THESIS

ISOMORPHISMS IN CO-TT GRAPHS

Submitted by

David K. Besen

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

ABSTRACT

ISOMORPHISMS IN CO-TT GRAPHS

A threshold tolerance graph is a graph where each vertex v is assigned a weight w_v and a tolerance t_v, and there is an edge between two vertices v_x and v_y if and only if $w_x + w_y \geq min(t_x, t_y)$. A co-TT graph is the complement of a threshold tolerance graph. Recognition of these graphs can be done in $O(n^2)$ time; however, no polynomial-time algorithm to identify isomorphisms between pairs of TT or co-TT graphs was previously known. We give an algorithm to identify these isomorphisms, which takes $O(n^2)$ time.

ACKNOWLEDGEMENTS

I would like to thank the following people, without whom I would never have succeeded:

Steven Colby, for reminding me that it's possible to find a calling;

Ross McConnell, for trusting that I would find something that genuinely interests me;

Cole Frederick, for his interest and encouragement;

My parents Robert and Linda, for their love and support;

Sanjay Rajopadhye, for unknowingly keeping me on track.

DEDICATION

This work is dedicated to Janusz Okolowicz, the most inspiring middle school math teacher a young nerd could ever wish for.

TABLE OF CONTENTS

CHAPTER 1

BACKGROUND

1.1 General graph isomorphism

A graph is a set of vertices V and a set of edges E such that each edge is associated with exactly two ordered vertices. All graphs are *directed*, and an *undirected* graph is a graph where the edge set is symmetric; that is, if there is an edge from vertex u to vertex v, then there is also an edge from v to u.

If two graphs G and G' have vertex sets V and V', an *isomorphism* is a bijection function $f : V \rightarrow V'$ such that $(f(u), f(v))$ is an edge of G' if and only if (u, v) is an edge of G for all $u, v \in V$. Two graphs that share at least one isomorphism are said to be *isomorphic*. A nontrivial isomorphism from a graph G to itself is called an *automorphism*, and if an automorphism for G exists, G is *automorphic*. Intuitively, isomorphism represents the concept of "sameness" between graphs.

The problem of deciding whether two general graphs are isomorphic is not easy. No polynomial time solution is known, and no proof has been given that it is NP complete. As of this writing, a quasipolynomial time algorithm capable of finding isomorphisms between general graphs has been offered and refined by Babai [2, 3], but has not yet been fully peer reviewed. (Quasipolynomial time means the algorithm runs in between polynomial time and exponential time; the worst case runtime of a quasipolynomial algorithm is $O(2^{(log\, n)^c})$ where the constant $c > 1$.)

In practice, whether or not two finite graphs are isomorphic is often easily solved. For example, if two graphs have a different number of vertices of some degree, they are clearly not isomorphic. Isomorphisms can be detected in polynomial time between graphs in many common graph classes, such as trees [1] and interval graphs [11] (graphs that can be represented by intervals on a line; see Section 1.2.1.). The well-known programs *nauty* and

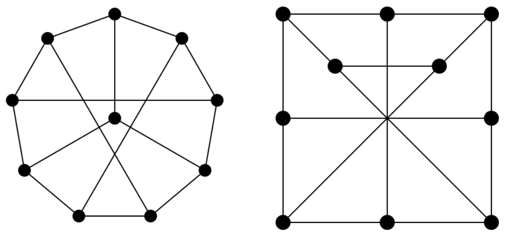


Figure 1.1: Two drawings of the Petersen graph

Traces implement heuristics that often work well in practice to provide isomorphisms for many common graphs [14].

For some graphs, the problem becomes more difficult. The Petersen graph in Figure 1.1 illustrates an example of a problem that can arise. In this graph, all vertices have exactly three neighbors, and yet a random vertex mapping between two Petersen graphs will be unlikely to produce an isomorphism.

1.2 Relevant graph classes, concepts, and definitions

We shall now give some related definitions and descriptions for some related graph classes and ideas.

A *clique*, synonymous with a *complete* graph or subgraph, is a set of vertices such that every pair of vertices in the set are adjacent. A *maximal clique* is a clique that is not a subset of any other clique. A *maximum clique* in G is a clique such that no other clique in G has more vertices.

Let $N(v)$ represent the neighbors of vertex v excluding v itself (the "open neighborhood"), and let $N[v] = N(v) \cup \{v\}$ (the "closed neighborhood"). A *simplicial* vertex is a vertex such that $N[v]$ induces a clique.

1.2.1 Interval graphs

An *interval model* is a set of closed intervals on a line. Each interval represents one vertex of a graph. Two vertices are adjacent if and only if their intervals intersect. An *interval graph* is a graph that can be represented by an interval model. An interval may be represented by two numbers $[a, b]$, where a is the interval's start point and b is the interval's end point.

If G consists of vertices V and edges E, and $V' \subseteq V$, then $G[V']$, the subgraph of G *induced* by V', is vertices V' with edges $E \cap (V' \times V')$. A property of a graph G is *hereditary* if any induced subgraph of G also has that property. For example, any induced subgraph of an interval graph is also an interval graph, since intervals may be removed from the interval model to obtain an interval model of the induced subgraph.

A *chordal* graph is a graph that has no chordless cycle of length 4 or greater.

Theorem 1. *[7] Every interval graph is chordal.*

Proof. Suppose an interval graph G has interval model I_G and has a chordless cycle $C = v_1, v_2, ..., v_k, v_1$ for $k \geq 4$. One interval $[a, b]$ in I_G must end first; without loss of generality, let this interval be v_1. Likewise, one interval $[a', b']$ begins latest; let this interval be v_b. Note $b \neq 2$ and $b \neq k$, and v_1 does not intersect v_b. Then, since C is a cycle, there are two paths between v_1 and v_b; one path $P_1 = v_1, v_2, ..., v_b$ and the other path $P_2 = v_1, v_k, v_{k-1}, ..., v_b$.

Both P_1 and P_2 must cross the range (b, a'). Let c be a point in this range. At least one vertex $u \neq v_1, v_b$ in P_1 must intersect c, since P_1 crosses the entire range. Likewise, at least one vertex $w \neq v_1, v_b$ in P_2 must intersect c. Therefore, since they both contain the point c, u and w intersect, contradicting that C is chordless. $\square$

1.2.2 Asteroidal triples

An *asteroidal triple* is a set of three vertices such that there is a path between any two vertices that does not pass through the neighborhood of the third. The vertices a, b, and c comprise such a triple in Figure 1.2, and this figure is proof that not every chordal graph is an interval graph. In the interval model for this figure, e and f must lie between b and c. Also, d must lie between e and f since it intersects e and f but not b and c. Then there is no place in the model to put a, since a must intersect d but not e and f. Lekkerkerker

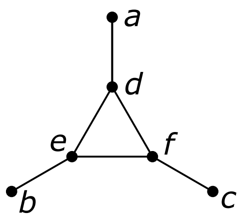


Figure 1.2: Example of a chordal graph that is not an interval graph.

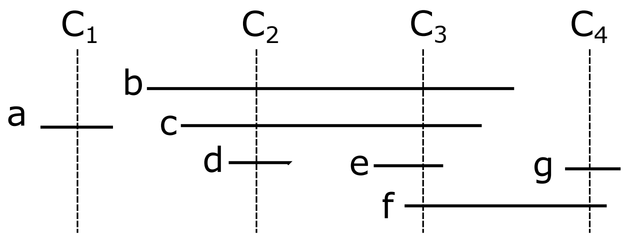
and Boland [10] give a proof that a graph is an interval graph if and only if it is chordal and contains no asteroidal triple.

Lemma 1. *[7] If a graph is chordal, then it contains a simplicial vertex. If it is not complete, then it contains at least two nonadjacent simplicial vertices.*

Proof. If G is complete, then all vertices in G are simplicial. If G is not complete, then we proceed by induction; assume the lemma is true for all graphs with fewer vertices than G. We choose two nonadjacent vertices a and b. Then we choose a minimal set of vertices S such that, if they are removed, they will separate a and b into different components. Let G_S be G with the vertices in S removed, and let A be the graph component in G_S containing a, and let B be the component containing b. It suffices to prove that the lemma applies when G is connected, since the result will follow for each connected component.

Let u and v be two nonadjacent vertices in S. Since S is minimal, every vertex in it must be adjacent to at least one vertex in A and one vertex in B. So, there must be some shortest path from u to v via A $[u, A_1, A_2, ..., A_k, v]$ and some shortest path through B $[u, B_1, B_2, ..., B_m, v]$. These paths form a cycle $[u, A_1, A_2, ..., A_k, v, B_1, B_2, ..., B_m, u]$. Since G is chordal, this cycle must contain a chord. The vertices A_i and B_j are not adjacent since S separates A and B. The vertices A_i can't contain a chord since they form a shortest path, and likewise for B_i. So, the only option is for the chord to be between u and v. All vertices in S form a clique.

Then, if $G[A \cup S]$ is complete, a is simplicial in that subgraph; it is also simplicial in G since a has no neighbors in B. Otherwise, we proceed by induction on the subgraph $G[A \cup S]$. This has two nonadjacent simplicial vertices, one of which must lie in A since S is a clique,



	C_1	C_2	C_3	C_4
a	1	0	0	0
b	0	1	1	0
c	0	1	1	0
d	0	1	0	0
e	0	0	1	0
f	0	0	1	1
g	0	0	0	1

Figure 1.3: Interval graph with maximal cliques C_1 - C_4 and clique matrix

and this vertex is simplicial in G. By symmetry, we can find the second simplicial vertex in B, and these two vertices are nonadjacent. □

Intuitively, interval graphs may not branch into three or more directions (because that creates an asteroidal triple), and they may not circle back on themselves (because that creates a chordless cycle).

1.2.3 The consecutive ones property

A *clique matrix* is a matrix consisting of only ones and zeroes, where each row represents a vertex and each column represents a maximal clique. The entry for a given vertex and clique has a 1 if the vertex is a member of that clique and a 0 otherwise. See Figure 1.3 for an example interval graph with its associated clique matrix.

Since every chordal graph has a simplicial vertex (Lemma 1), and the chordal property is hereditary, this implies a *perfect elimination order* of the vertices in a chordal graph. We choose a simplicial vertex, remove it, and repeat to create this order. Likewise, if a graph has a perfect elimination order, that means it must be chordal, since a chordless cycle of length 4 or greater does not have a simplicial vertex.

For non-chordal graphs, the number of columns in the clique matrix may be exponential in the number of vertices. For example, let G_d be a graph with vertices v_1, v_2, ..., v_{2n}. Then let v_{2i-1} be adjacent to v_{2i} for $i \in 1, 2, ..., n$. So, v_1 is adjacent to v_2, v_3 is adjacent to v_4, etc. Then let G'_d be the complement of G_d. Note G'_d is not chordal. In G'_d, each maximal clique is of size n, and consists of exactly one member of each pair of vertices. A maximal clique may be built by selecting one vertex from each pair. There are 2^n ways to select unique maximal cliques this way, so G'_d contains that many maximal cliques.

For chordal graphs, the number of maximal cliques is bounded by the number of vertices n. Since a chordal graph has a perfect elimination order, the first vertex v in this order is part of exactly one maximal clique of size $s = |N[v]|$. If we remove this vertex and repeat, we can proceed for a maximum of n iterations before we run out of vertices to remove. A vertex may be part of a clique but not part of a maximal clique, so the number of maximal cliques may be less than n. This suggests a method to build the clique matrix for a chordal graph: remove vertices according to the perfect elimination order, and for each vertex that is the beginning of a new maximal clique, add a column to the matrix. To tell if a vertex is the beginning of a new maximal clique, when we encounter a vertex w which is the start of a new maximal clique C (including the first vertex processed), w will have a vertex x which is next in the elimination order. We attach a list of the vertices $N(w) - x$ to x. Then when we move on to x, we check x's neighborhood against that list. If x has any neighbors not in that list, then those neighbors are part of one or more new maximal cliques. We can then continue on in that same fashion; if vertex y follows vertex x, then when we're processing x we attach the list of vertices $N(w) - x - y$ to y, and check the list when we get to y, discovering y's neighbors which start a new maximal clique.

The *consecutive ones property for rows* means the matrix's columns may be permuted in such a way that all of the ones in each row are consecutive.

Theorem 2. *[7] A graph is an interval graph if and only if its clique matrix has the consecutive ones property for rows.*

Proof. If a graph has an interval model, the maximal cliques may be read from the model by finding the points on the number line where a left endpoint p_1 is followed immediately by a right endpoint p_2. The intervals that contain the line segment between p_1 and p_2 form a maximal clique. This clique is a column in our matrix. Taking these columns in the order in which they appear in the model gives a consecutive-ones ordering.

Conversely, given a clique matrix for a graph in consecutive-ones form for rows, the interval model may be read directly from the matrix by drawing each interval directly over each set of consecutive ones. This is a valid interval model since two vertices are adjacent if and only if they're part of a common clique. $\square$

1.2.4 Threshold graphs

An *independent set* of vertices, sometimes referred to as a *stable set*, is a set of vertices such that no two vertices in the set are adjacent. Chvátal and Hammer [5] provide the idea of a *threshold dimension*, described as follows:

Let $V = \{v_1, v_2, ..., v_n\}$ be the set of vertices in an undirected graph G. Then any subset $X \subseteq V$ can be represented by a *characteristic vector* $x = \{x_1, x_2, ..., x_n\}$ where for all i

$$x_i = \begin{cases} 1 & \text{if} \quad v_i \in X, \\ 0 & \text{if} \quad v_i \notin X. \end{cases} \qquad [5]$$

This allows us to plot any subset of V onto the corner of a hypercube in $\mathbb{R}^n$ space. Then, if we plot all subsets of V on this hypercube, we ask the question of whether we can place a hyperplane such that all of the independent sets of V lie on one side of the plane and the other vertices lie on the other side. If we can, then G is a *threshold graph.* Equivalently, a threshold graph is a graph that can be represented by a real number S (a "threshold") and a set of vertex weights w. Then two vertices u and v are adjacent if and only if $w_u + w_v > S$. Note that we may assume w_u, w_v, and S are all positive integers without loss of generality.

Lemma 2. *[5] The complement of a threshold graph is a threshold graph.*

Proof. Assume we have a positive integer vertex labeling w_i and threshold S for a threshold graph $G = (V, E)$. Let $\overline{G} = (V, \overline{E})$ be the complement of G. Then:

$$xy \in E \Leftrightarrow w_x + w_y > S$$

$$xy \notin E \Leftrightarrow S \geq w_x + w_y$$

$$xy \in \overline{E} \Leftrightarrow -w_x - w_y \geq -S$$

$$xy \in \overline{E} \Leftrightarrow (S - w_x) + (S - w_y) \geq S$$

$$xy \in \overline{E} \Leftrightarrow (S - w_x + 1) + (S - w_y + 1) \geq S + 2$$

$$xy \in \overline{E} \Leftrightarrow (S - w_x + 1) + (S - w_y + 1) > S + 1$$

This gives us $\overline{w}_i = (S - w_i + 1)$, and $\overline{S} = S + 1$ as a valid labeling and threshold for $\overline{G}$. $\square$

Since the complement of a threshold graph is a threshold graph, that means threshold graphs have a somewhat symmetric structure – on one side, we have the independent sets of the graph, and on the other side, we have the cliques of the graph. Then the relationship between the sets on these two sides is of a predictable structure.

Let $\delta_0 = 0$. Let $\delta_1 < \delta_2 < ... < \delta_k < |V|$ be the sorted, distinct degrees of the nonisolated vertices in G. For example, if a graph has 3 vertices of degree 0, 5 vertices of degree 1, 0 vertices of degree 2, and 1 vertex of degree 3, then $\delta_0 = 0$, $\delta_1 = 1$, $\delta_2 = 3$, and $k = 2$. Then a *degree partition* for vertex set $V = D_0 + D_1 + ... + D_k$ where D_i is the set of all vertices of degree δ_i. Only D_0 is possibly empty. Chvátal and Hammer [5] also show us that given only the vertices V with their degrees in a graph G, we can determine whether or not G is a threshold graph in time proportional to $|V|$. Let y be a vertex of largest weight in G. Let x be a vertex of degree 1 or greater and w be a neighbor of x. Then, $w_y \geq w_w$, $w_x + w_w > S$, and therefore $w_x + w_y > S$, so x and y are adjacent. This implies all vertices in D_k are adjacent to all vertices in D_1. Each vertex in D_1 has exactly one neighbor. If another vertex z is adjacent to a vertex f in D_1, then z must have a weight $w_z \leq w_y$, and y will necessarily be adjacent to all neighbors of z, contradicting that f is in D_1. So, y must be the only neighbor of every vertex in D_1, and $|D_1| = k$. Then, by induction, we can take the subgraph of G induced by $V' = V - D_0 - D_k$ and repeat. Therefore, if G is a threshold graph, then for all distinct vertices $x \in D_i$ and $y \in D_j$, xy is an edge in G if and only if $i + j > k$. Then

$$\delta_{i+1} = \delta_i + |D_{k-i}| \text{ for } 0 \le i \le \lfloor k/2 \rfloor - 1, \text{ and}$$

$$\delta_i = \delta_{i+1} - |D_{k-i}| \text{ for } k \ge i \ge \lfloor k/2 \rfloor + 1.$$

These recurrences can be used to test whether a graph is a threshold graph using only the degrees of its vertices in time proportional to $|V|$. We first make an array of size $|V|$ and fill it with a count of the vertices of each degree by walking through the list of degrees. Then from that we can calculate the δ_i list and the size of each element in the D_i list. Then we can walk through those two new lists and verify whether or not the recurrences hold.

1.2.5 Threshold tolerance and co-TT graphs

Threshold tolerance graphs (also called *TT graphs*), are similar to threshold graphs, but more general. In threshold tolerance graphs, each vertex v is assigned both a weight w_v and a threshold t_v, and two vertices v_x and v_y are adjacent if and only if $w_x + w_y \ge min(t_x, t_y)$. A threshold tolerance graph is a tolerance graph if all of the vertex tolerances are the same.

A *co-TT graph* is the complement of a threshold tolerance graph.

We can redefine interval graphs as the following:

Definition 1. *[6] A graph $G = (V, E)$ is an interval graph if and only if there exist functions $a, b : V \mapsto \mathbb{R}$ such that:*

- *$a(x) \le b(x)$ for all $x \in V$;*
- *$xy \in E \Leftrightarrow a(x) \le b(y) \wedge a(y) \le b(x)$ for all $x, y \in V$.*

We can also redefine co-TT graphs as the following, which is simply the previous definition with one requirement removed:

Definition 2. *[15] A graph $G = (V, E)$ is a co-TT graph if and only if there exist functions $a, b : V \mapsto \mathbb{R}$ such that:*

- *$xy \in E \Leftrightarrow a(x) \le b(y) \wedge a(y) \le b(x)$ for all $x, y \in V$.*

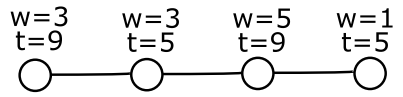


Figure 1.4: TT model of P4, where w is weight and t is threshold

Intuitively, these definitions correspond well with the fact that interval graphs are a subclass of co-TT graphs. In interval graphs, each interval has a starting and an ending point, and the ending point must be after the starting point. If we relax that requirement, and allow the ending point to occur before the starting point, the co-TT graph class results.

Each co-TT graph can be represented by a *co-TT model*, that is, interval models where some of the intervals are reversed. We refer to reversed intervals as "red", and forward intervals as "blue". Note that the model for a graph may not be unique.

True twin vertices v, v' are vertices such that $N[v] = N[v']$, and *false twin* vertices are vertices such that $N(v) = N(v')$.

Since co-TT graphs are the complement of TT graphs, any graph that is isomorphic with its own complement must be in both classes or in neither. For example, a cycle of length 5 is neither TT nor co-TT. It is not co-TT because it is not an interval graph and it has no simplicial vertices. It is not TT because it is its own complement and is not in co-TT.

Another example is that a path of length 4 is both TT and co-TT. It is co-TT because it is an interval graph. It is TT because it is its own complement. In addition, a TT model for it is given in Figure 1.4.

1.2.6 PQ trees

The *PQ tree*, introduced by Booth & Lueker [4], is a data structure which can solve the problem of ordering a matrix to test if it is consecutive-ones and to provide the ordering if so. See Figure 1.5 for an example PQ tree, and Figure 1.6 for its associated clique matrix. The root of the PQ tree represents the entire set of columns, and the leaves of the tree each represent a single column. Each internal node represents a subsequence of this set. Using an internal node in this way allows us to enforce that the leaf descendants of a node are

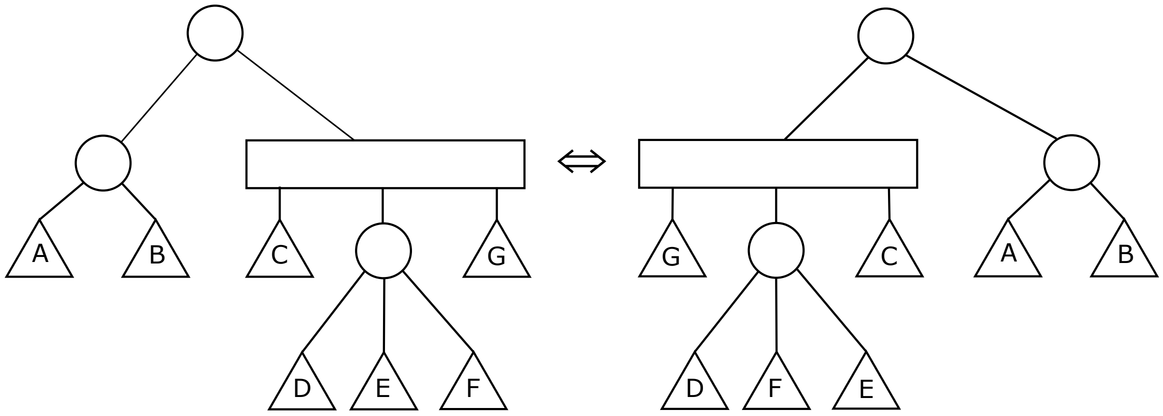


Figure 1.5: Example PQ tree and a valid rearrangement; circles are P nodes, rectangles are Q nodes, and triangles are leaves.

consecutive, since the children of two separate nodes cannot be in an intermixed order when we traverse the tree. PQ trees have two types of internal node: P nodes and Q nodes. A P node's children may be permuted arbitrarily, and a Q node's children may only be reversed. (Note that if a node has only two children, the behavior of the two types of internal node is identical; in this case, we say it is a P node by convention.)

If a graph is empty, its associated PQ tree will contain only one node, and that node must be a P node, since any order is a consecutive-ones ordering. To see why P nodes are alone insufficient and we need Q nodes, examine the clique matrix for a path, shown in Figure 1.7. A path can be represented by a single Q node with each vertex as a child, and in this clique matrix, the columns may be reversed, but the order cannot be permuted arbitrarily without separating two of the ones. With these two internal node types, the PQ tree can simultaneously represent all possible consecutive-ones permutations of the columns in our matrix. When we permute a non-leaf node of the tree according to the P and Q rules, we get a different valid permutation of columns in the leaves. We can read this order by doing a traversal of the tree to collect the leaves.

The PQ tree is built incrementally, by adding each row of the matrix one at a time and rearranging the tree according to the new row to make a new valid tree before moving on to the next row. We begin with a tree that contains a single P node; this tree can

	A	B	C	D	E	F	G
v_1	1	1	0	0	0	0	0
v_2	0	0	1	1	1	1	0
v_3	0	0	0	1	1	1	1

Figure 1.6: Clique matrix for the PQ tree in Figure 1.5

	C_1	C_2	C_3	...	C_{k-2}	C_{k-1}	C_k
v_1	1	0	0		0	0	0
v_2	1	1	0		0	0	0
v_3	0	1	1		0	0	0
...							
v_{k-2}	0	0	0		1	1	0
v_{k-1}	0	0	0		0	1	1
v_k	0	0	0		0	0	1

Figure 1.7: Clique matrix for a path

represent any permutation of the columns of the matrix. Then, one by one, we add each row of the matrix, and rearrange the tree to reflect this new information. The rearrangement is done via a set of 10 template replacements, some quite simple and some more complex, done in careful order at each step of building the tree. Each template consists of a pattern to match inside the tree, and a replacement structure for the part of the tree that the pattern matched. Because of this set of replacements, the PQ tree is sometimes considered a difficult or complex algorithm, and a potentially simpler alternative called the *PC tree* has been provided by Hsu and McConnell [9]. Intuitively, the PQ tree can be thought of as a children's mobile, arbitrarily rotating and rearranging itself in the breeze to represent any particular order of the matrix at a particular time.

When a new row is added to the PQ tree, some of the new elements may be a 1 and some may be a 0. If a node's leaf descendants are all 1 in the new row, we say the node is *full*. If they're all 0, we say the node is *empty*. Otherwise, we say the node is *partial*. See Figure 1.8 for an example node labeling.

We start at the leaves, and work our way up the tree. If a node is full or empty, then no change is necessary for that node. If, however, a node is partial, then some template will

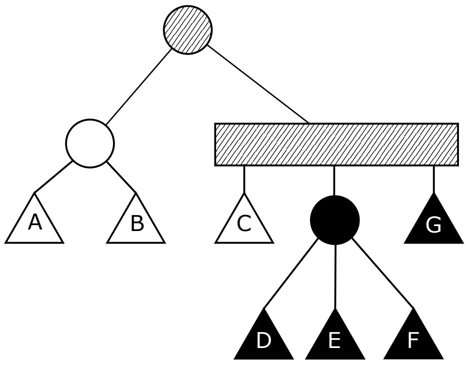


Figure 1.8: Example of PQ tree row labeling, for new matrix row $abcdefg \rightarrow 0001111$; empty nodes are empty, filled nodes are full, and striped nodes are partial.

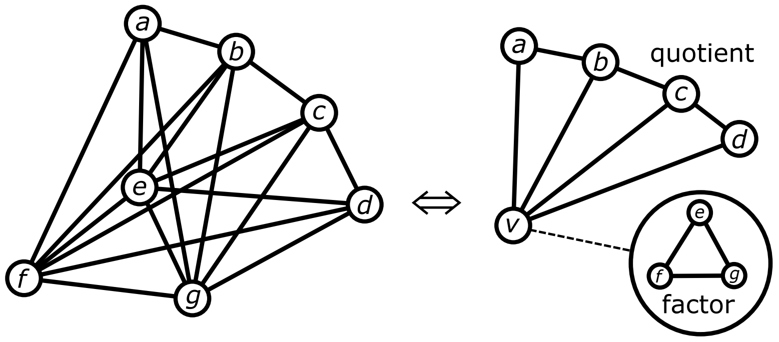


Figure 1.9: Example of one-layer factorization

have to be applied to resolve the conflict. As we're working our way up the tree, we can apply a template to a node without having to worry that the change will affect the node's children. All parents of a partial node are partial, and any change made can have additional effects further up the tree. If at some point we come to a partial node, and none of the templates match, then the matrix is not a consecutive-ones matrix.

The PQ tree solves the problem of ordering a potential consecutive-ones matrix in $O(n+m)$ time [4].

1.2.7 Modular decomposition

We shall now investigate what happens when we use a single vertex of a graph to represent a certain kind of subgraph, as depicted in Figure 1.9. What makes this possible is that the set of vertices in the subgraph are a *module*.

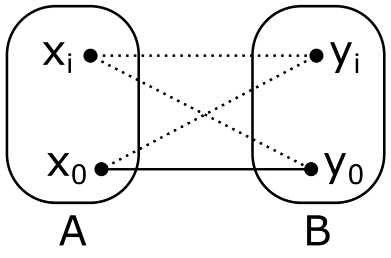


Figure 1.10: Edges between modules

Let graph G with vertices V have subgraph G' with vertices V'. Then V' is a module of G if, for every vertex v in $V - V'$, v is either adjacent to every vertex in V' or adjacent to none of the vertices in V'. If v has at least one neighbor and one non-neighbor in V', then v is a *spoiler* (since it spoils V''s chances of being a module). If V' is a module, then G' is the subgraph induced by V' and is a *factor* of G, and the graph created by replacing G' in G with a single vertex is called a *quotient*. Single vertices are modules, as is the set of all vertices in a graph.

Since a module has no spoilers, we can represent the relationship between the module and a vertex with a single edge or non-edge. In addition, we can represent the relationship between two modules with a single edge or non-edge. See Figure 1.10. Let A and B be disjoint modules of G. Let x_i be the vertices in A, and y_i be the vertices in B. The two vertices x_0 and y_0 (the only two vertices that must exist here) have a relationship; without loss of generality, assume they are adjacent. Then if x_0 and some $y_{i \neq 0}$ are not adjacent, x_0 will be a spoiler for B. So x_0 must be adjacent to every y_i. Via a similar argument, every x_i must be adjacent to y_0, and in addition every x_i must be adjacent to every y_i. So, a single edge or non-edge is sufficient to represent the relationship between two entire modules. This is why we can represent a module with a single vertex in the quotient.

We can perform this modularization of a graph recursively, resulting in a tree. It can be done in a unique, canonical way, called a *modular decomposition*, defined as follows. See Figure 1.11. Given a graph G with vertex set V, the root of the modular decomposition is V, and the n one-element subsets of V are the leaves. The children of V are obtained as follows:

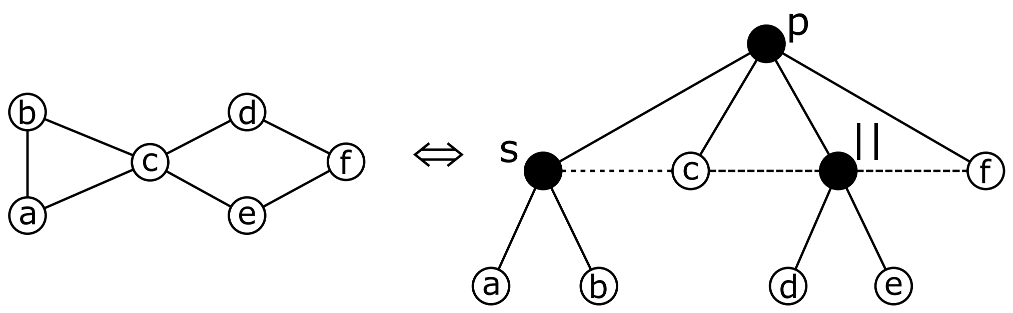


Figure 1.11: Example of a graph with its modular decomposition tree. Prime node is marked p, series node is marked s, and parallel node is marked ||. The annotated quotient of the prime node is shown with dashed lines.

If G is disconnected, its connected components are the children. If $\overline{G}$ is disconnected, the connected co-components of $\overline{G}$ are the children. Otherwise, the maximal modules that are proper subsets of V are a partition of V, and these are the children. The rest of the tree is obtained by recursion on the subgraphs induced by the children that have more than one vertex.

This tree of modules represents every possible way to split the graph into modules at once. In the definition of the modular decomposition, there are three cases, and we shall revisit each case in slightly more detail.

First, consider a graph G with several disconnected components. In this case, our quotient graph will be a disjoint set of vertices, and each one will be associated with a module consisting of the related component in G. However, this simple model is complicated by the fact that a module may consist of *two* of the connected components, even though they are disconnected. Indeed, it is clear that *any* non-empty combination of the connected components of G is a module, and this leads to many possible ways to decompose G into quotient and factors. So, we use the factorization that treats each disconnected component as its own module, and we say the module is a *parallel* module. The children of a parallel module may be selected in any combination to form a module.

Now, consider a graph G whose complement is disconnected. Each co-component has same relationship with every other co-component in the graph. So again we have a case where any combination of children may be selected to be a module. The only difference is that this time all of the module's children are adjacent instead of disconnected. In this case, we say the module is a *series* module.

Lemma 3. *If A, B, and C are disjoint subgraphs of undirected graph G, and both $A \cup B$ and $B \cup C$ are modules of G, then the following are also modules of G:*

- *A, B, and C;*
- *$A \cup C$; and*
- *$A \cup B \cup C$.*

Proof. Let v_A, v_B, and v_C be vertices in A, B, and C, respectively. Assume v_A is adjacent to v_C. Then since $A \cup B$ is a module, v_C is adjacent to v_B. And since $B \cup C$ is a module, v_A is adjacent to v_B. So, all three are a clique. Then if v_A is not adjacent to v_C, via a similar argument, all three are disconnected.

Let V_G represent the vertex set of G, and let v be a vertex in $V_G - (V_A \cup V_B \cup V_C)$. If no such vertex exists, the remainder of the lemma is trivially true. Since $A \cup B$ is a module, v must have the same relationship with v_A as v_B, and likewise for v_B as v_C. So, v must have the same relationship with every vertex in $V_A \cup V_B \cup V_C$. □

There is just one more case left to deal with. If G is a connected graph whose complement is also connected, V is the vertex set of G, and G' is a module of G, then G will contain some vertices outside G' which are adjacent to it, and some vertices outside G' which are not adjacent to it. In this case, there is only one way to factorize into nontrivial maximal modules, and we say it is a *prime* module. In the definition of the modular decomposition, we saw that the children of a prime module are a partition of that module, and this is due to Lemma 3.

Each node's set of leaf descendants defines the module associated with that node. To store the modular decomposition, we don't need to store a list of vertices at each node, since we can collect the vertices for the node by doing a DFS on the node's children and collecting the leaves.

A *strong* module is a module that doesn't properly overlap with any other module, and a *weak* module is one that does overlap. Due to Lemma 3, Every weak module is a union of siblings.

If the modular decomposition tree is annotated properly, it can be a complete representation of its associated graph. Let the *associated quotient* for a node m in a graph G be the graph obtained by replacing every child of m with a single vertex representing that child. For series and parallel nodes, this associated quotient is trivial and can be recreated given only the node type and number of children. For prime nodes, we must store the node's associated quotient in order to reconstruct the original graph. The quotient can be stored as a set of edges between the children of the node, as shown in Figure 1.11.

1.2.8 Δ trees

As we have seen, we can use a single vertex to represent an entire subgraph. We shall now explore the same idea again, but from the perspective of the interval model; that is, we shall use a single interval to represent an entire interval subgraph.

One way to represent a set of intervals on a line is to use a *string representation.* The string representation allows us to represent endpoint orderings without using the numerical start and end points of each interval. The string representation is simply an ordered list of the interval names, where each interval name appears twice in the string. The first time it appears, it represents the start point of that interval, and the second appearance represents the end point. If two intervals a and b overlap, their string representation might be *abab.* If a contains b, the string representation will be *abba.* A string representation may be created easily given an interval model by reading the order of start and end points of the intervals, and an interval model may be created from a string representation by assigning monotonically increasing values to the start and end points of each interval in order.

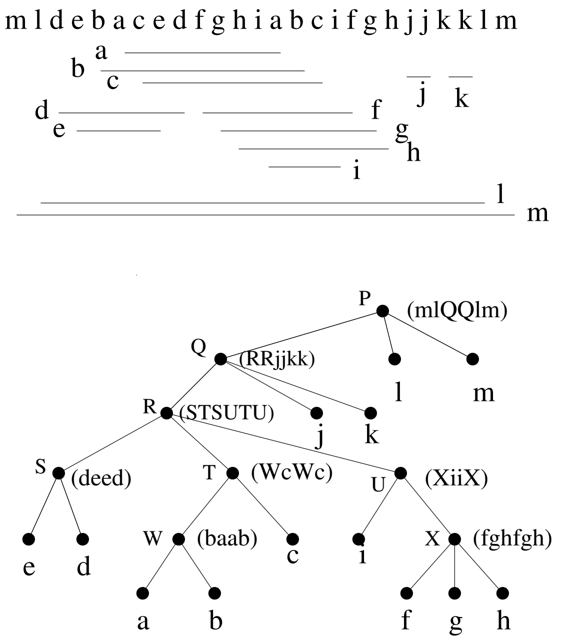


Figure 1.12: [13] Example Δ tree with interval realizer. Letters in parentheses are Δ tree labels indicating the start and end point order of nodes.

We may perform substitutions in the string representation. There are two types of substitution. We shall substitute a string R_i in place of an interval a in string R_j. The first type of substitution is if a's start and end points are adjacent in R_j. In this case, we may simply replace aa with R_i in R_j. For the second type of substitution, R_i must have the property that all of its start points occur before all of its end points. We replace the first occurrence of a in R_j with all of the start points in R_i, and the second occurrence of a with all of the end points in R_i. For example, if $R_1 = abab$, and $R_2 = xyyx$, substituting R_2 in R_1 for a yields the string $xybyxb$.

In a co-TT model, each pair of vertices and their associated intervals u, v have an *intersection type*. There are four possible intersection types for (u, v):

- Non-intersection: u and v do not intersect.
- Overlap: u and v strictly overlap. That is, they share at least one point, but each contains at least one point not inside the other.
- Containment: u contains v.

- Subset: v contains u.

Every ordered pair of vertices must have one of these intersection types. We represent all of the intersection types for a graph in a matrix called the *intersection type matrix*.

A Δ tree (Delta tree) is a tree that can represent all valid interval models for a given intersection type matrix. The Δ tree is similar to the PQ tree; where the PQ tree stores the maximal cliques of a graph as its vertices, and can be thought of as operating on the clique matrix of an interval graph, the Δ tree stores the vertices directly and can be thought of as operating on the interval model. The vertices are stored in the Δ tree leaves, and the Δ tree internal nodes have string representations associated with them. Just as with the modular decomposition, the Δ tree is a quotient-based recursive decomposition of a graph; in this case, it must be an interval graph. The nodes in each layer in the tree contain string representation substitutions for the layer below, and these substitutions correspond with the module-based quotient relationship between a parent and its children. This allows us to reorder large parts of the interval model all at once, by reordering the string representation associated with an internal node. An example Δ tree is given in Figure 1.12. Δ trees may be created in $O(n+m)$ time [12].

The modular decomposition has three different node types (prime, series, and parallel), and the Δ tree has four: prime, linear, degenerate (full), and degenerate (empty). See Figure 1.13 for example interval modules for linear and degenerate node types. Linear nodes are not in the modular decomposition, and are nodes in which the order of the children may not be permuted. These nodes are added by Δ trees since two intervals may have a containment/subset relationship, a relationship which is unimportant in the modular decomposition. Degenerate nodes are nodes where the children may be permuted in any order, and correspond to the modular decomposition series and parallel nodes. Prime nodes are all other nodes, and their children may be reversed. In a Δ tree, just as in a modular decomposition tree, each vertex has the same relationship with every vertex inside a Δ module, and Lemma 3 applies to Δ trees as well.

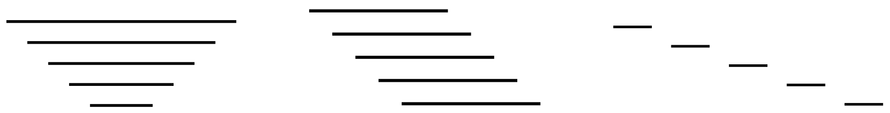

(a) Linear **(b)** Degenerate (full) **(c)** Degenerate (empty)

Figure 1.13: Example interval models for some types of Δ tree nodes

1.2.9 The tree isomorphism algorithm

In [1], an algorithm is given to detect isomorphisms between rooted trees in $O(n)$ time. In this algorithm, we start at the leaves, and we work our way up the tree, layer by layer, labeling each node with a label that completely describes that subtree's structure. The labels at each layer share a data structure between trees, so if two subtrees are isomorphic, they will receive the same label. Each label depends on its children's labels, so when we're done with the labeling procedure the label on the root represents the structure for the entire tree. Then we can compare the root labels to see if two trees are isomorphic.

The labels given to each node have the following requirements in order for the algorithm to work:

- If the tree is permuted, the label must not change.
- If two tree nodes are isomorphic, their labels must be the same.
- If two tree nodes are not isomorphic, their labels must be different.

This means a given node's label must be determined entirely by the structure of the tree below that node. Since we move up the tree layer by layer, the label for a node may be thought of as including the layer number of that node, even though it is not explicitly included in the algorithm in [1].

CHAPTER 2
UNKNOWN ISOMORPHISMS

We now turn to the main new result of this thesis; testing whether or not two co-TT graphs share an isomorphism.

Let $\pi_{A,B}$ represent an isomorphism between A and B. Let G and G' be co-TT graphs that have an unknown isomorphism $\pi_{G,G'}$, and let n be the number of vertices in G, and m be the number of edges. We will show that it is possible to identify $\pi_{G,G'}$ in $O(n^2)$ time.

Theorem 3. *[6] It is possible to recognize co-TT graphs in $O(n^2)$ time.*

If neither G nor G' are co-TT, this method does not apply. If one is co-TT but the other is not, the two graphs are not isomorphic. We will proceed assuming both graphs are co-TT.

2.1 Performing a blue-red division

Golumbic, et. al. [8] give a very convenient method to identify which vertices are red and which are blue in any co-TT graph, restated as the following:

Lemma 4. *[8] If G is a co-TT graph, then there exists a co-TT model where the red vertices are the simplicial vertices that have no true twins in G, and the blue vertices are all others.*

The proof for their characterization proceeds as follows. Red vertices must be simplicial, because for a red vertex u to intersect another vertex v, v must be blue, and must contain u. All vertices that contain u must intersect, so all neighbors of u must intersect, meaning u is simplicial. Then, if blue vertex v is simplicial and does not have a true twin in G, its neighbors must all be blue. If it had a red neighbor u, then u's interval would be contained in v's, implying all neighbors of u intersect v, meaning u and v would be true twins, a contradiction. All the simplicial vertices without true twins may be considered red, since we do not change their adjacency by coloring them red and placing them at the intersection of all their neighbors.

By Lemma 3 of [6], a blue-red partition may be found in $O(n + m)$ time.

Using Lemma 4, we can color the vertices of G. Let B be the set of blue vertices of G, and let R be the set of red vertices. Note that this is a partition of G; each vertex is in exactly one of B or R. Note also that no two red vertices are neighbors with each other, since red vertices cannot intersect in a co-TT model. Likewise, let B' and R' be the vertices in the blue/red division of G'.

Let $G[V]$ denote the subgraph of G induced by the set of vertices V.

Proposition 1. *Let π be an isomorphism between G and G', let U be a subset of the vertices of G, and let U' be their images in G'. Then the mapping between U and U' given by π is an isomorphism between $G[U]$ and $G'[U']$.*

Lemma 5. *An isomorphism, if it exists, must map simplicial vertices to simplicial vertices, and likewise non-simplicial to non-simplicial vertices.*

Proof. Without loss of generality, let π be an isomorphism that maps a simplicial vertex v to a non-simplicial vertex v'. Since π maps v to v', it must also map $N[v]$ to $N[v']$, by Proposition 1. Since v is simplicial, $N[v]$ is complete. But since v' is not simplicial, $N[v']$ is not complete, a contradiction. □

Lemma 6. *All isomorphisms between G and G' map red vertices to red, and blue to blue, when red and blue are as defined by Golumbic, et. al in [8].*

Proof. Without loss of generality, assume two isomorphically mapped vertices v and v' are in B and R'. If v' is in R', then it must be simplicial and have no true twins in G'. Since v and v' are isomorphically mapped, and due to Lemma 5, v must also be simplicial and have no true twins. All vertices in G that are simplicial and have no true twins are in R, so v must be in R, and therefore cannot be in B, a contradiction. □

2.2 Reducing the co-TT graph

Later, we will build a co-TT model that represents both G and G' interchangeably. In order to do this, it will be helpful to partition the graph into equivalency classes that represent the sets of twins in the graph. We will build classes for the true twins if they're in B, and for the false twins in R. (The algorithm for recognizing co-TT graphs in [6] builds the same classes.)

To identify these equivalency classes, we will need to sort all the blue vertices by their closed neighborhoods, and all the red vertices by their open neighborhoods. Note that vertex names here may be different between G and G', and that will affect the order of the sorts, but it will not affect the twin classes.

To sort the vertices by their neighborhoods, we can use a form of radix sort given in [1] that runs in time proportional to the total length of all strings plus the size of the alphabet. In our case, the length of the strings corresponds to the number of edges and the alphabet size corresponds to the number of vertices, so for us it will run in $O(n + m)$.

Let H be G with each set of true twins in B and each set of false twins in R replaced by a representative vertex that has a count of its twin equivalency class members associated with it (a "multiplicity label"). Likewise, let H' be the reduced G'.

A *label-preserving isomorphism* between two reduced graphs (i.e. H and H') is an isomorphism that only maps two vertices if their multiplicity count and color labels (as defined in [8]) are the same.

Lemma 7. *A label-preserving isomorphism between H and H' exists if and only if an isomorphism between G and G' exists.*

Proof. If G and G' are isomorphic, then H and H' are isomorphic, because V_H is a subset of V_G, and by Proposition 1.

If H and H' have a label-preserving isomorphism, then G and G' are isomorphic, because G and G' can be reconstructed from H and H' and their vertex labels, by expanding each vertex that has a multiplicity count into that many twin vertices, where blue vertices are given true twins and red vertices are given false twins. $\square$

2.3 Finding the blue-only interval model

The subgraphs induced by B and B' contain only blue vertices, and are interval graphs by definition.

Lemma 8. *[9] An interval model can be constructed from an interval graph in $O(n+m)$ time.*

Let $\mathcal{I}_B$ and $\mathcal{I}_{B'}$ be interval models for B and B', constructed using any existing method such as PQ trees or PC trees. Note these models may not be the same even if G and G' are isomorphic.

2.4 Creating the red extension of the blue model

The red extension is a concept presented by [6] and is the blue-only interval model with red intervals added. Each red interval is added such that its left endpoint is just to the right of the rightmost left endpoint of any of its neighbors, and its right endpoint is just to the left of the leftmost right endpoint of any of its neighbors. If left endpoints coincide, we order them left-to-right by ascending order of neighborhood size, and if right endpoints coincide, we order them left-to-right by descending order of neighborhood size. It is possible for the newly added red interval to intersect an existing blue interval with which it is not a neighbor, so the red extension for a graph G is not necessarily a co-TT model for G.

Let E be the red extension of $\mathcal{I}_B$ using R. Likewise let E' be the red extension of $\mathcal{I}_{B'}$ using R'.

2.5 Creating the intersection type matrices

Golovach, et. al. [6] provide an algorithm called Co-TT-Model(G', B', R') that produces the intersection types and co-TT model for the given co-TT graph and its red/blue division. (In that paper it is numbered as Algorithm 2; here it is reproduced as Algorithm 1).

```
Data: The graph $G' = (V', E')$ is a co-TT graph with blue-red partition $(B', R')$, $B'$ has no true
      twins, and $R'$ has no isolated vertices or false twins
Result: A labeling of elements of $A_V$ with their intersection types in a co-TT model $\mathcal{I}(B', R')$ of $G'$
for $(x, y) \in A_{V'}$ do
    Find whether $N[x] \subset N[y]$ (Lemma 7);
end
for $(r_1, r_2) \in A_{R'}$ do
    Find whether $N(x) \subset N(y)$ (Lemma 7);
end
for $(b_1, b_2) \in A_{B'}$ (Lemma 10) do
    if $b_1b_2 \notin E'$ then
        Assign $(b_1, b_2)$ to $E_n$ ;
    end
    else if $N[b_1] \subset N[b_2]$ then
        Assign $(b_1, b_2)$ to $A_s$ and $(b_2, b_1)$ to $A_c$ ;
    end
end
for $(b_1, b_2) \in A_{B'}$ do
    if $(b_1, b_2)$ has not already been assigned then
        Assign $(b_1, b_2)$ to $E_o$ ;
    end
end
Construct an interval model $\mathcal{I}_B$ of $G[B']$ realizing these labels (Lemma 8) ;
Let $\mathcal{I}'$ be the red extension of $\mathcal{I}_B$ ;
for $(r_1, r_2) \in A_{R'}$ do
    if $N(r_1) \subset N(r_2)$ then
        Assign $(r_1, r_2)$ to $A_c$ and $(r_2, r_1)$ to $A_s$ (Lemma 8);
    end
    else if $(r_1, r_2) \in E_n(\mathcal{I}')$ then
        Assign $(r_1, r_2)$ to $E_n$ (Lemma 11);
    end
end
for $(b, r) \in B' \times R'$ do
    if $br \in E'$ then
        Assign $(b, r)$ to $A_c$ and $(r, b)$ to $A_s$ (Lemma 1);
    end
    else if $(b, r) \in A_s(\mathcal{I}')$ then
        Assign $(b, r)$ to $A_s$ and $(r, b)$ to $A_c$ (Lemma 12);
    end
    else if $(b, r) \in E_n(\mathcal{I}')$ then
        Assign $(b, r), (r, b) \in E_n$ (Lemma 12);
    end
end
for $(x, y) \in A_{V'}$ do
    if $(x, y)$ has not been assigned then
        Assign $(x, y)$ to $E_o$ ;
    end
end
Apply Lemma 8 to find a co-TT model $\mathcal{I}(B', R')$ of $G'$;
```

Algorithm 1: [6] Co-TT-Model(G', B', R'). Lemma numbers are from [6].

Lemma 9. *The entry corresponding to vertices u and v in the intersection type matrix produced by Co-TT-Model(G', B', R') is determined uniquely by $N[u]$ and $N[v]$. This is true for all entries in the matrix.*

Proof. Several conditionals are used in calculating the entries of the matrix:

- The tests on lines 2, 5, and 11, and 23 are directly based on $N[u]$ and $N[v]$, or $N(u)$ and $N(v)$.

- The tests on line 8 and 31 are based on the entries in E' (the edge set for the source graph) that correspond to u and v.

- The tests on line 16 and 42 are based only on previous tests.

- For the test on line 26, both u and v are red. If $G[N(u) \cup N(v)]$ is not a complete subgraph of G, then u and v do not intersect in the co-TT model. If $G[N(u) \cup N(v)]$ is a complete subgraph of G, then u and v do intersect in the co-TT model. Note this does not mean the vertices of G represented by u and v have an edge between them; if two red intervals intersect in the co-TT model, they do not share an edge in the related graph. So, the results of the test are directly correlated to the neighborhoods of u and v.

- For the test on line 37, one of u and v is red and the other is blue. Assume without loss of generality that u is red and v is blue. Then if $N(u) \nsubseteq N(v)$, the intervals for u and v do not intersect. And if $N(u) \subseteq N(v)$, the intervals for u and v do intersect. So, again, the results of the test are directly correlated to the neighborhoods of u and v.

- For the test on line 34, assume without loss of generality that u is red and v is blue. Then we are testing if u contains v. Each neighbor b' of u is blue and contains u. First assuming u does contain v in the red extension, then u must also contain v in the co-TT model, since v must be contained by b' by transitivity, and u is added by the red extension procedure in such a way as to be contained by b' and to contain v.

For the converse, we reverse the roles of the red extension and the co-TT model. If u contains v in the co-TT model, then it must also contain it in the red extension, since we have added u to the red extension in such a way that it contains v. So, the containment is decided entirely by neighborhoods of u and v, and the fact that we're using a red extension.

□

For isomorphism, we must attach some extra information to the intersection type matrix. Vertices are shown both as rows and as columns in the matrix. For just the rows, we attach each vertex's color and multiplicity count from when we reduced G to H.

Let M and M' be the intersection type matrices built from E and E' as in [6].

Two matrices are isomorphic if their rows and columns can each be permuted in such a way that the matrix values are the same. When the matrix is labeled, we add in the requirement that the row labels (vertex color and multiplicity count) must also be the same in the permuted matrices for the matrices to be isomorphic.

Lemma 10. *The labeled matrices M and M' are isomorphic if and only if G and G' are isomorphic.*

Proof. By Lemma 9, The intersection type for each pair of vertices u, v in matrix M is uniquely determined by $H[N[u] \cup N[v]]$, $\mathcal{I}_B$ and $\mathcal{I}_{B'}$. The closed neighborhoods of selected vertices of isomorphic graphs are isomorphic, by Proposition 1. The matrix rows have the same vertex label data as H and H'. So, M and M' are isomorphic if and only if H and H' are. From Lemma 7, H and H' are isomorphic if and only if G and G' are. So, M and M' are isomorphic if and only if G and G' are. □

2.6 Isomorphisms between Δ trees

Let D and D' be the Δ trees created from M and M' using the method in [13]. Also, attach each vertex's color and multiplicity count to that vertex's leaf in the tree.

In general, two Δ trees are isomorphic if they can be permuted in such a way that their structure is the same. We will add the requirement that the multiplicity labels and vertex colors must also be the same for D and D' to be isomorphic.

We would like to use the tree isomorphism algorithm in Section 1.2.9 to detect whether or not our two Δ trees D and D' are isomorphic. In order to do this, we must construct a label scheme that follows the isomorphism label requirements. We may use a node's type as part of its isomorphism label since the node type is determined entirely from the structure of G.

Given the requirements, the node labels for Δ trees naturally arise as comprising the following information:

- The node's multiplicity count
- The node's color (red or blue)
- The node type (prime, linear, or degenerate)
- Data about the node's children, depending on its node type:

  Prime An ordered list of the labels of the node's children, possibly reversed such that its lexicographic order is smallest.

  Linear An ordered list of the labels of the node's children.

  Degenerate The type of degenerate node (full or empty), and an unordered set of the labels of the node's children.

Theorem 4. *G and G' are isomorphic if and only if the labeled D and D' are isomorphic.*

Proof. We can reconstruct the graph that a Δ tree was built from as follows:

1. Choose an arbitrary permutation of the Δ tree.
2. From the Δ tree, construct the co-TT model.

3. Read vertices, including vertex color and multiplicity count, from the co-TT model.

4. Read edges from the co-TT model based on which intervals intersect.

In our case, this will reconstruct H and H', along with both vertex labels, from D and D'. By Lemma 7, if H and H' are isomorphic, then G and G' are. Therefore, if D and D' are isomorphic, G and G' are as well.

D and D', including labels, were created deterministically from M and M'. So if M and M' are isomorphic, D and D' must also be. By Lemma 10, if G and G' are isomorphic, M and M' are too, so if G and G' are isomorphic, D and D' also are. □

By Theorem 4, applying the tree isomorphism algorithm described in Section 1.2.9 to the labeled Δ trees allows us to determine if G and G' are isomorphic.

2.7 Obtaining $\pi_{H,H'}$

To create the isomorphism to return, we must permute D and D' into the same order. To do this, we can radix sort each tree node's children using the Δ tree labels created in Section 2.6. Then traverse the trees using any reasonable method (such as in-order traversal) to create two lists of vertices. These two lists, when mapped in order, are $\pi_{H,H'}$.

2.8 Obtaining $\pi_{G,G'}$

To obtain $\pi_{G,G'}$ from $\pi_{H,H'}$, we need to expand each class of twins. Any bijection that maps an isomorphic class of either true or false twins between the two graphs will be part of a correct isomorphism. So, we need to identify which classes of twins are isomorphic using $\pi_{H,H'}$; then, we can expand each to their vertices in G and G' by arbitrarily pairing vertices that were removed in the creation of H and H'.

2.9 Algorithm

The entire process, from start to finish, is given in Algorithm 2 and Algorithm 3. Doing the initial co-TT checks and building the Δ trees is $O(n^2)$ by [6], labeling the Δ trees is $O(n)$, the tree isomorphism algorithm is $O(n)$ by [1], and expanding the twins in the isomorphism is $O(n)$. So, the overall process is $O(n^2)$.

```
   Data: A graph $G$
   Result: A $\Delta$ tree labeled for isomorphism
1  $H$ = reduction of $G$ (Section 2.2);
2  $(R, B)$ = identify red and blue vertices of $H$ (Section 2.1);
3  $\mathcal{I}_B$ = interval model for $B$ (Section 2.3);
4  $E$ = red extension of $\mathcal{I}_B$ and $R$ (Section 2.4);
5  $M$ = intersection type matrix for $E$ (Section 2.5);
6  $D$ = $\Delta$ tree for $M$ (Section 1.2.8);
7  return isomorphism-labeled $\Delta$ tree for $D$ (Section 2.6);
```

Algorithm 2: MakeLabeledDeltaTree

```
Data: Two graphs $G$ and $G'$
Result: An error if neither graph is co-TT, or the isomorphism between the graphs,
        if it exists.
numCoTT = 0;
if $G$ is co-TT then
    numCoTT++;
end
if $G'$ is co-TT then
    numCoTT++;
end
if numCoTT == 0 then
    exit error;
end
else if numCoTT < 2 then
    return not isomorphic;
end
$D$ = MakeLabeledDeltaTree($G$) (Algorithm 2);
$D'$ = MakeLabeledDeltaTree($G'$);
if not treeIsomorphism($D$, $D'$) (Section 1.2.9) then
    return not isomorphic;
end
$\pi_{H,H'}$ = build the isomorphism from $\Delta$ trees $D$ and $D'$ (Section 2.7);
$\pi_{G,G'}$ = expand the multiplicity count of $\pi_{H,H'}$ (Section 2.8);
return $\pi_{G,G'}$;
```

Algorithm 3: Determine if two co-TT graphs are isomorphic, and if so, return the isomorphism.

BIBLIOGRAPHY

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[3] László Babai. Fixing the upcc case of split-or-johnson. http://people.cs.uchicago.edu/~laci/upcc-fix.pdf, 2017. Accessed December 2018.

[4] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, December 1976.

[5] Václáv Chvátal and Peter L. Hammer. *Set-packing and threshold graphs.* University of Waterloo CORR 73-21, Canada, 1973.

[6] Petr A. Golovach, Pinar Heggernes, Nathan Lindzey, Ross M. McConnell, Vinícius Fernandes dos Santos, Jeremy P. Spinrad, and Jayme Luiz Szwarcfiter. On recognition of threshold tolerance graphs and their complements. *Discrete Applied Mathematics*, 216:171–180, 2017.

[7] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57).* North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[8] Martin Charles Golumbic, Nirit Lefel Weingarten, and Vincent Limouzy. Co-tt graphs and a characterization of split co-tt graphs. *Discrete Appl. Math.*, 165:168–174, March 2014.

[9] Wen-Lian Hsu and Ross M. McConnell. Pc trees and circular-ones arrangements. *Theor. Comput. Sci.*, 296(1):99–116, March 2003.

[10] C. Lekkekerker and J. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51(1):45–64, 1962.

[11] George S. Lueker and Kellogg S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26(2):183–195, April 1979.

[12] Ross M. McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37(2):93–147, Oct 2003.

[13] Ross M. McConnell and Jeremy P. Spinrad. Construction of probe interval models. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 866–875, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[15] C. L. Monma, B. Reed, and W. T. Trotter. Threshold tolerance graphs. *Journal of Graph Theory*, 12(3):343–362, 1988.

[16] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363.