A Small Report on Graph and Tree Isomorphism

Marthe Bonamy

November 24, 2010

Abstract

The graph isomorphism problem consists in deciding whether two given graphs are isomorphic and thus, consists of determining whether there exists a bijective mapping from the vertices of one graph to the vertices of the second graph such that the edge adjacencies are respected. The graph isomorphism problem is a well-known open problem that was first listed as an important open problem by Karp, over three decades ago. It is something that is not inherent to graph theory in itself. When it comes to the manipulation of graphs, they are theoretical objects, their representation is not primordial, and we consider two graphs to be equal if they can be represented the same way. However, when we started doing computer-aided graph theory, this was something we could not be satisfied with, since the computer does everything with the representation, and a stricter equality is not acceptable. So we had to build algorithms as efficient as possible to test graph isomorphism. This report is a small survey on the basic algorithms and results on this isomorphism problem.

1 Tree isomorphism

As usual, we start with a smaller, yet useful class of graphs. Namely trees. We start with a very specific case of tree isomorphism and broaden it to the general cases in three steps.

1.1 Rooted ordered tree isomorphism

Rooted trees are ordered when there is a strict order on the sons of every vertex. $\forall v \in V$, $first[v]$ refers to the first son of v in that order, if there is one, and $next[v]$ refers to the first son of its father that is after v in the order, if there is one. $root$ is, of course, the root.

See the formal definition in [1].

Definition 1. *Two rooted ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are **isomorphic**, denoted by $T_1 \cong T_2$, if there is a bijection $\sigma : V_1 \to V_2$ such that $\sigma(root_1) = root_2$ and the following conditions:*

- $\sigma(first_1[v]) = first_2[\sigma(v)]$ *for all nonleaves* $v \in V_1$
- $\sigma(next_1[v]) = next_2[\sigma(v)]$ *for all nonlast child* $v \in V_1$

are satisfied. In such a case, σ is an ordered tree isomorphism of T_1 to T_2.

For any two rooted ordered trees, there exists a linear algorithm that exhibits an isomorphism if there is one. We do a depth-first search and number the vertices as we come by them ($O(n)$). Then we check that the induced isomorphism ($\forall u \in V_1, \sigma(u) = v$ iff u and v have been labelled with the same number) is indeed an isomorphism ($O(n)$). So, overall, this is a linear algorithm in both time and space.

Theorem 1. *Rooted ordered tree isomorphism is decidable in linear-time in the number of vertices.*

1.2 Rooted tree isomorphism

Now, we consider simple rooted trees (which could be seen as oriented trees). To manipulate those, we use an array of lists, **sons**, which takes a vertex as an input and returns the list of its sons. In that case, if T_1 and T_2, of roots u_1 and u_2, are isomorphic, then there exists an isomorphic permutation σ such that $\sigma(u_1) = u_2$. We define this more rigorously in [1].

Definition 2. *Two rooted trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are **isomorphic**, denoted by $T_1 \cong T_2$, if there is a bijection $\sigma : V_1 \to V_2$ such that $\sigma(root_1) = root_2$ and the following condition:*

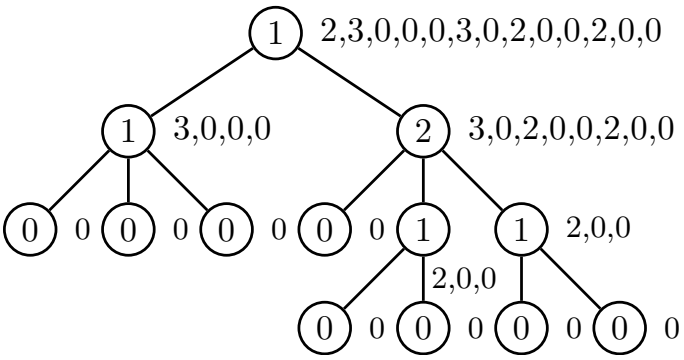
- $\sigma(v) = w, v \in sons_1[u] \Rightarrow w \in sons_2[\sigma(u)]$ *for all nonroot* $v \in V_1$

is satisfied. In such a case, σ is a rooted tree isomorphism, or just a tree isomorphism, of T_1 to T_2.

The following algorithm (which we will call **IsoRooted**) is of the divide-and-conquer kind, and was presented in [1]. It is apparented to a merge sort. We associate a unique label to each subtree (that is stable up to isomorphism) and deduce from it the label of the father. And we go on.

Algorithm 1. ***Label***$(u$*,sons)*=
number←*Length(sons[*u*])*
offspring ← []
next ← $sons[u]$
While $next \neq$ [] *do*
vertex← *Head(next)*
next← *Tail(next)*
offspring←***Label***(*vertex,sons*)::*offspring*
done
Sort(offspring)
Return (number,clean(offspring))

Where clean takes a list as an input and returns a string. $clean[a;b] = "a,b"$.
For example on the following tree.



Two trees are isomorphic if and only if the labels of their roots are equal. Indeed, the label of the root characterizes the whole graph and is unique.

Consequently,

Algorithm 2. ***IsoRooted***$(root_1, sons_1, root_2, sons_2)$=
(***Label***$(root_1, sons_1) =$ ***Label***$(root_2, sons_2)$)

If we wanted to exhibit the isomorphism, all we would have to do is keep track of the place of each vertex in the labels of its successive fathers. And then, match the two labels to map the vertices of the two trees.

The sorting costs the number of vertices in the induced subtree, and we call Sort for every vertex. Hence, we get an algorithm in $O(n^2)$ in time, $O(n)$ in space. But we can do better than that.

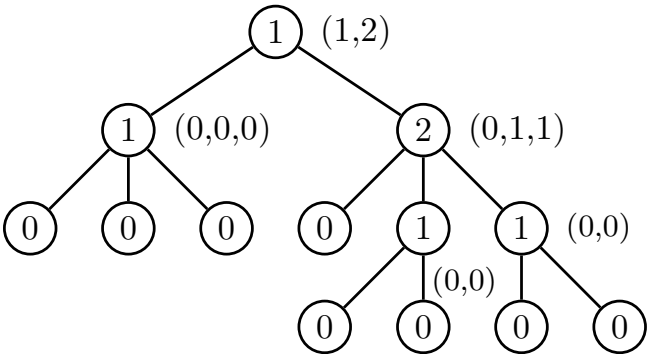
Theorem 2. *Rooted tree isomorphism is decidable in time linear in the number of vertices.*

Proof. As presented in [2].

Algorithm 3. *T_1, T_2 the two rooted trees we want to test. Let n be their size (if it isn't the same, they are not isomorphic). We call the **level** of a vertex the height of the tree minus its distance to the root.*
We assign to all leaves of T_1 and T_2 the integer 0. L_1 (resp. L_2) the list of the leaves of T_1 (resp. T_2).
For $i = 1$ to n do

1. *All vertices of T_1 and T_2 at level $i-1$ have been assigned integers. L_1 is a list of the vertices of T_1 at level $i-1$ sorted by non-decreasing value of the assigned integers. L_2 is the corresponding list for T_2.*
2. *Assign to the nonleaves of T_1 at level i a tuple of integers by scanning the list L_1 from left to right and performing the following actions:*
   - *For each vertex on list L_1 take the integer assigned to v to be the next component of the tuple associated with the father of v.*
   - *On completion of this step, each nonleaf w of T_1 at level i will have a tuple $(i_1, i_2, .., i_k)$ associated with it, where $i_1, .., i_k$ are the integers, in non-decreasing order, associated with the sons of w.*
   - *Let S_1 be the sequence of tuples created for the vertices of T_1 on level i.*
3. *Repeat step 2 for T_2 and let S_2 be the sequence of tuples created for the vertices of T_1 on level i.*
4. *Sort S_1, S_2. Let S_1', S_2', respectively, be the sorted sequence of tuples.*
5. *If S_1' and S_2' are not identical, then halt: the trees are not isomorphic. Otherwise, assign the integer 1 to those vertices of T_1 on level i represented by the first distinct tuple on S_1', assign the integer 2 to the vertices represented by the second distinct tuple, and so on. As these integers are assigned to the vertices of T_1 on level i, replace L_1 by the list of the vertices so assigned. Append the leaves of T_1 on level i to the front of L_2. Do the symmetric for L_2.*

done
T_1 and T_2 are isomorphic iff the roots of T_1 and T_2 have been assigned the same integer, .



We will refrain from studying the correctness and complexity, and will assume that it is correct, and it is linear. See [2] for further information.

□

1.3 Unrooted tree isomorphism

We can now extend the problem to classical trees (ie connected, undirected graphs without loops).

The previous subsection provides us with a linear algorithm to determine whether two rooted trees are isomorphic. But what about the isomorphism problem in the general case, when we do not have such a useful start as a root? We can trivially adapt **IsoRooted** to make him work on unrooted trees: we only have to pick a vertex in the first tree and declare it to be its root, and try all the vertices of the second tree one after the other: the two trees will be isomorphic if and only if there is one vertex in T_2 which, when declared as a root, prompts **IsoRooted** to answer positively.

Definition 3. *Two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are **isomorphic**, denoted by $T_1 \cong T_2$, iff $\exists u \in V_1, \exists v \in V_2$ such that the rooted trees obtained when T_1 is rooted in u and T_2 in v admit a rooted tree isomorphism σ of the first to the second.*

In such a case, σ is also a tree isomorphism, of T_1 to T_2.

This makes a $O(n^2)$ time complexity (we still have the $O(n)$ bound for space). Can we do something better? We could pick for example a vertex of maximal degree in the first tree, and then we only have to consider the vertices of maximal degree (if it isn't the same, the game is over) in the second one. But with for example a chain, this does not help. We have to find a better pre-selection (and one that does not cost more than $O(n)$, preferably), we want to find a set as small as possible that is stable up to isomorphism. It is quite natural, from the counter-example of the chain, to add a notion of diameter. For example, if there is only one vertex of maximal degree, pick it, and if there is more than one, consider:

The bound $M = \max_{\{u,v \in V | deg(u)=deg(v)=maximum\ degree\}}\{d(u,v)\}$.
The set $S = \{u, v \in V | u, v$ are of maximum degree and $d(u,v) = M\}$.

This can be computed in $O(n^2)$ time and $O(n)$ space, so that's not acceptable, though S is of size at most $log(n)$. This would induce a complexity of $O(nlog(n))$, which is better...except for the quadratic-time preselection which raises the time complexity to quadratic.

But we can do much better than that! We define

$S = \{u \in V |$ for T_i the connected components of $T \backslash \{u\}, \forall i, |T_i| \leq \frac{|V|}{2}\}$.

S is of size 1. And we can determine the only inhabitant of S in linear time (we could apparent it to a kind of "gravity center" of the tree). We arbitrarily root the tree in some $v \in V$. We associate to each vertex the weight of its induced subtree. This costs $O(|V|)$. Then we apply **Shift**

Shift(u) := If no neighbour of u has a weight $> \frac{n}{2}$,

- Return u.
- Otherwise,
  - We shift the weight to this neighbour (let's call it w):
  - $\omega(u) \leftarrow \omega(u) - \omega(w)$
  - $\omega(w) \leftarrow \omega(u) + \omega(w)$
  - **Shift**(w)

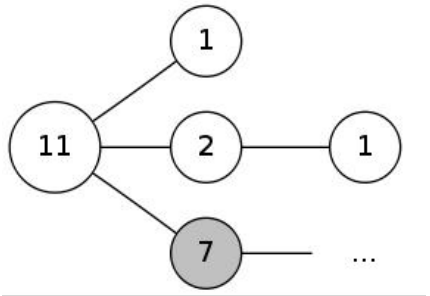


Figure 1: Before

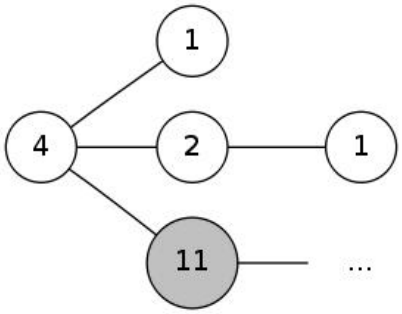


Figure 2: After

And at first we call **Shift** on the root. On the whole, **Shift** costs at most $O(|V|) + O(|E|)$, so $O(|V|)$.

Consequently, given two trees with no added information, we arbitrarily root them, we weigh them, and launch **Shift** on their root. We root them according to the result of it. Then we can use **IsoRoot** on them, since the two trees are isomorphic iff their rooted selves are.

Theorem 3. *Tree isomorphism is decidable in linear time in the number of vertices.*

1.4 Generalisation to Chordal graph isomorphism

Definition 4. *Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic**, denoted by $G_1 \cong G_2$, if there is a bijection $\sigma : V_1 \to V_2$ such that:*

- $\forall u \in V_1, \{\sigma(v) \in V_1 | (u, v) \in E_1\} = \{w \in V_2 | (\sigma(u), w) \in E_2\}$

In such a case, σ is a graph isomorphism, of G_1 to G_2.

Since chordal graphs are so structurally close to trees, it is very tempting to try and apply the same techniques to them. Finding a "clique root" in the same spirit is linear too (just have to use the clique tree, which is build in linear time). This is quite nice because this means that we afterwards only have chordal graphs of halved size. Two main problems occur: first, we do not know which one should match to which other one. Then, we cannot really merge solutions together: they have first to be coherent on the main clique, and nothing guarantees us that it will be the case. Another difficulty lies in the non-uniqueness of the clique tree representation.

However, the non-uniqueness of the clique tree decomposition can be tackled by doing a small modification on the clique tree. Instead of considering the set of maximal cliques, we consider the set of maximal cliques and of minimal separators. Between any maximal clique and any minimal separator, there is an edge if and only if the separator is included in the clique. This representation is unique (by construction) and characterizes the chordal graph (we have the set of maximal cliques). And it can be built in linear time too. This seems a rather powerful tool to use, but there was nothing to be found on it on the internet, and it does not look trivial.

Another thing to notice is that the notion of "gravity center" introduced for trees can be extended to chordal graphs, and in that case we are not looking for a gravity center being a vertex, but for a gravity center being a clique. It is defined as the minimal separator such that no connected component induced by it is greater than half the size of the whole graph, and is also unique.

But this is not enough to tackle the isomorphism problem.

2 Cograph isomorphism

However, we do have a unique tree representation for graphs: the modular decomposition!

When the labels are "Clique" or "Independent", their equality can be tested in constant time. However, when the labels are prime graphs, we have to test their isomorphism. Building the modular decomposition takes $O(m+n)$. Testing whether the two trees are isomorphic (assuming we have an oracle for prime graphs isomorphism) takes $O(m+n)$. In the case where the answer is positive, restricting the isomorphism to the leaves of the modular decomposition tree (which are the vertices of the graph) takes $O(n)$. The restriction is valid, because the set of leaves is stable up to isomorphism. Consequently, testing the isomorphism of two general graphs is about as hard as testing the isomorphism of two prime graphs.

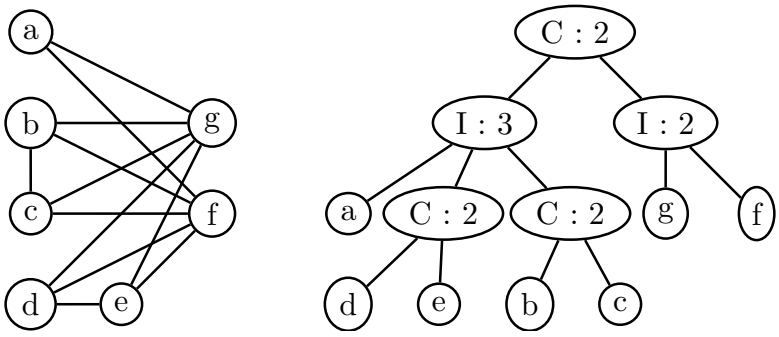


Figure 3: Unique representation of a cograph

Another thing we can deduce from it is mainly a linear algorithm ($O(m+n)$) to check whether two cographs are isomorphic. Indeed, a cograph is a graph that induces a P_4. But a prime graph always induces a P_4. Since every label of the modular decomposition tree is an induced subgraph of the initial graph, a cograph is a graph without any prime graph in its modular decomposition. So we never have to use the oracle. Meaning that:

Theorem 4. *Cograph isomorphism is decidable in linear time in the number of edges and vertices.*

3 Isomorphism-complete

3.1 Chordal graphs

A little above, we tried to extend the tree techniques to chordal graphs. Actually, if it had worked properly, it would have been material for an excellent paper. Indeed, the isomorphism problem is not known to be NP-complete, but neither do we know a polynomial algorithm. So the notion of "isomorphism-complete" was introduced, precisely to study which graph subclasses were known to be polynomial for the problem and which were as hard as for the general case. In the previous sections, we specified that rooted and non-rooted trees and cographs could be treated in polynomial time. We also exhibited a proof that prime graphs were isomorphism-complete. We are now going to prove that chordal graphs are isomorphism-complete too.

Theorem 5 (3)**.** *Graph isomorphism can be polynomially reduced to chordal graph isomorphism.*

Proof. Let $G = (V, E)$ be a connected, non-oriented graph, such that $|V| \geq 4$. We define $V' = V \cup E$. We build the edges as follows: $\forall u, v \in V, (u, v) \in E'$ and $\forall u, v \in V, (u, v) \in E \Rightarrow (v, (u, v)) \in E'$. $C(G) := (V', E')$.

1. G' is clearly polynomial in the size of G. Let us prove that G' is chordal.
2. In G', by construction there can be no edge between elements of E. Consequently, in any induced cycle of G', at least one vertex in two is an element of V. But in G', all the elements of V are adjacent. We deduce that there can't be any chordless induced cycle of size over 3. Hence G' is chordal.
3. We still have to prove that $\forall G_1 G_2, G_1 \cong G_2 \Leftarrow C(G_1) \cong C(G_2)$. In $C(G)$, every vertex has size either 2 (edge of G) or $|V| - 1$ (vertex of G). Since G contains at least four vertices, $2 < |V| - 1$. Consequently, in $C(G)$ we can tell apart the vertex-vertices from the edge-vertices. This means we can rebuild V, and then, just by looking at the two neighbours of the edge-vertices, rebuild E. Consequently two graphs G_1 and G_2 whose images by C are isomorphic must be isomorphic.

This means chordal graph isomorphism is polynomially as hard as graph isomorphism. □

3.2 Extension

Actually, we can extend this proof to a much stronger result.

Theorem 6. *Graph isomorphism can be polynomially reduced to 2-split graph isomorphism.*

Where:

Definition 5 (Tyshkevich Chernyak 1979)**.** *$G = (V, E)$ is a **split graph** if V can be partitioned in I and S such that $G|_I$ is an independent and $G|_C$ is a clique.*

Definition 6. *(invented for the occasion) $G = (V, E)$ is a **2-split graph** if G is a split graph where every vertex of the independent is linked to exactly two vertices of the clique.*

Proof. We re-use the same fonction C that was introduced in the previous proof. We claim that for any graph G of size at least 4, $C(G)$ is not only a chordal graph, but also a 2-split graph. Indeed, we can pick V for the clique and E for the independent, and in $C(G)$, every edge-vertex is linked to exactly two vertex-vertices. □

This is quite disconcerting because the 2-split graphs subclass seems a very easy-to-grasp, easy-to-manipulate subclass. And yet!

3.3 Quick survey of funny results

There are of course many other results on various subclasses. A rather useful, though trivial lemma is:

Lemma 1. *For A and B two graph subclasses, if $A \subseteq B$ and A isomorphism is isomorphism-complete, then so is B isomorphism.*

This means that thanks to the result on 2-split graphs, we suddenly get without extra effort the following result.

Lemma 2. *The problem of isomorphism on split graphs, interval graphs, chordal graphs, perfect graphs is polynomially as hard as graph isomorphism.*

But there is still more to it. We will restrict ourselves to two more intuitive subclasses, but the number of various classes whose completeness is known is quite impressive.

Lemma 3. *Bipartite graph isomorphism is polynomially as hard as graph isomorphism.*

Proof. We use the very same method as for chordal graphs, except that instead of transforming the set of vertices into a clique, we transform it into an independent. Let $G = (V, E)$ be a graph. We define $V' = V \cup E$ and E' such that $\forall u, v \in V, \forall (u, v) \in E, (u, (u, v)) \in E'$. $S(G) := (V', E')$.

1. $S(G)$ is build from G in polynomial time
2. $S(G)$ is trivially bipartite
3. $\forall G_1, G_2, S(G_1) \cong S(G_2) \Rightarrow G_1 \cong G_2$. Indeed, when we know one vertex of $S(G)$ to be an edge-vertex (resp. vertex-vertex), we know that exactly the vertices at even distance are of the same type. And knowing that, we can rebuild the edges.
   - Either $S(G)$ is a cycle. Then G is a cycle of half size (o)
   - Or $S(G)$ is not a cycle. Then there is a vertex that is not of degree 2. It must be a vertex-vertex.

   . Consequently, we can always rebuild G up to isomorphism, knowing $S(G)$.

□

Lemma 4. *Compact graph isomorphism is polynomially as hard as graph isomorphism.*

Definition 7. *$G = (V, E)$ is a **compact graph** if $\forall x, y \in V, d(x, y) \leq 2$*

Proof. Let $G = (V, E)$ be a graph. We add a vertex that is linked to all the existing vertices. The result is a compact graph. To go back to the former graph, we only have to detect a vertex that is linked to all the others (this is easy) and remove it. It doesn't matter if there was more than one, the result will still be isomorphic to the initial graph. Consequently, two graphs are isomorphic if and only if their images by this transformation are isomorphic. Hence the lemma holds. □

3.4 Neither NP-complete nor polynomial?

If $P \neq NP$, it is proved that there will be some intermediary problems, in NP, not NP-complete, but neither in P. It is strongly suspected that graph isomorphism might be one of those. As said in [4], graph isomorphism is of great interest since it is one of the few problems contained in NP that is neither known to be computable in polynomial time nor to be NP-complete. Presently, there is no known polynomial-time algorithm for graph isomorphism and further, there is strong evidence that the problem is not NP-complete. Mathon [5] demonstrates that the problem of counting the number of isomorphisms between two labeled graphs is Turing reducible to graph isomorphism; this gives indication that it is unlikely to be NP-complete since for almost all NP-complete problems their counting versions are of much higher complexity than themselves. The inability to find a polynomial-time algorithm for the graph isomorphism problem demonstrates evidence that it is unlikely that the problem is in P.

4 Bibliography

The two first were used to get to know the tree isomorphism problem, the last one was used in order to have an idea on which subclasses were isomorphism-complete.

[1] : *Algorithms on Trees and Graphs*, G. Valiente

[2] : *The Design and Analysis of Computer Algorithms*, A. Aho, J. Hopcrot, J. Ullman

[3] : *Problems Polynomially Equivalent to Graph Isomorphism*, K. Booth, C. Colbourn

[4] : *Graph Isomorphism Completeness for Perfect Graphs and Subclasses of Perfect Graphs*, C. Boucher, D. Loker

[5] : *A note on the graph isomorphism counting problem*, R. Mathon