Graphs' decompositions and resolutions of combinatorial problems

Stéphane Secouard
Supervised by : Florent Madeleine

Caen University - Computer science

25 october 2016

université de Caen Basse-Normandie

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.*

Objectives (of my presentation)

Now, I am going to present :

- *important concepts to understand the goals of my project ;*
- *an example where this concepts help to solve a problem of coloration.*

Contents

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 :

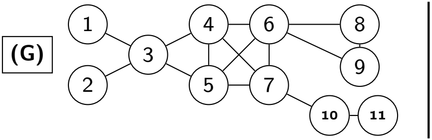
1. If u and v are neighbors in G, then there is a bag of T containing both of them (a bag is a node of the tree).
2. For every vertex v of G, the bags of T containing v form a connected subtree

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 :

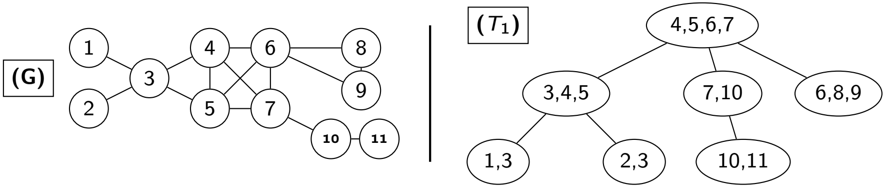


Definition (Tree decomposition of a graph)

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 :

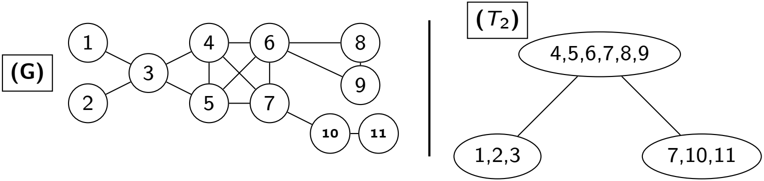


Definition (Tree decomposition of a graph)

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 :

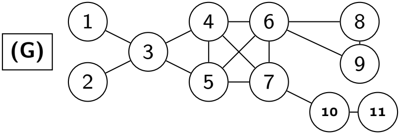


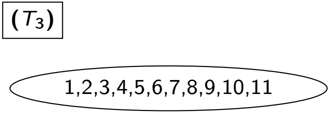
Definition (Tree decomposition of a graph)

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)

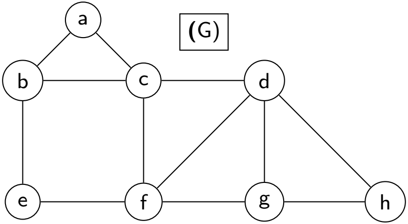
- The *width* of a decomposition is (largest bag size - 1).
- The *treewidth* of a graph is the lowest width of all decompositions of this graph.

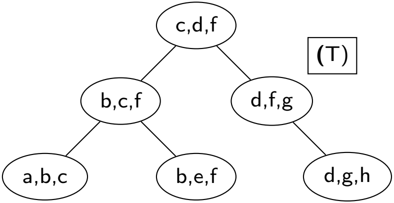
Treewidth

Definition (treewidth)

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

Example : Assume (T) is one of the representations of (G) which are giving the treewidth of (G) :

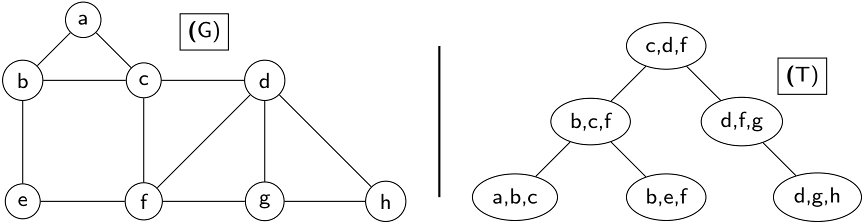




Definition (treewidth)

- 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 : Assume (T) is one of the representations of (G) which are giving the treewidth of (G) :

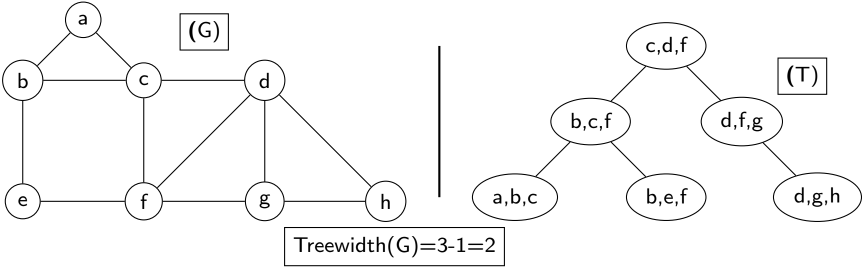


Treewidth(G)=3-1=2

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 : Assume (T) is one of the representations of (G) which are giving the treewidth of (G) :



Remark

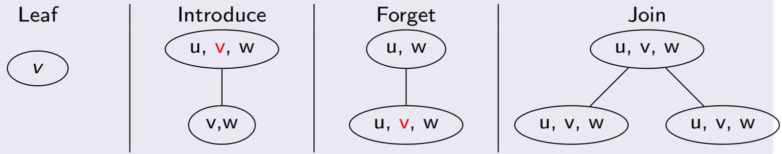
- *The treewidth of a tree is 1.*
- *If a graph has a treewidth of 1 we can state that this graph is a forest (i.e. a collection of trees).*

Nice tree

Definition (nice tree)

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

- **Leaf :** no children, $|B_x| = 1$ $\quad$ (B_x names a bag of the tree containing x)
- **Introduce :** 1 child y, $B_x = B_y \cup \{v\}$ for some vertex v
- **Forget :** 1 child y, $B_x = B_y \backslash \{v\}$ for some vertex v.
- **Join :** 2 children y_1, y_2 with $B_x = B_{y_1} = B_{y_2}$

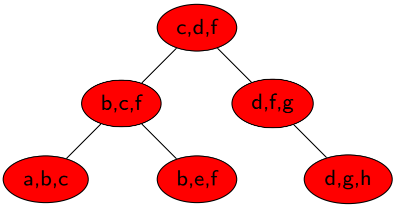


Remark

- *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).*

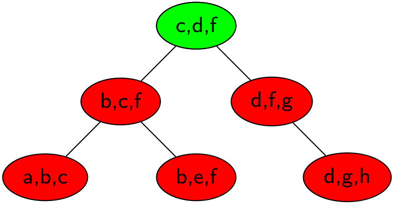
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



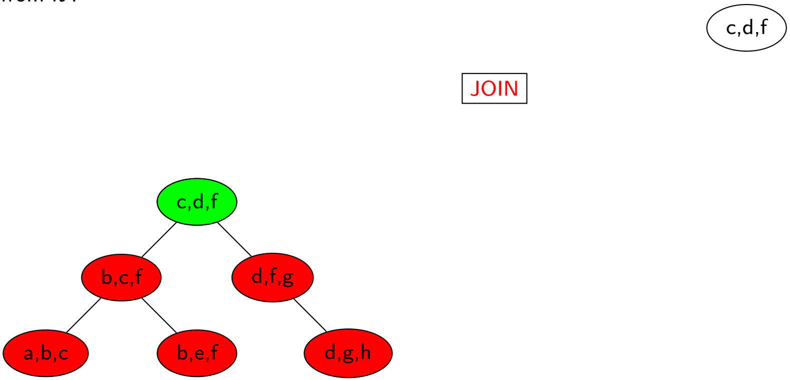
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



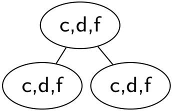
Example

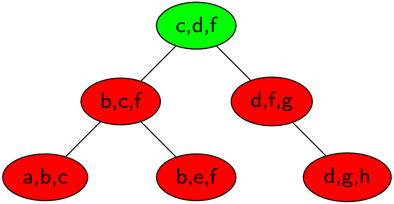
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?

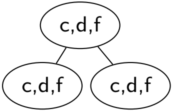
JOIN





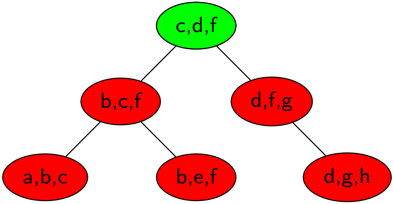
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



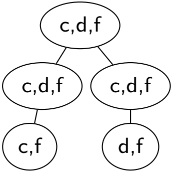
JOIN

FORGET



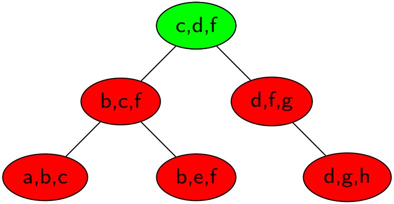
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



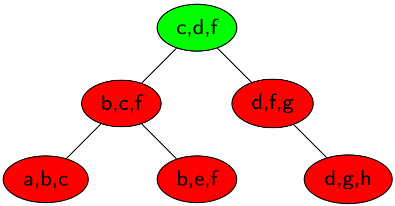
JOIN

FORGET



Example

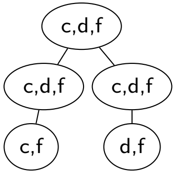
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



JOIN

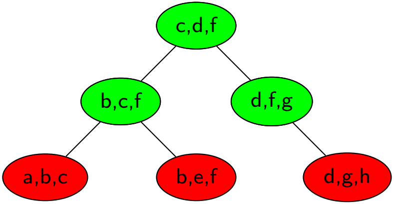
FORGET

INTRODUCE



Example

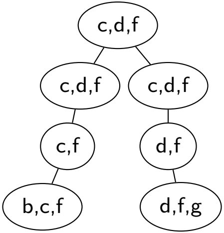
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



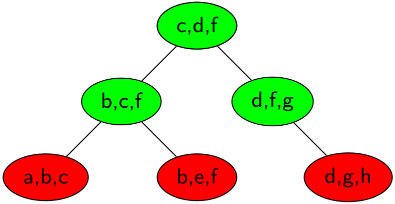
JOIN

FORGET

INTRODUCE



This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?

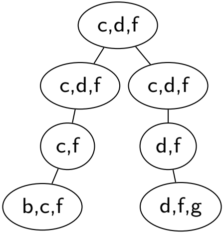


JOIN

FORGET

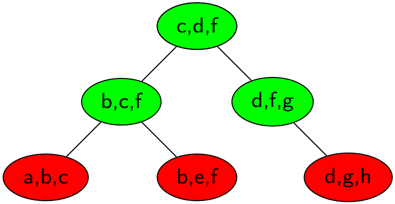
INTRODUCE

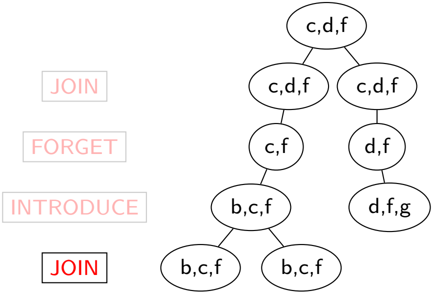
JOIN



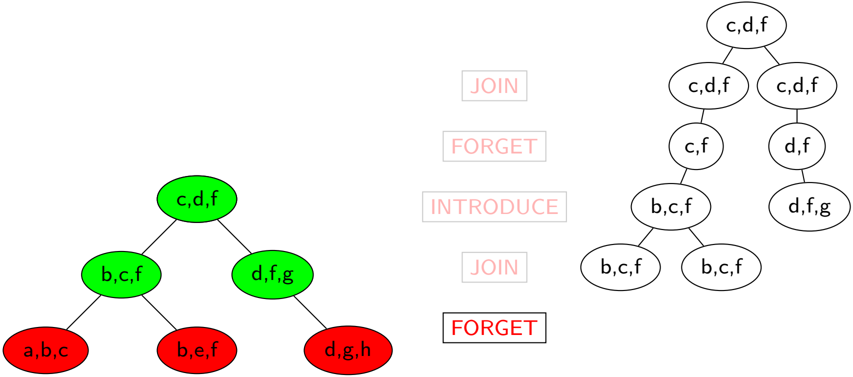
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



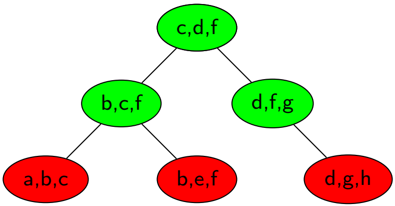


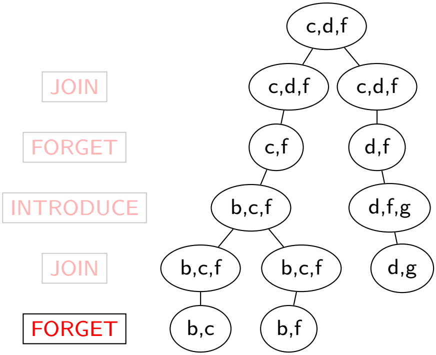
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



Example

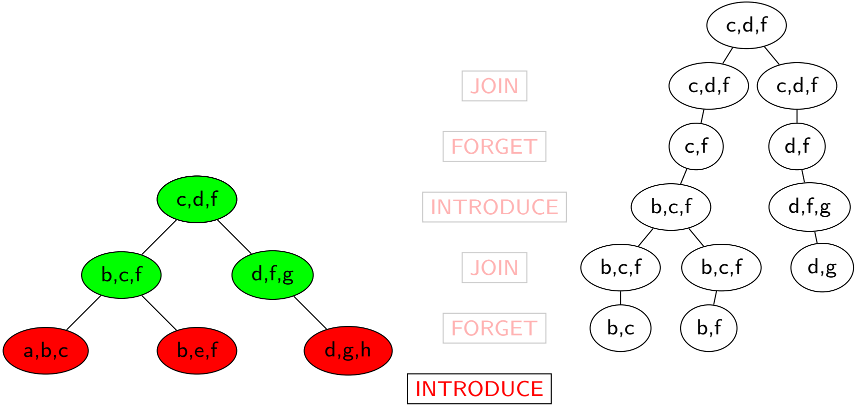
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?





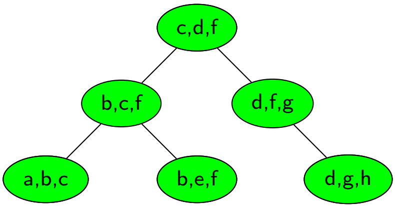
Example

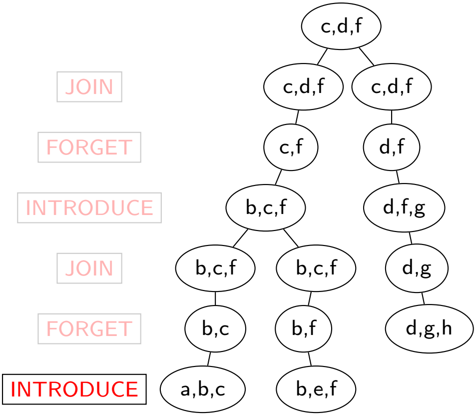
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



Example

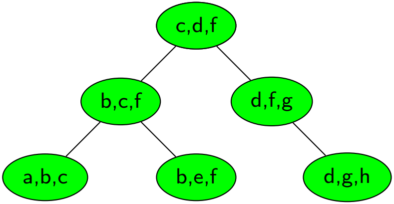
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?

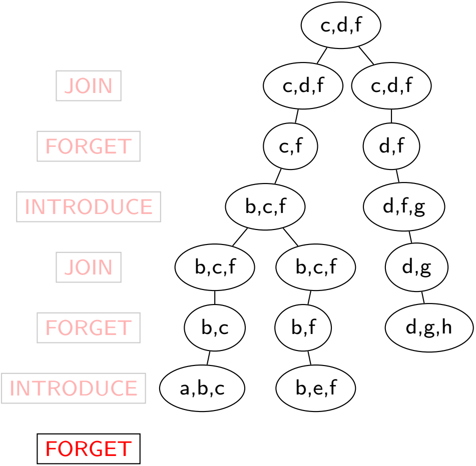




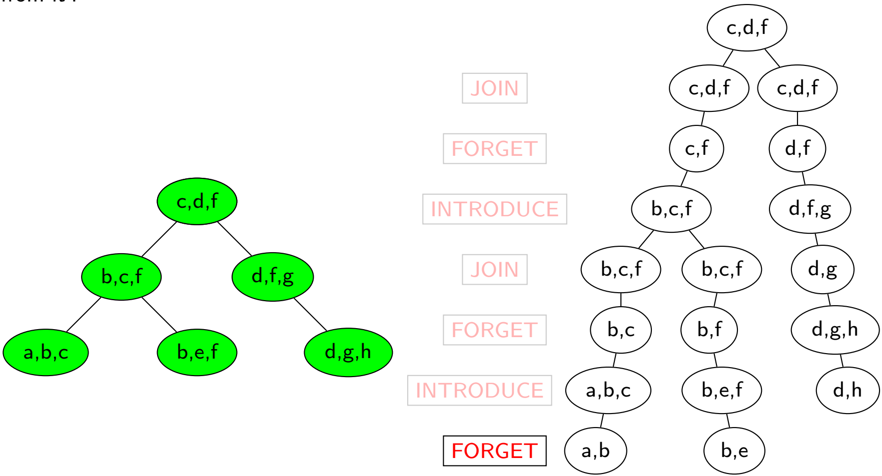
Example

This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



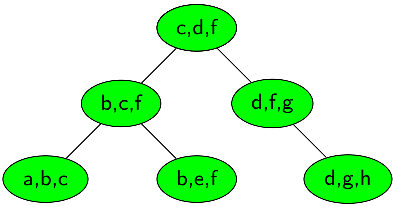


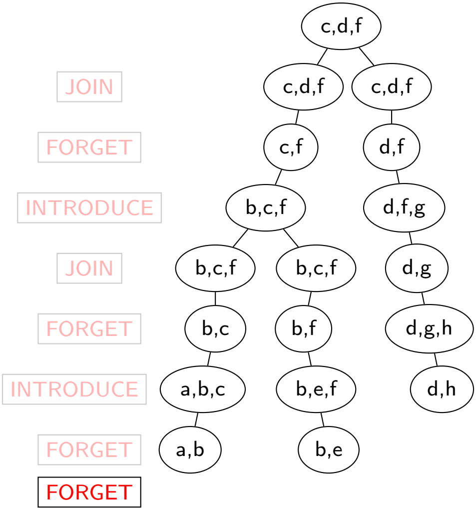
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



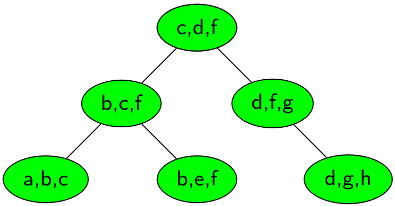
Example

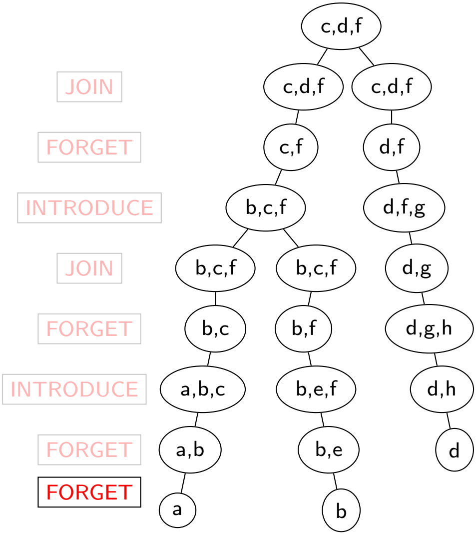
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?



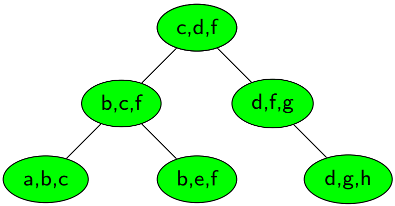


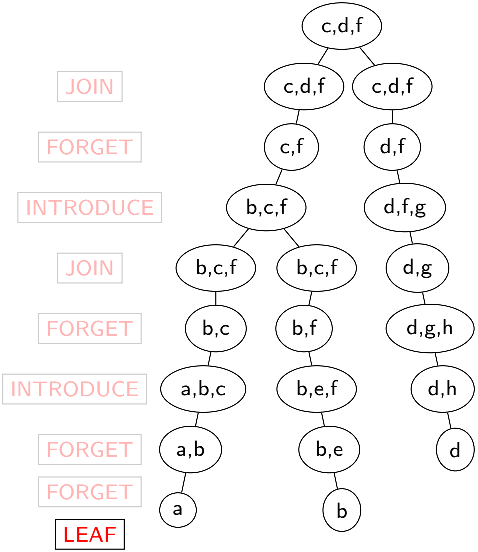
This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?





This is a tree decomposition of the graph seen previously. How can we get a nice tree from it ?





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)).*

the problem

Problem (k-color)

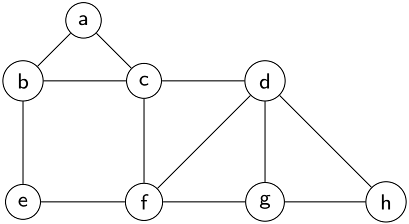
Let (G) be a graph and k an integer. We want to know if it is possible to draw each vertice of the graph so that two neighbors have never the same color and with only k colors.
This problem is a problem of decisions problem which is NP-complet.

Solution

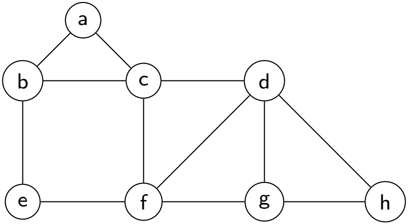
tree-width : *k-color is possible for a graph (G) if and only if $k >$ treewidth(G).*

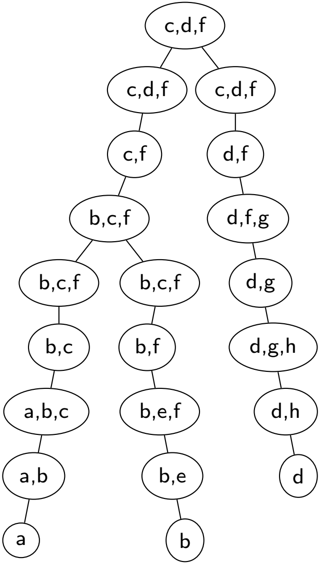
nice tree : *a nice tree of (G) gives a way to find a k-coloration of (G) (if $k >$ treewidth(G)).*

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

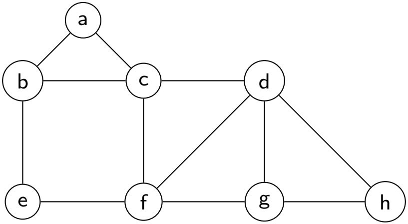


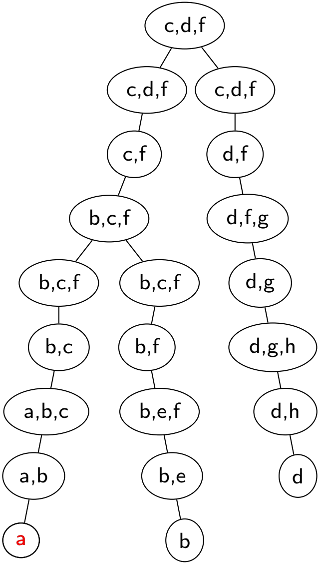
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



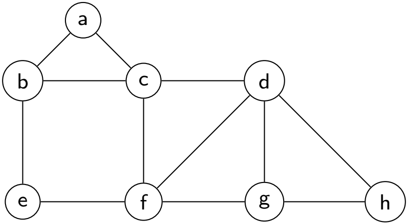


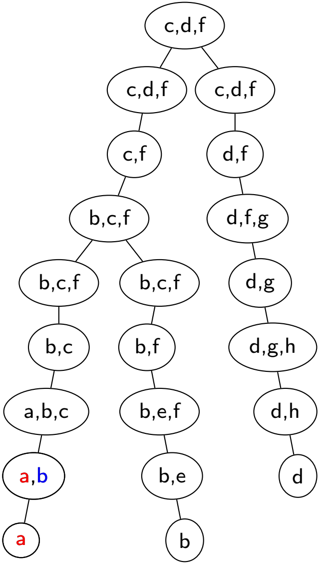
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



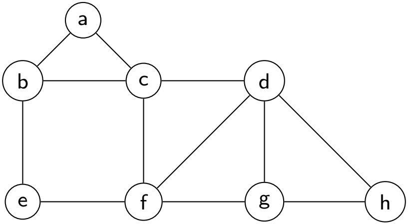


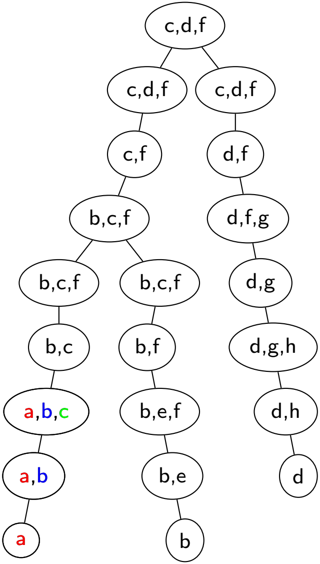
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





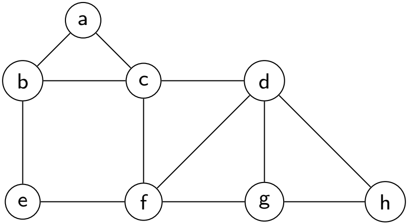
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.

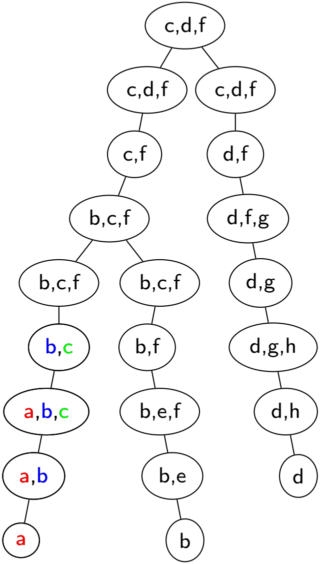




nice tree of the graph

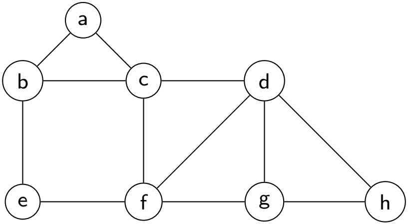
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

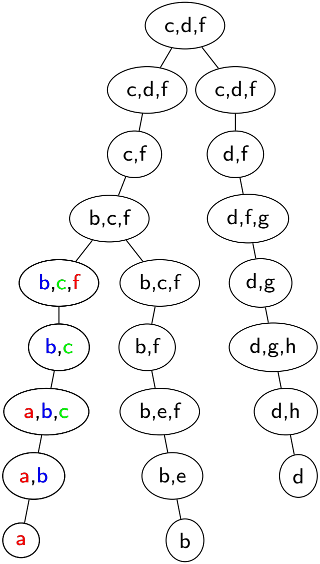




nice tree of the graph

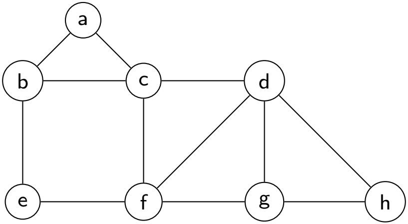
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

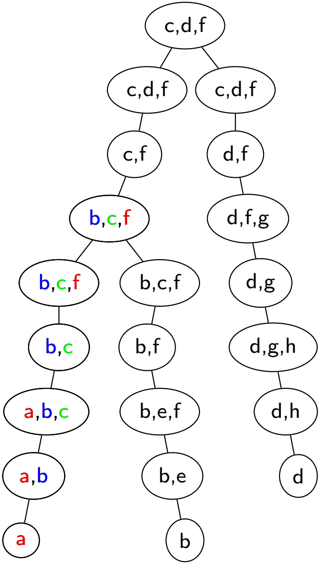




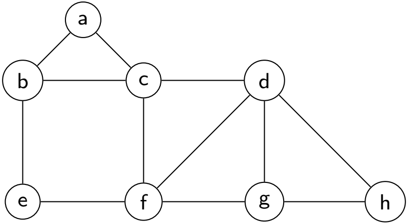
nice tree of the graph

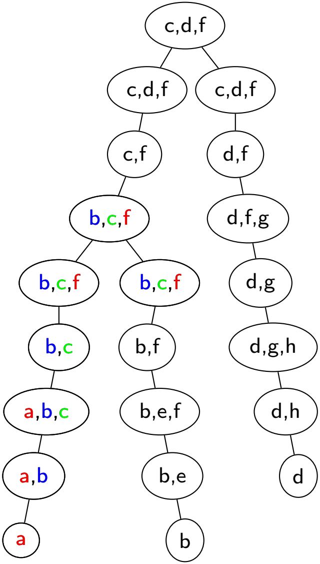
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



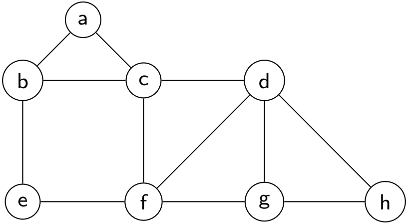


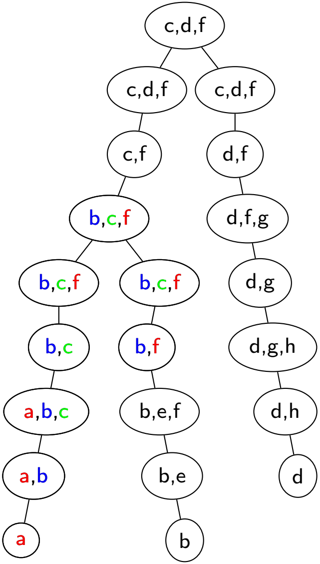
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.





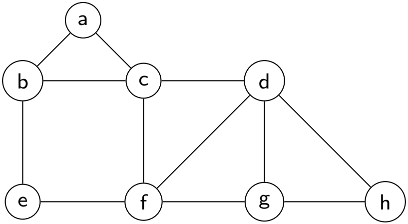
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.

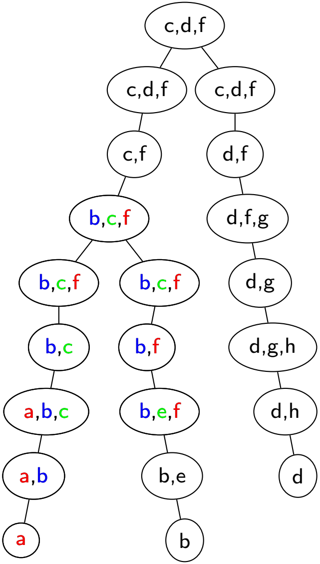




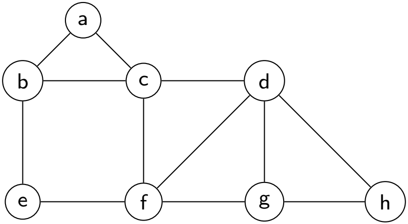
nice tree of the graph

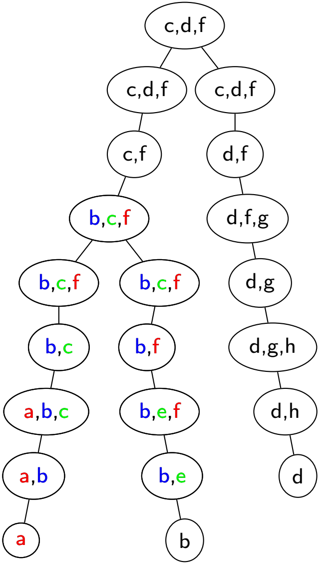
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





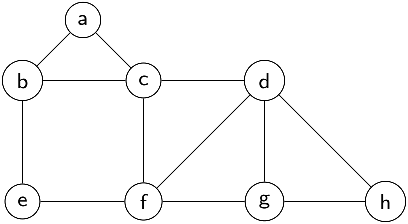
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

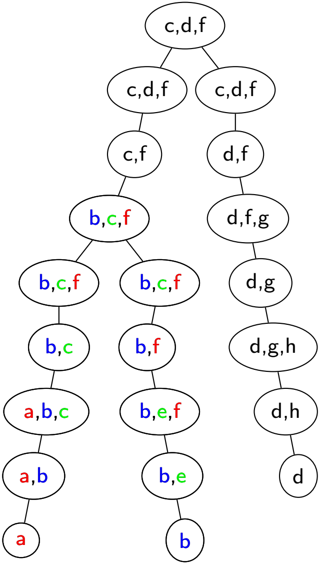




nice tree of the graph

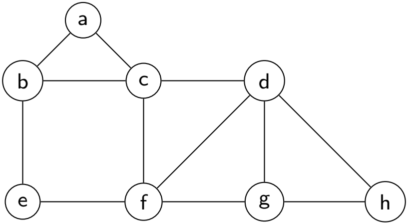
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

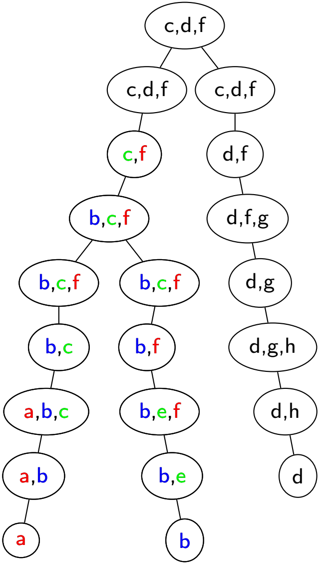




nice tree of the graph

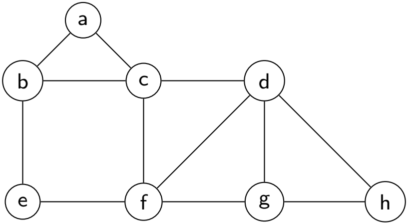
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

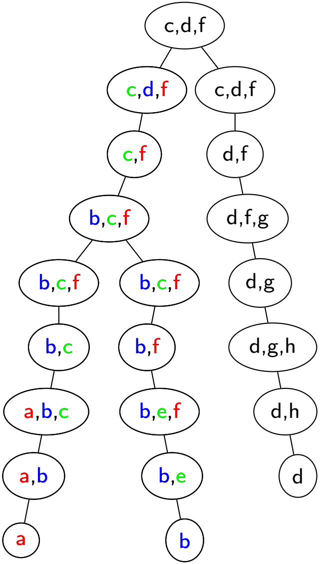




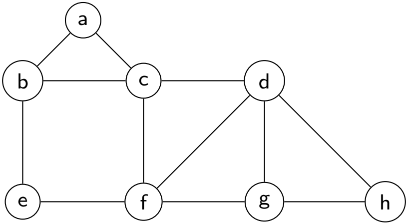
nice tree of the graph

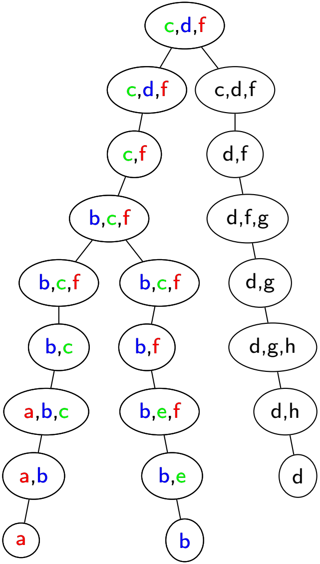
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



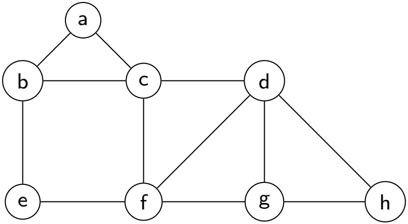


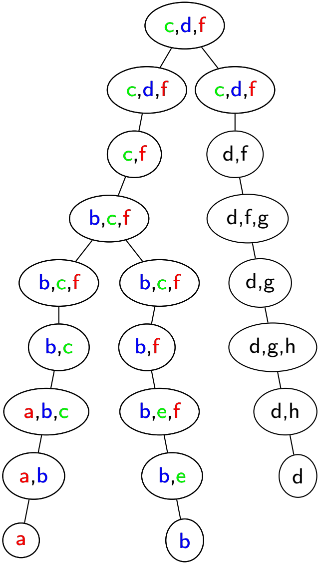
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



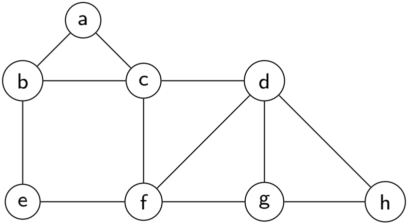


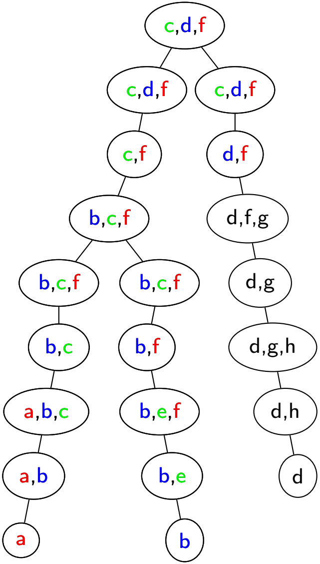
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.





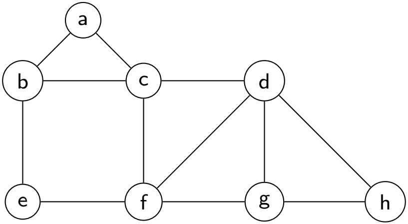
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

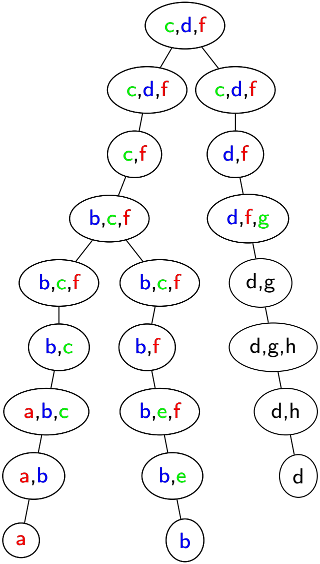




nice tree of the graph

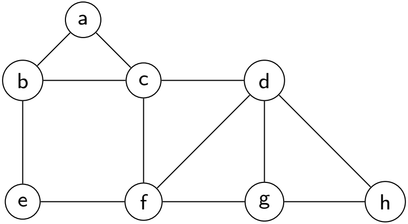
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.

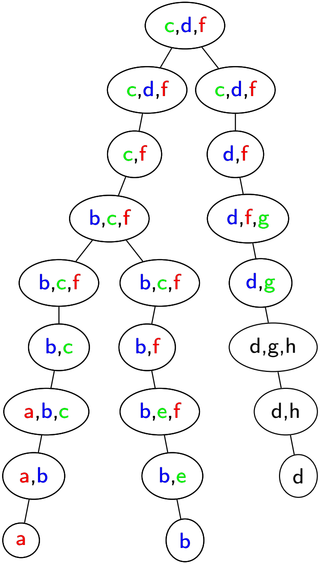




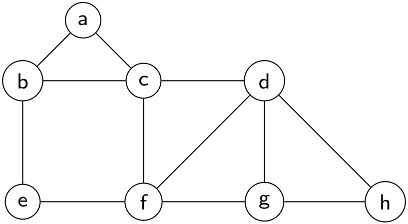
nice tree of the graph

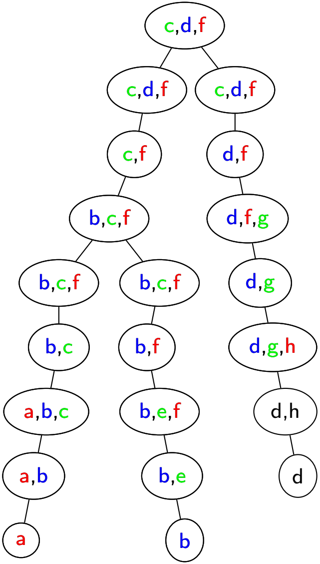
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a “Leaf”
- when we meet an “Introduce” we add a color
- when we meet a “Forget” we can state that the vertex which has disappeared won’t come back (property of the decomposition) and so we can reuse its color.



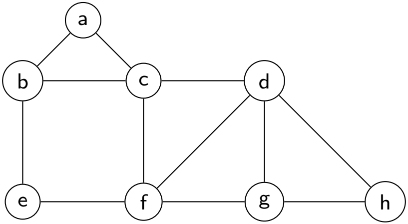


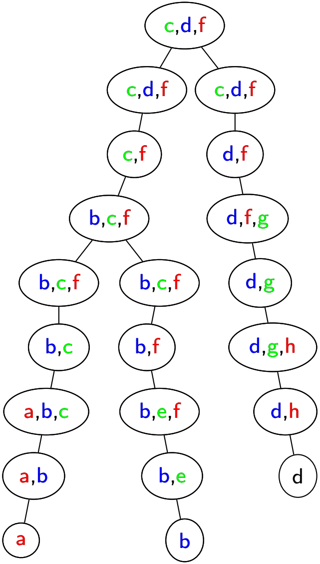
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



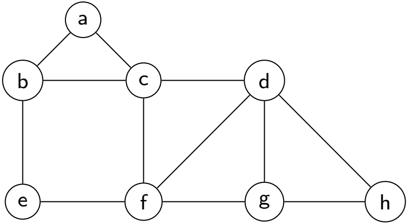


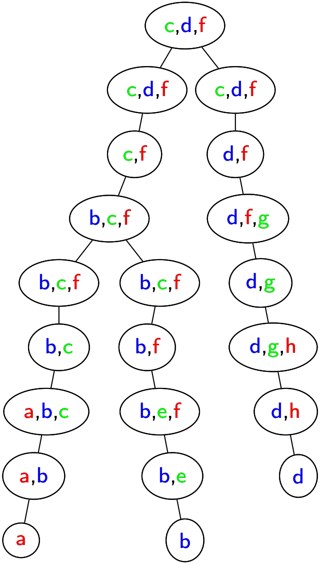
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.



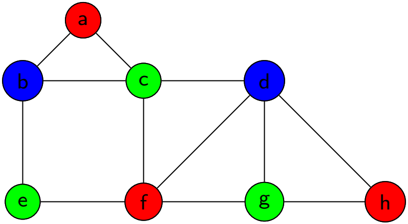


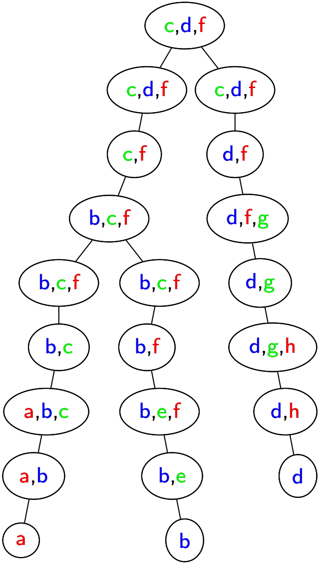
- treewith(G)=2 : we can solve the problem with 3 colors
- we can fix a color for a "Leaf"
- when we meet an "Introduce" we add a color
- when we meet a "Forget" we can state that the vertex which has disappeared won't come back (property of the decomposition) and so we can reuse its color.





- 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.





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