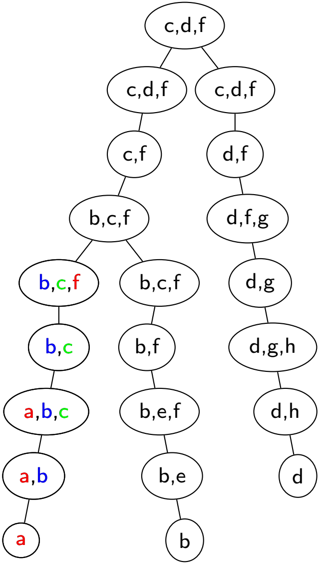
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



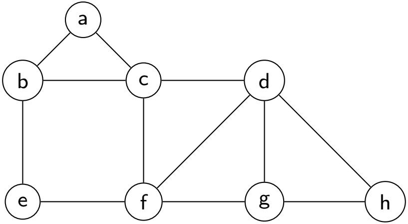


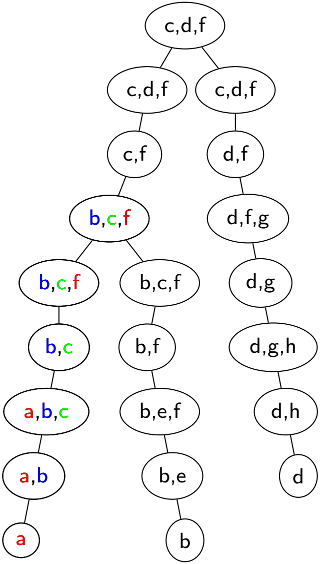
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



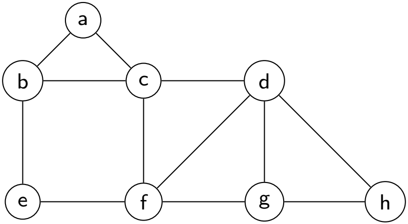


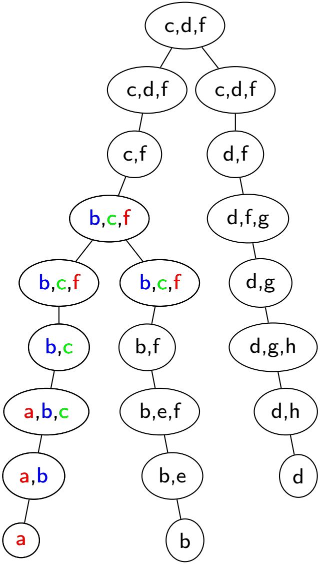
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



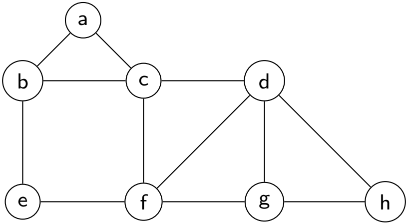


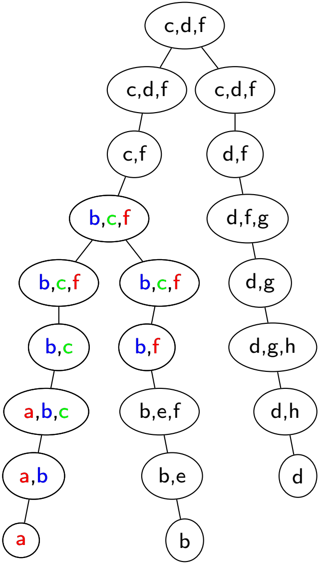
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.





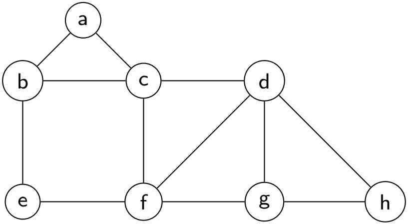
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

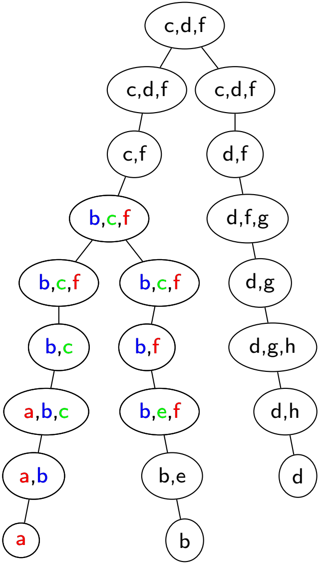




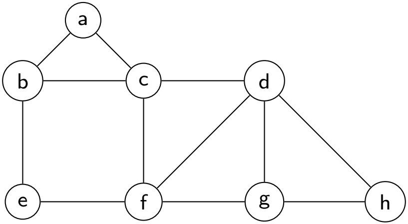
nice tree of the graph

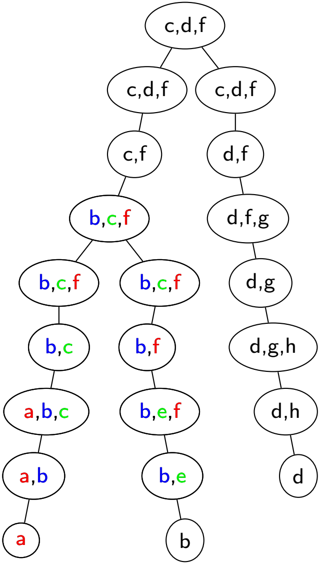
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





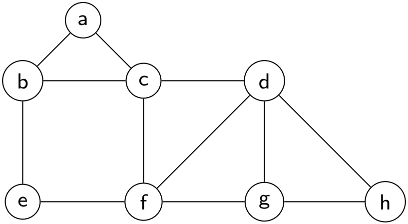
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

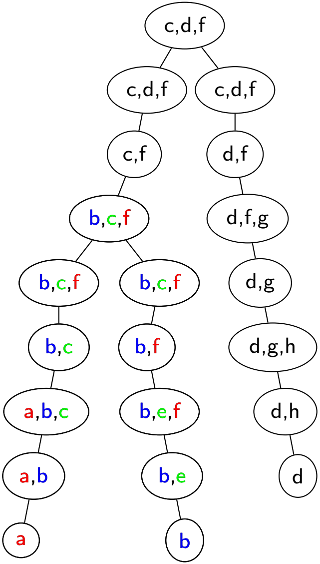




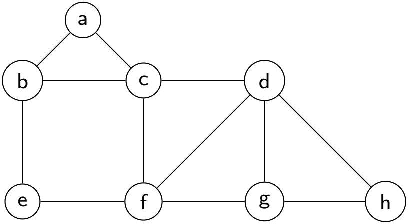
nice tree of the graph

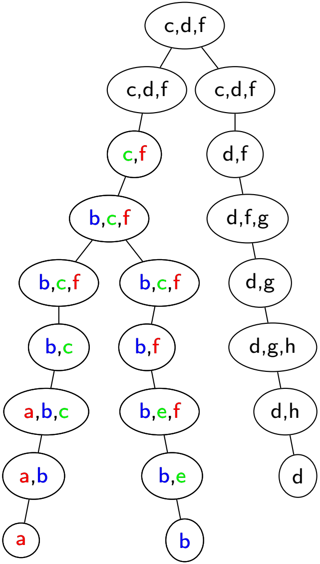
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





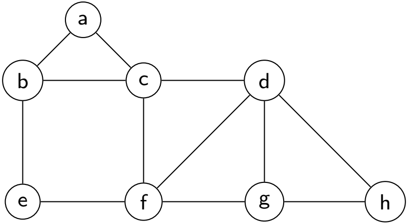
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

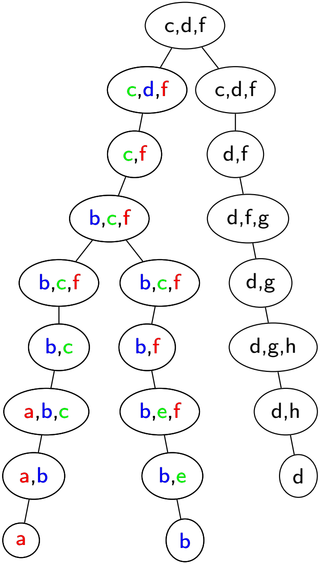




nice tree of the graph

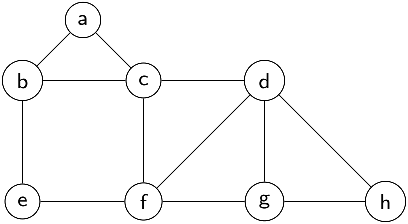
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

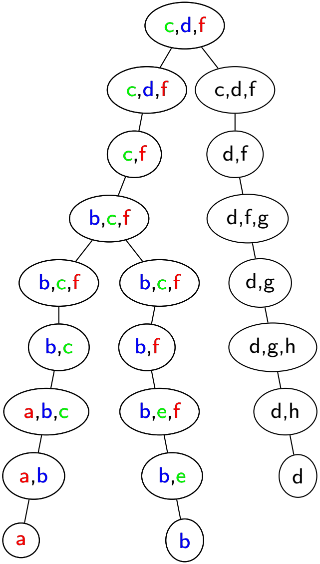




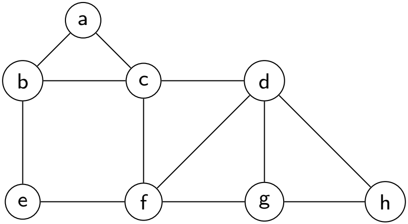
nice tree of the graph

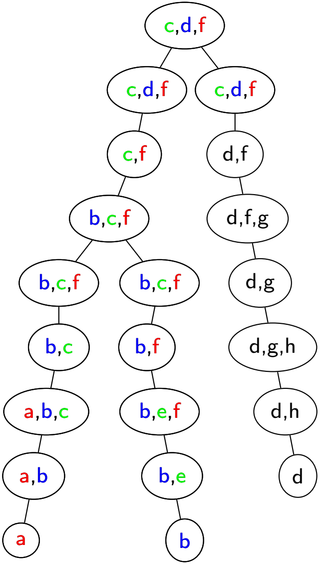
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



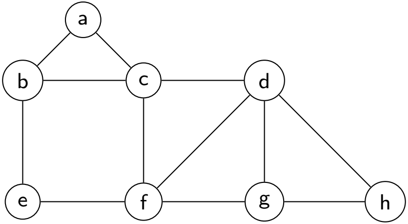


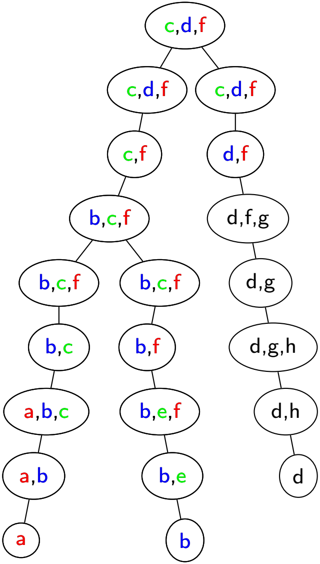
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





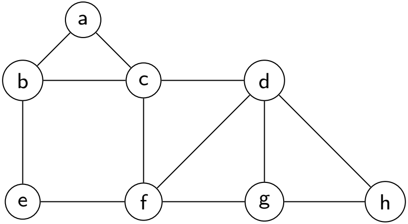
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

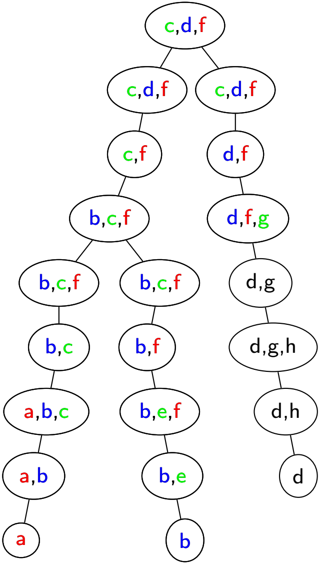




nice tree of the graph

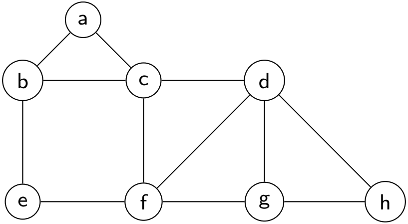
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

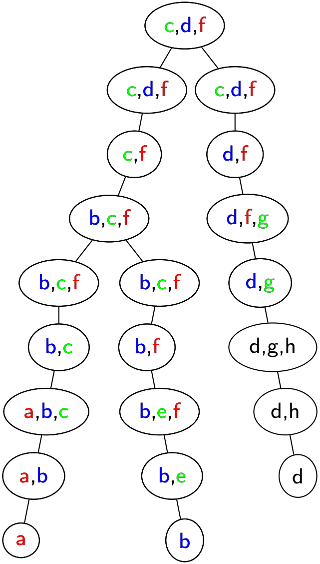




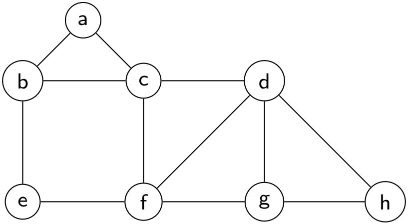
nice tree of the graph

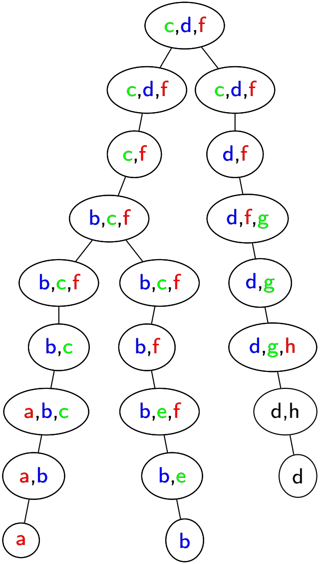
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





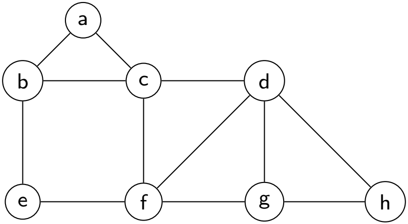
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

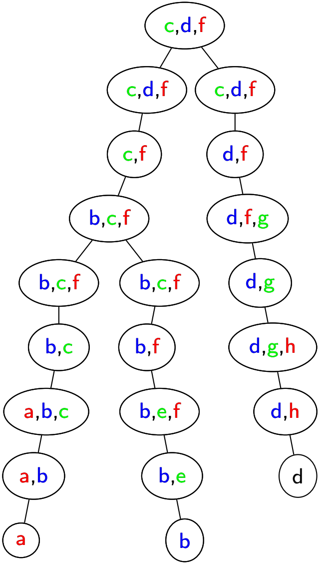




nice tree of the graph

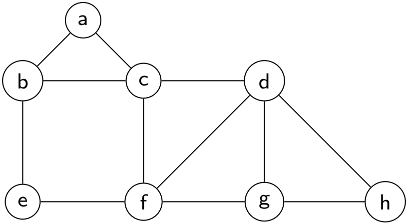
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.

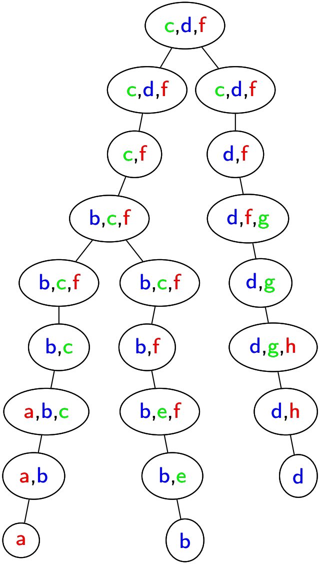




nice tree of the graph

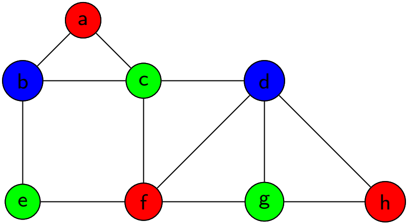
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.

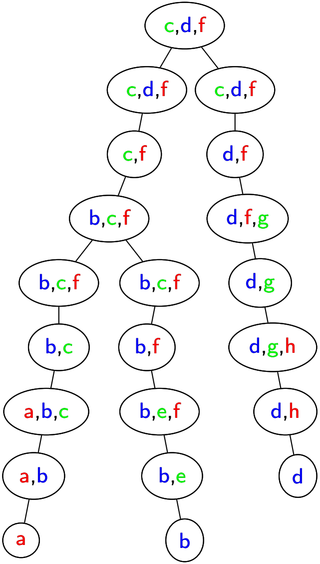




nice tree of the graph

- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





Contents

Bibliography

(1) Florent Madeleine : lesson for M2 Decim "Complexité des CSP et des requêtes"

(2) Dániel Marx : Fixed Parameter Algorithms

(3) wikipedia : articles of the graph section