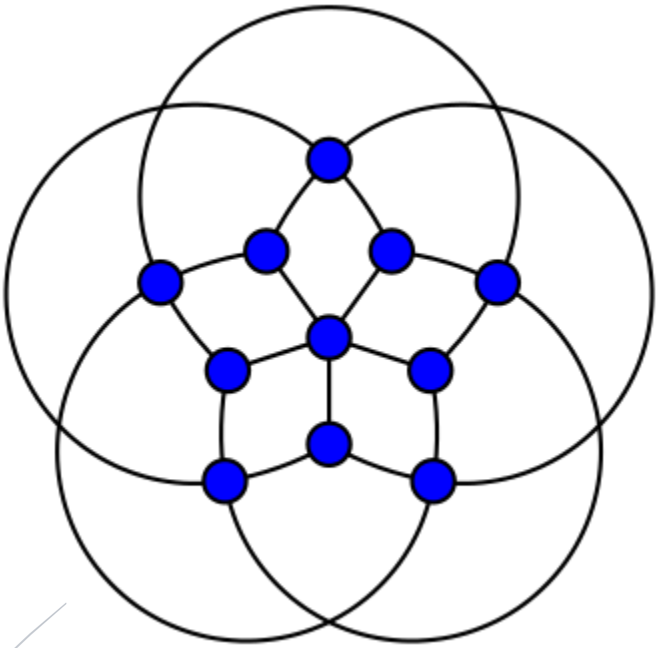
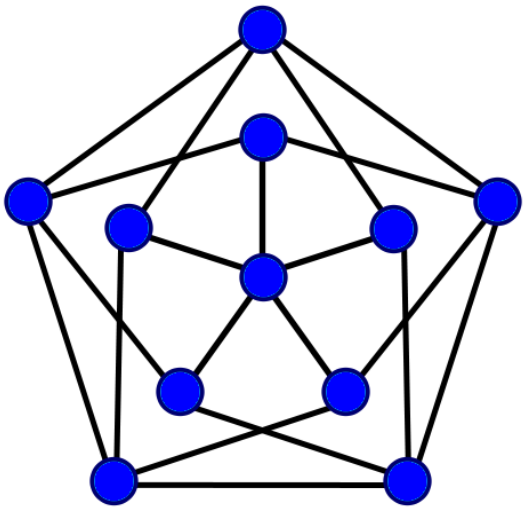
Recognizing Isomorphic Graphs





By: Vladi Shtompel, Tarek Khalaily
Supervisor: Dr. David Tankus

Table of Contents

Chapter 1: Introduction to Graph Theory and Isomorphism

1.1 Graphs

Graphs are abstract mathematical structures used to model pairwise relations between objects. A graph in this context is a set of items connected by edges. Each item is called a vertex. Formally, a graph is a set of vertices and a binary relation between vertices, adjacency. A graph may be *directed* or *undirected*.

Undirected - There is no distinction between the two vertices associated with each edge, meaning that the edge (a, b) and the edge (b, a) are the same.
Intuition: Facebook friendship. If you are my friend, I'm your friend as well.

Directed - Edges have direction, so in this case the edges (a, b) and (b, a) are not the same edge.
Intuition: Twitter following. If I follow you, it doesn't necessarily mean that you also follow me (although you could) and vice versa.

In our project, we worked with both types.
Checking for isomorphism on undirected graphs, while some of the algorithms used only work on directed graphs, so conversion between the two was necessary.

Formal definition

A graph G can be defined as a pair (V, E), where V is a set of vertices, and E is a set of edges between the vertices such that $E \subseteq \{(u, v) \mid u, v \in V\}$. If the graph is undirected, the adjacency relation defined by the edges is symmetric, or $E \subseteq \{\{u, v\} \mid u, v \in V\}$ (sets of vertices rather than ordered pairs). If the graph is undirected then self-loops are not allowed and adjacency is irreflexive (for example, the edge (a, a) cannot exist).

Example 1.1.1:
$G_1(V_1, E_1)$
$V_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$E_1 = \{(1, 2), (1, 4), (1, 5), (3, 2), (6, 2), (3, 4), (3, 7), (4, 8), (5, 6), (6, 7), (7, 8), (8, 5)\}$

G_1 is an *undirected* graph!

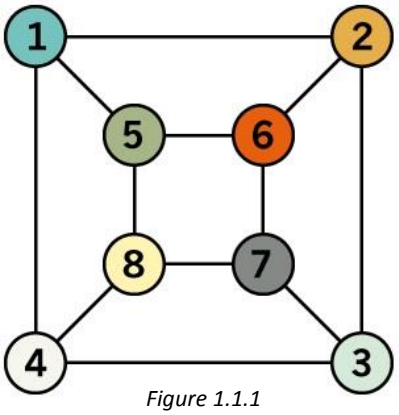


Figure 1.1.1

1.2 Graph Isomorphism

The problem of graph isomorphism can be simply explained as follows:
Do two graphs that appear to look different, actually have the same structure?

Formal definition

Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two graphs. An isomorphism is a bijection F between the vertex sets V_1, V_2 such that:

$$F: V_1 \rightarrow V_2$$
$$\forall\, u, v \in V_1 : (u, v) \in E_1 \Leftrightarrow \big(F(u), F(v)\big) \in E_2$$

Example 1.2.1:

$G_2(V_2, E_2)$
$V_2 = \{a, b, c, d, g, h, i, j\}$
$E_2 = \{(a, g), (a, h), (a, i), (b, g), (b, h), (b, j), (c, g), (c, i), (c, j), (d, h), (d, i), (d, j)\}$

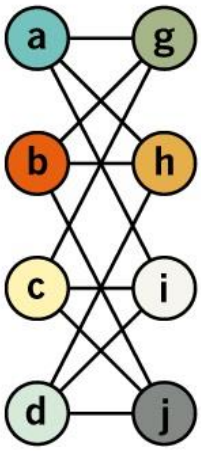


Figure 1.2.1

G_1 from the previous example (*figure 1.1.1*) is Isomorphic to G_2 from this example.
The Isomorphism is:

$$1 \rightarrow a$$
$$2 \rightarrow h$$
$$3 \rightarrow d$$
$$4 \rightarrow i$$
$$5 \rightarrow g$$
$$6 \rightarrow b$$
$$7 \rightarrow j$$
$$8 \rightarrow c$$

Computational Complexity

The problem is not known to be solvable in polynomial time nor to be NP-complete, at the same time, isomorphism for many special classes of graphs (in our project, trees) can be solved in polynomial time, and in practice graph isomorphism can often be solved efficiently.

This problem is a special case of the subgraph isomorphism problem, which is known to be NP-complete.

Since the graph isomorphism problem is neither known to be NP-complete nor known to be tractable, researchers have sought to gain insight into the problem by defining a new class GI, the set of problems with a polynomial-time Turing reduction to the graph isomorphism problem. If in fact the graph isomorphism problem is solvable in polynomial time, GI would equal P.

Applications

Graphs are commonly used to encode structural information in many fields, including computer vision, chemistry and pattern recognition just to name a few. Some concrete examples are:

- Cryptography: zero knowledge proofs.
- In automata theory: multiple uses, mainly to show that some two languages are equal.
- In verification of many things: computer programs, logic proofs, or even electronic circuits.
- In compilers: to perform various optimizations.
- Civil engineering, city planning, building interior planning (this may fall into "search" category as well, e.g. recognizing geographic locations with desired properties).
- Analysis of social structures (special cases may include schools, military, parties, events, etc.), big part of it is search again.
- Any system that performs clustering would benefit from fast graph-isomorphism algorithm, e.g.
    - Linking two Facebook accounts of the same person
    - Recognizing web users based on their behavior
    - Recognizing plagiarism in students' solutions

1.3 Trees

Formal definition

A graph $G = (V, E)$ is a tree $T = (V, E)$ if it satisfies any two of the following conditions:

- G is connected
- There are no cycles in G
- $|V| = |E| - 1$

In other words, a tree $T = (V, E)$ is a connected acyclic graph. There is a unique path between every pair of vertices in T. A tree with N number of vertices contains $(N - 1)$ number of edges.

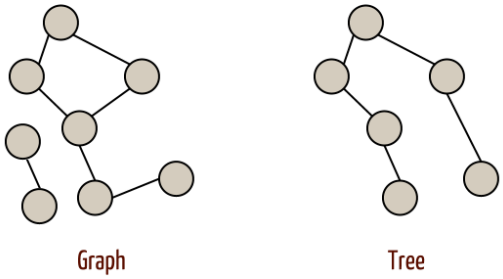


Figure 1.3.1

Terminology used in trees

Root - The top node in a tree.

Child - A node directly connected to another node when moving away from the Root.

Parent - The converse notion of a child.

Descendant - A node reachable by repeated proceeding from parent to child.

Ancestor - A node reachable by repeated proceeding from child to parent.

Leaf - A node with no children.

Level - The level of a node is defined by 1 + (the number of connections between the node and the root).

Height of node - The number of edges on the longest path between that node and a leaf.

Height of tree - The height of a tree is the height of its root node.

Depth - The depth of a node is the number of edges from the tree's root node to the node.

Rooted Tree - A rooted tree is a tree in which one vertex has been designated the root.

1.4 Rooted Trees & Isomorphism

Formal definition

Let $T_1 = (V_1, E_1, r_1)$ and $T_2 = (V_2, E_2, r_2)$ be two rooted trees.
An isomorphism would then be a bijection F between the vertex sets V_1, V_2 such that:

$$F: V_1 \rightarrow V_2$$
$$\forall\, u, v \in V_1 : (u, v) \in E_1 \Leftrightarrow \big(F(u), F(v)\big) \in E_2 \ \wedge\ F(r_1) = r_2$$

Therefore, it is important to note that two trees may be isomorphic as ordinary trees, but not as rooted trees, as shown below:

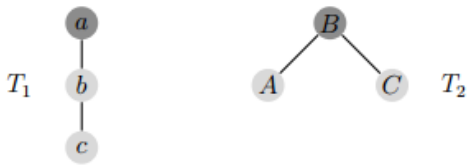


Figure 1.4.1
'a' is the root for T_1 and 'B' is the root for T_2

For this reason, in our project we are rooting the trees at their *centers*.

Tree Center - The center of a tree is a vertex with minimal eccentricity. The eccentricity of a vertex v in a tree T is the maximum distance between the vertex v and any other vertex of the tree. The maximum eccentricity is the tree diameter. If a tree has only one center, it is called Central Tree and if a tree has two centers, it is called Bi-central Tree. Every tree is either central or Bi-central.

A simple way to find the center of the tree, is to repeatedly remove the leaves until only one or two are left, at which point we have found our center (or Bi-center).

Example 1.4.1:
A demonstration of the process of finding the center of some arbitrary tree over all of its iterations is shown in the figure below, iterating from left to right.

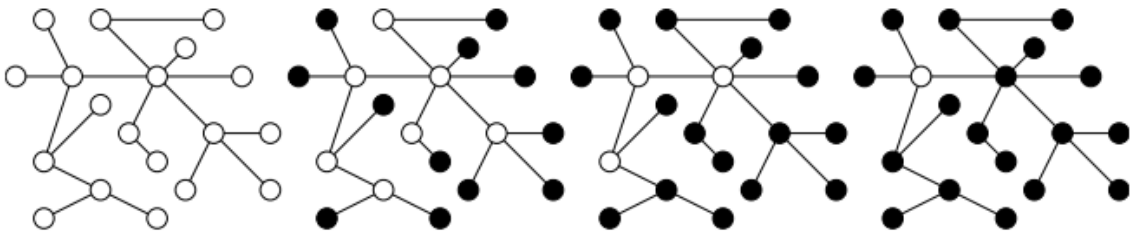


Figure 1.4.2

The black circles represent the vertices that have been removed from the tree. After three iterations, all of the leaves are removed and we finally discover the center of the tree, represented by the only remaining white circle.

1.5 Bipartite Graph

A bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V (that is, U and V are each independent sets) such that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph.
Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles.

Formal definition

A graph $G = (V, E)$ is bipartite if there exist $V_1, V_2 \subseteq V$ such that:

$$V = V_1 \cup V_2 \ \wedge V_1 \cap V_2 = \emptyset \ \wedge \ \forall (u, v) \in E \rightarrow u \in V_1, v \in V_2$$

Example 1.5.1:

$G = (V, U, E)$

$V = \{v_0, v_1, v_2, v_3, v_4\}$

$U = \{u_0, u_1, u_2, u_3, u_4\}$

$E = \{(v_0, u_0), (v_0, u_3), (v_0, u_4), (v_1, u_0), (v_1, u_1), (v_2, u_0), (v_2, u_1), (v_2, u_2), (v_3, u_1), (v_3, u_2), (v_3, u_3), (v_4, u_2), (v_4, u_3), (v_4, u_4)\}$

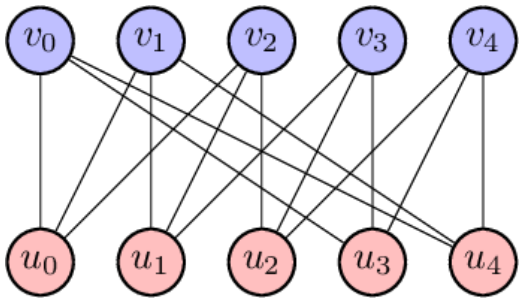


Figure 1.5.1

Matching

A matching in a graph is a subset of its edges, no two of which share an endpoint. In many cases, matching problems are simpler to solve on bipartite graphs than on non-bipartite graphs.

Example 1.5.2:

suppose that a set P of people are all seeking jobs from among a set of J jobs, with not all people suitable for all jobs. This situation can be modeled as a bipartite graph (P, J, E) where an edge connects each job-seeker with each suitable job. A perfect matching, marked in red in the figure below, describes a way of simultaneously satisfying all job-seekers and filling all jobs.

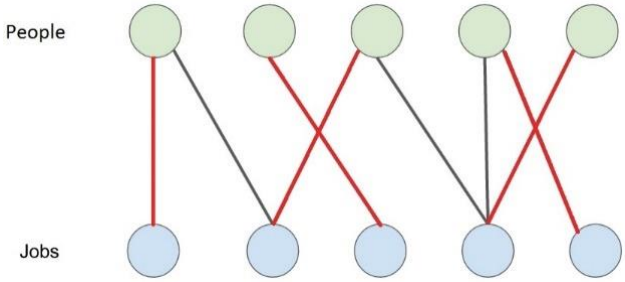


Figure 1.5.2

1.6 Flow Network

In graph theory, a flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge.

A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow.

A network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

Formal definition

A flow network is a tuple $N = (G, c, s, t)$ where:
$G = (V, E)$ is a directed graph of vertices V and directed edges E.
$c: V \times V \rightarrow \mathbb{R}_0^+$ is a mapping from the edges to the non-negative reals, $c(e)$ is the capacity of edge e.
$s, t \in V$ are special vertices of G called the source and sink, respectively.

A flow f for a flow network N is a mapping $f: V \times V \rightarrow \mathbb{R}$ satisfying the following constraints:

$$\forall\, u, v \in V: 0 \leq f(u, v) \leq c(u, v)$$

$$\forall\, u, v \in V: f(u, v) = -f(v, u)$$

$$\forall\, u \in V \setminus \{s, t\}: \sum_{\substack{v \in V \\ f(v,u)>0}} f(v, u) = \sum_{\substack{v \in V \\ f(u,v)>0}} f(u, v)$$

The size of f is defined as:

$$|f| = \sum_{u \in V} f(s, u) = \sum_{u \in V} f(u, t)$$

Example 1.6.1:

The notation for the edges is $flow/capacity$.

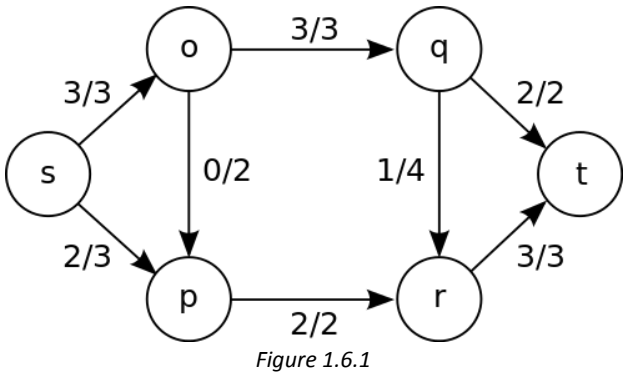


Figure 1.6.1

Chapter 2: How our program processes input Graphs

2.1 File Location and Format

Your files for the two graphs you want to check should both be ".txt" files containing the following information in text:

Number of vertices in the graph, followed by the adjacency matrix of the graph, like so:

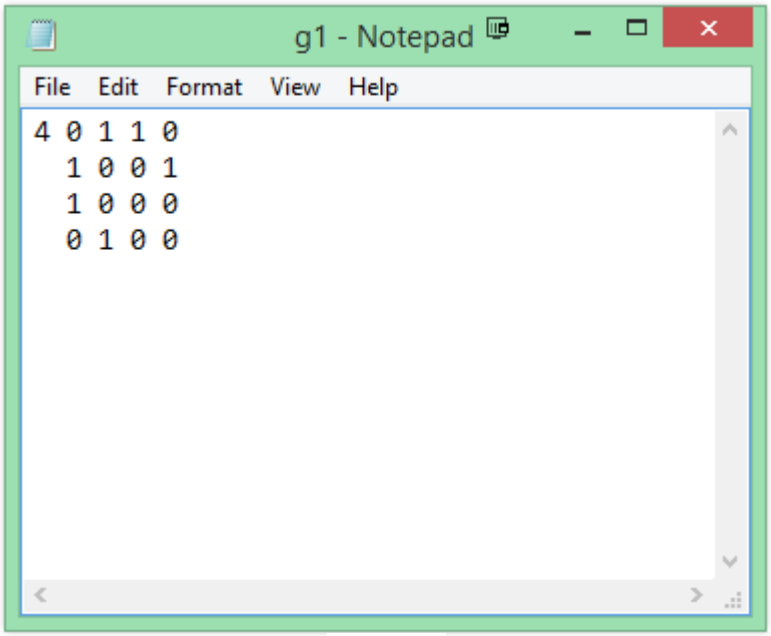


Figure 2.1.1

Note that the indentations and new lines aren't necessary.

By default, your' files should be named "g1.txt" and "g2.txt" and should be located in the "Graphs" folder in the project's main directory.
However, this can be changed to a location of your preference in the "Driver.java" file:

```
public class Driver {

    public static void main(String[] args)
    {
        Graph g1, g2;

        try
        {
            g1 = new Graph("Graphs/g1.txt");
            g2 = new Graph("Graphs/g2.txt");
        }
        catch(Exception e)
        { System.out.println(e.getMessage()); return; }
```

Figure 2.1.2

2.2 Graph Properties

The matrices must both be symmetric and anti-reflexive, otherwise an error will be displayed and the program won't run.

If the input is correct, the program will scan the matrices and construct 2 "Graph" objects.

Aside from asserting that each graph's matrix is of the correct size, and that it is indeed symmetrical and anti-reflexive, the program will also analyze the graph and store its properties.

For each vertex, a Vertex object is created. First the program will go over the matrix and count the rank (number of neighbors) of each vertex as well as construct an ArrayList<Vertex> of aforementioned neighboring vertices.

The program will check how many triangles and squares are in the graph and will also record in how many of those triangles and squares each vertex appears.

The program will also count edges.

Each Vertex object holds a list of its neighbors, its own rank, and its own name (as Integer).
In addition to those generic properties, it will also hold some properties that are required for the different algorithms, which we shall discuss at length later.

The last checks to perform is to see if $|V| = |E| - 1$ and check for cycles using a slightly modified BFS algorithm.

Once this is done we can assert if the graph is a tree or not, and then use the appropriate algorithms.

```
public class Graph
{

    public int [][] mat;    //the 0-1 matrix
    public int v_num;       //number of vertices
    public int e_num;       //number of edges
    public int tri_num;     //number of triangles in graph
    public int squ_num;     //number of squares in graph

    public boolean tree;        //true\false
    public boolean anti_reflex; //should always be true
    public boolean symmetric;   //should always be true

    public ArrayList<Vertex> vertices;  //a list of the graph vertices as Vertex objects
```

Figure 2.2.1

Chapter 3: Tree Isomorphism Algorithm

3.1 Rooting the Trees

If it has been asserted that the graph is indeed a tree, our program will convert it into the Tree object which is an extension of the Graph object.

Then we find the tree's center(s) and root them.
In the case that the trees both have just one center, we choose it as the root.
If both trees have two centers, say $\{u, v\}$ for the first tree and $\{x, y\}$ for the second, then any isomorphism will map u to either x or y. Thus, we root the first tree at u and the second at x. If the rooted isomorphism algorithm is successful then we return the resulting isomorphism; otherwise, we root the second tree at y and run the algorithm again. If the trees turn out to have different numbers of centers, they cannot be isomorphic.

Example 3.1.1:
Here we have the same tree that appears in *figure 1.4.2*, we have colored the vertices in accordance to the iteration in which they were removed when searching for the center.
The outermost leaves are in blue, then orange, then green and finally the center is in red.

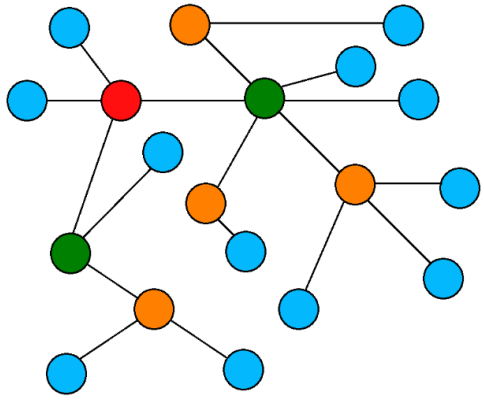


Figure 3.1.1

Here is the same tree after it has been rooted:

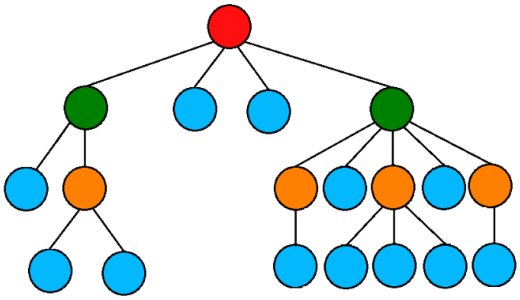


Figure 3.1.2

Once rooted the trees are ready for *Aho, Hopcroft & Ullman's* algorithm, which runs in a time that is linear in proportion to the sum of the numbers of vertices in both trees: $O(|V_1| + |V_2|)$.

3.2 Aho, Hopcroft & Ullman's Algorithm

The algorithm receives two rooted trees as input and returns the isomorphism mapping if they are indeed isomorphic, and an empty mapping if they are not.

The algorithm assigns integers to the vertices of the two trees, starting with the leaves at the deepest level and working up towards the roots, in such a way that the trees are isomorphic if and only if their root's ordered lists of children's labels are the same.

Example 3.2.1

Each vertex holds an ordered list of its children's labels. Leaves have no children, so their lists are empty. Starting at the deepest level, the algorithm looks at the lists of the vertices at that level, and assigns an integer value to each vertex based on how his list compares lexicographically against the lists of the other vertices at the same level.

Since all leaves have the same empty list, they are all assigned the value 0.

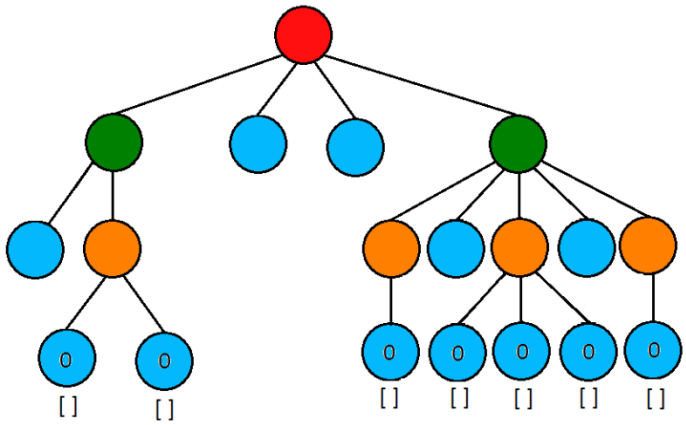


Figure 3.2.1

Then, at the next level leaves are again assigned 0, for their lists are empty.
Non-leaf vertices are assigned different values, based on their lists. Some hold the list $[0]$, one vertex holds the list $[0, 0]$, and the last one hold $[0, 0, 0]$. Lexicographically, 0 comes before 00, which comes before 000, so the vertices holding $[0]$ are assigned 1, the vertex holding $[0, 0]$ is assigned 2, and finally the vertex holding $[0, 0, 0]$ is assigned 3.

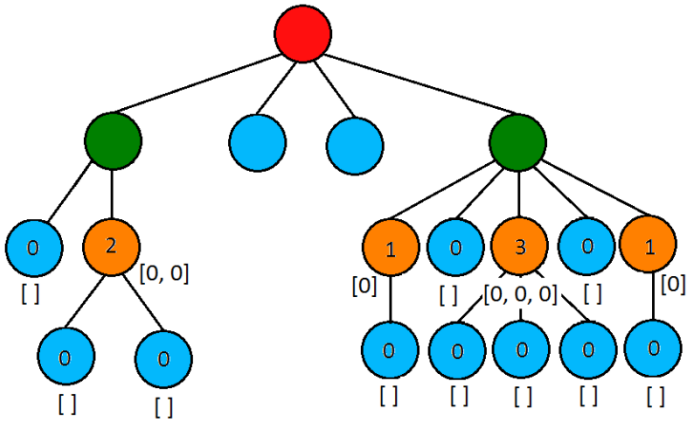


Figure 3.2.2

At the next level, we yet again have some leaves that get assigned 0.
Then we have a vertex holding the list $[0, 2]$ and another one with $[0, 0, 1, 1, 3]$.
Sine 00113 precedes 02 lexicographically, that vertex gets assigned 1 and the other one gets 2.

Finally, the root holds a list of integers: $[0, 0, 1, 2]$.

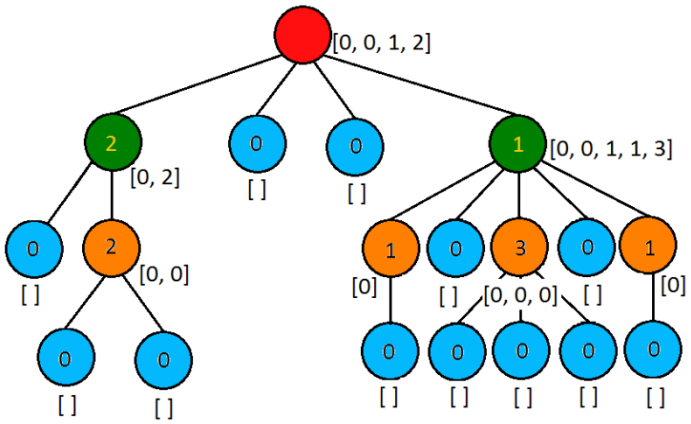


Figure 3.2.3

This process is applied on both trees simultaneously, such that the algorithm compares labels from both trees at the same level. Since the order of the children is a tree isomorphism invariant, it is enough to know that at each level, for each vertex at that level in T_1, we can match its list of ordered children labels with a vertex at same level in T_2.

The trees are Isomorphic if and only if the roots hold equal lists at the end.

If Isomorphism has been asserted, the mapping $F: V_1 \rightarrow V_2$ is generated.

```
private void generateMapping(Vertex v, Vertex w, List<List<Vertex>> M)
{
    ArrayList<Vertex> pair = new ArrayList<Vertex>();
    pair.add(v);
    pair.add(w);
    M.add(pair);
    for(int i = 0; i<v.orderedchildren.size(); i++)
    {
        Vertex x = new Vertex(v.orderedchildren.get(i));
        Vertex y = new Vertex(w.orderedchildren.get(i));
        generateMapping(x, y, M);
    }
}
```

Figure 3.2.4

This sub-routine maps T_1 to T_2 recursively, starting at the roots and working its way down to the lowest levels, pairing each vertex in T_1 with a matching one from T_2.

Chapter 4: General Graph Isomorphism Algorithm

4.1 Looking for Similarities

If the graphs both aren't trees, they will be checked by the general algorithm.

At first, we check that both graphs have the same numbers of vertices, edges, triangles and squares since all of these are Isomorphism invariant.

If this first stage didn't rule out isomorphism, then we start comparing vertices.
Each vertex in G_1 is compared against each vertex in G_2. We compare their ranks, in how many of the triangles and squares in the graph they each appear in, as well as ordered lists of their neighbors ranks, appearance in triangles and squares, since all of those are isomorphism invariants as well.

Suppose we performed the aforementioned checks on $v \in V_1$ and $w \in V_2$ and everything was equal. This means that w is a "possible candidate" to be mapped to v, so it will be added to v's list of possible candidates. Each vertex holds such list, and after comparing all vertices in G_1 against all vertices in G_2 those lists are complete.

Example 4.1.1:
Let us look at the following isomorphic graphs:

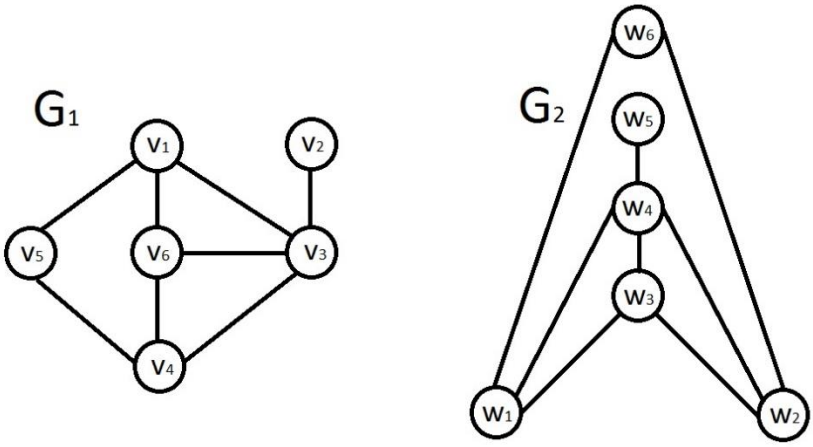


Figure 4.1.1

After checking for vertex similarities, the lists of candidates for G_1's vertices are:

Vertex:	v_1	v_2	v_3	v_4	v_5	v_6
Candidate(s):	w_1, w_2	w_5	w_4	w_1, w_2	w_6	w_3

Should any of those lists be empty, isomorphism is immediately ruled out, otherwise we start looking for possible mappings.

4.2 Looking for a Single Possible Match

Constructing Gb

Once the lists are complete, we use them to construct a new, bipartite graph $G_b = (V, U, E)$, as follows:

$$V = V_1$$
$$U = V_2$$
$$(v, u) \in E \Leftrightarrow u \in v.PossibleCandid$$

Such that V_1, V_2 are the sets of vertices for G_1, G_2 respectively and $v.PossibleCandid$ is v's list of possible candidates for mapping.

Example 4.2.1:

Here is G_b for the graphs G_1, G_2 from the previous example:

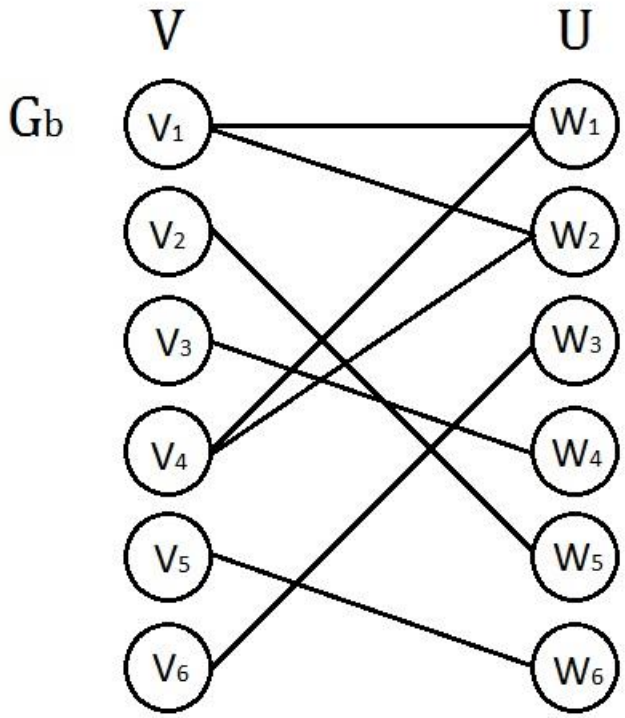


Figure 4.2.1

We will be working with G_b soon enough, but for now we will use it to construct yet another new graph, a flow network G_{fn}.

Constructing Gfn

We shall now build upon $G_b = (V, U, E)$ and turn it into a flow network $G_{fn} = (V', E')$:

$$V' = V \cup U \cup \{s, t\}$$
$$E' = \{(s, v) | v \in V\} \cup \{(u, t) | u \in U\} \cup E$$
$$\forall v \in V: c(s, v) = 1$$
$$\forall u \in U: c(u, t) = 1$$

The edges in E must be directed from V to U and can have arbitrary positive capacity.
In our program we used 1000, but it really makes no difference due to the capacity restrictions on the other edges.

Thanks to the restrictions on the edges from s and to t, by finding a maximum flow f_{max} on the network such that $|f_{max}| = |V| = |U|$ we effectively find a perfect matching.

Example 4.2.2:

Here is G_{fn} built upon G_b from the previous example:

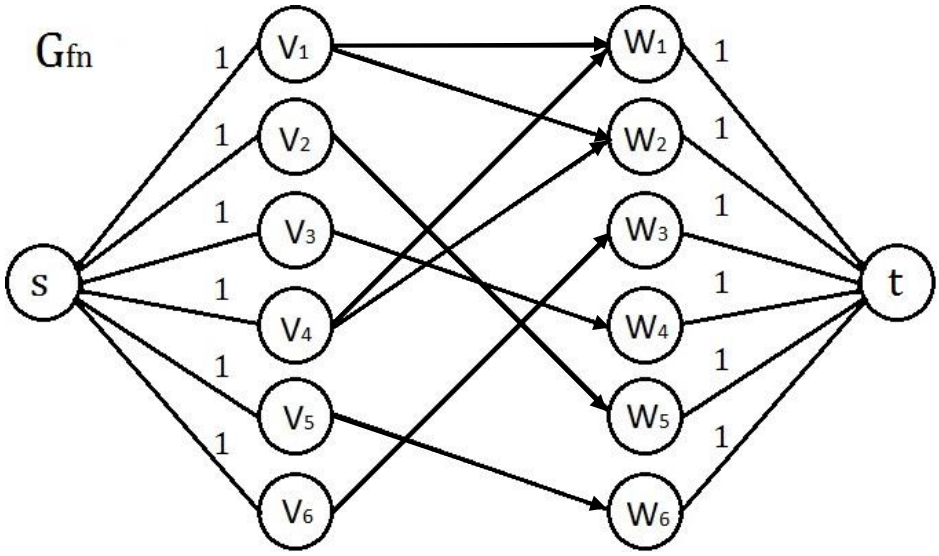


Figure 4.2.2

Now we run *Ford-Fulkerson* algorithm to find f_{max}.
If $|f_{max}| < |V|$ isomorphism is ruled out.

Otherwise if $|f_{max}| = |V|$ isomorphism is possible, but further checks are necessary.
To obtain the matching itself from f_{max}, we take only the edges:

$$E_M = \{(v, u) \mid v \in V \wedge u \in U \wedge f(v, u) = 1\}$$

Now that we have one match, we can use an algorithm for Finding All the Perfect Matchings in Bipartite Graphs by K. Fukuda and T. Matsui.

4.3 Finding All the Perfect Matchings

Now that we have a matching, let us denote a sub-problem $P(G_b, E_M)$.
We direct the edges in G_b as such:

$$\forall e \in E - E_M : V \longrightarrow U$$
$$\forall e \in E_M : V \longleftarrow U$$

Suppose $E_M = \{(v_1, w_1), (v_2, w_5), (v_3, w_4), (v_4, w_2), (v_5, w_6), (v_6, w_3)\}$, so the new directed bipartite graph we obtain, G_b', is:

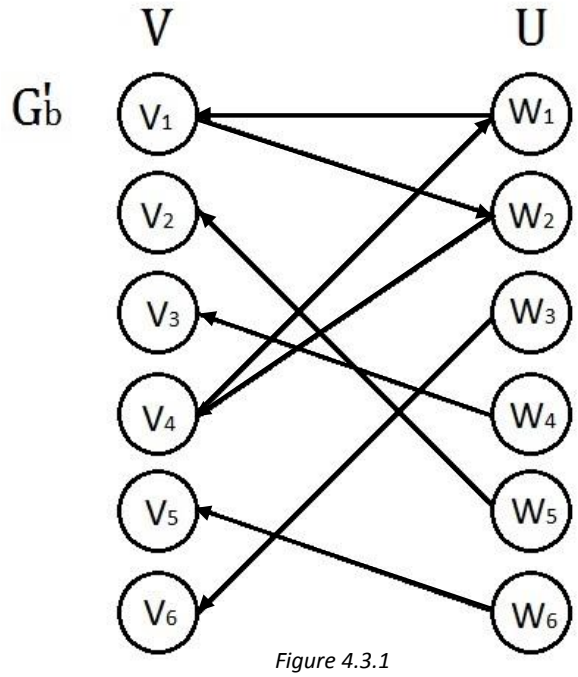


Figure 4.3.1

Now we check for elementary cycles in G_b'.
If there are none, then there are no more matches.
If we found a cycle, such as $C = \langle v_1 \to w_2 \to v_4 \to w_1 \to v_1 \rangle$, we remove the direction from its edges and denote them $E(C)$, in this case:

$$E(C) = \{(v_1, w_2), (w_2, v_4), (v_4, w_1), (w_1, v_1)\}$$

Now to obtain another match E_M', we simply take the symmetric difference:

$$E_M' = E(C) \,\Delta\, E_M$$
$$E_M' = \{(v_1, w_2), (v_2, w_5), (v_3, w_4), (v_4, w_1), (v_5, w_6), (v_6, w_3)\}$$

In this case, there are no more matches so we halt here.

But generally, once we have two distinct matches, we pick any edge $e = (i, j)$ such that $e \in E_M - E_M'$ and construct two new bipartite graphs as such:

$$G_{b1} = (V, U, E_1)$$
$$E_1 = E - \{(i, j)\}$$

$$G_{b2} = (V, U, E_2)$$
$$E_2 = \{(v,u) | v \in V - \{i\} \wedge u \in U - \{j\}\} \cup \{(i,j)\}$$

So in G_{b1}, e doesn't exist. In G_{b2} it's fixed, so it's the only edge from v, and the only edge to u.

Now we obtain two new sub-problems:

$$P'(G_{b2}, E_M)$$
$$P''(G_{b1}, E'_M)$$

We apply to them the same methods as we did to $P(G_b, E_M)$. Clearly, by repeating this process we create a binary tree of different sub-problems, each giving us either a completely new matching and two more sub-problems, effectively splitting that branch, or nothing, in which case we can think of that branch of the tree as ended.

Confirming Isomorphism

Now that we found all the matchings, we must check that they really are isomorphisms.

To do that, we simply compare each vertex from G_1 along with its neighbors against each vertex from G_2 and its neighbors.

So if, for example, v_1 was matched with w_2, it can only be a correct match if and only if each of v_1's neighbors was mapped to one of w_2's neighbors.

```
for(List<Vertex> l: mlst)
{
    boolean flag = true;
    for (Vertex v : vertices)
        for (Vertex w : vertices)
        {
            viso = l.get(v.name).name;
            wiso = l.get(w.name).name;

            if (v.neighbours.contains(w))
            {
                if (other.mat[viso][wiso] != 1)
                    flag = false;
            }

            else if (other.mat[viso][wiso] == 1)
                flag = false;
        }

    if(flag)
        finlst.add(l);
}
return finlst;
```

Figure 4.3.2

Once confirmed the isomorphism(s) can be printed to the screen and we're done.

Chapter 5: Problems faced and Solved

5.1 Finding squares

The code for finding triangles and squares in the graph was written very early in the projects development, we tested it and it appeared to work.

Later when testing for isomorphism on some high symmetry graphs with many squares, we realized that we have a problem. The method for counting triangles was working well, but the one for squares was miscounting.

It took us a long time and many tests to realize that the code wasn't bugged, it's just that the same idea we used for triangles couldn't be applied to squares. At least not as simply as we thought.

We realized that we need to test for rotated squares (e.g. $ABCD \xrightarrow{rotate} DABC$) as well as reversed squares (e.g. $ABCD \xrightarrow{reverse} DCBA$) if we don't want to count them many more times than necessary.

The final method came out a lot more complicated than we first expected:

```
int c=0, f;
List<List<Integer>> sq = new ArrayList<>();
for(Vertex v1: vertices)
    for(Vertex v2: v1.neighbours)
        for(Vertex v3: v2.neighbours)
            if(v3.name != v1.name)
                for(Vertex v4: v3.neighbours)
                    if(v4.name != v2.name && v4.name != v1.name)
                        if(v4.neighbours.contains(v1))
                        {
                            f = 0;
                            ArrayList<Integer> l = new ArrayList<Integer>();
                            ArrayList<Integer> lr = new ArrayList<Integer>();
                            l.add(v1.name); l.add(v2.name); l.add(v3.name); l.add(v4.name);
                            lr.add(v4.name); lr.add(v3.name); lr.add(v2.name); lr.add(v1.name);
                            for(int i=0; i<l.size(); i++)
                            {
                                if (sq.contains(l) || sq.contains(lr)) { f = 1; }
                                l = rotate(l);
                                lr = rotate(lr);
                            }
                            if(f == 0)
                            {
                                sq.add(l);
                                c++;
                                for(Vertex v: vertices)
                                    if(v.name == v1.name || v.name == v2.name || v.name == v3.name || v.name == v4.name)
                                        v.squares++;
                            }
                        }
```

Figure 5.1.1

5.2 Finding all matches

When we finished writing the code for *Aho, Hopcroft & Ullman*'s algorithm for rooted trees, we thought that the hardest part was behind us.

"now we just write a couple of loops and that's it" – or so we thought, at least.

When we actually got to the loops, we realized we had a big problem:
We had to run over each $v.PossibleCandidate$ list, $\forall v \in V$. Sounds simple enough, until you realize that a dynamic number of loops is required. Moreover, each loop runs a different number of times, which is also changing mid-run while the program is checking.

We tried to solve this with recursion, but it was too complicated.

We tried to look for a solution online, but to no avail.

We asked our supervisor, Dr. David Tankus, which after thinking for some time did come up with a solution, but it was very complicated and we weren't 100% sure on how to implement it as well.

Only after a long time of thinking on it, we realized that this could be translated to another problem entirely. We realized that this is exactly a "find the perfect match" problem, that could be solved by constructing a flow network on top of a bipartite graph.

Later we realized that this is actually "find all the perfect matches", plural, problem.
We looked online and thankfully we found a paper by two Japanese mathematicians describing exactly the algorithm we needed. We jumped in on it and immediately started trying to figure it out.
It wasn't simple, took us a few days to fully understand it, and even then our work was far from over.

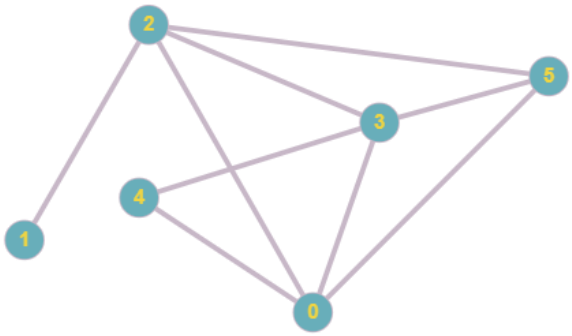
In order to implement their algorithm, we had to introduce a lot of new code to our project, around 40% more.

We ended up adding 4 new classes, conversion between directed and undirected graphs, flow, and more, which was not what we expected at all… and all of this because we couldn't "just write a couple of loops and that's it".

Chapter 6: Appendix

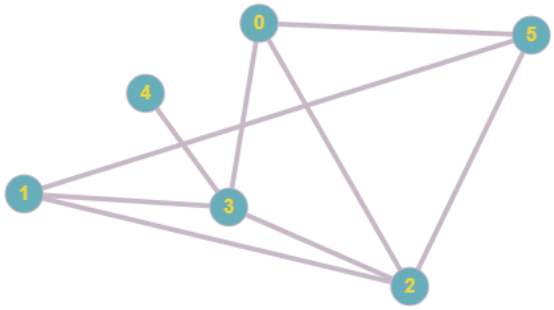
In this section, we show examples of tests we ran.
In each example we have an illustration of the graphs created on *http://www.graphonline.ru/en/* as well as the output from our program. Note that the vertex names in the outputted mappings are not related to the vertex names in the illustrations!

Example 6.1 – low symmetry



Graph info:
ag1.txt

0 0 1 0 1 1
0 0 1 0 0 0
1 1 0 1 0 1
0 0 1 0 1 1
1 0 0 1 0 0
1 0 1 1 0 0



Number of vertices: 6
Number of edges: 8
Triangles in graph: 2
Squares in graph: 3
Symmetric: true
Anti-reflexive: true
Tree: false

Graph info:
ag2.txt

0 0 1 1 0 1
0 0 1 1 0 1
1 1 0 1 0 0
1 1 1 0 1 0
0 0 0 1 0 0
1 1 0 0 0 0

Number of vertices: 6
Number of edges: 8
Triangles in graph: 2
Squares in graph: 3
Symmetric: true
Anti-reflexive: true
Tree: false

The Graphs are Isomorphic!
Our program only checked 2 combinations instead of 720
That's roughly 360.0 times faster!
The Isomorphisms are:
1
V1 --> V1
V2 --> V5
V3 --> V4
V4 --> V2
V5 --> V6
V6 --> V3

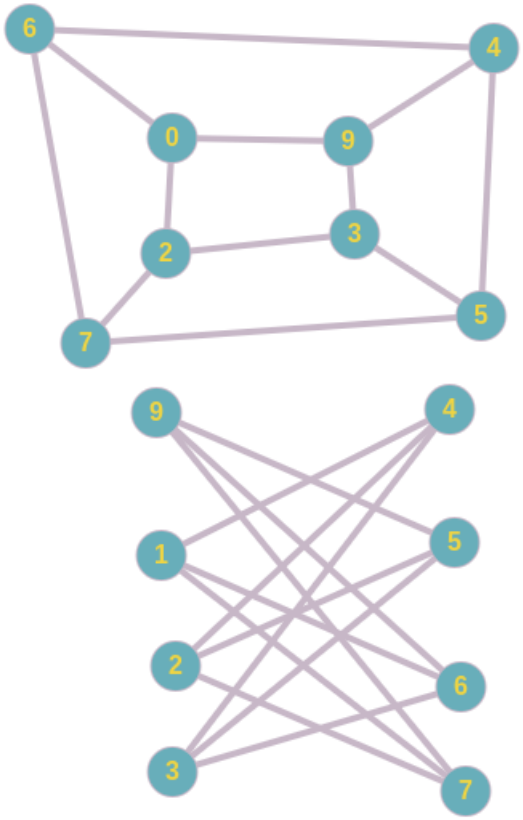
2
V1 --> V2
V2 --> V5
V3 --> V4
V4 --> V1
V5 --> V6
V6 --> V3

Example 6.2 – high symmetry
In this example, due to highly symmetrical graphs, our program failed to reduce the amount of checks.

Graph info:
ag11.txt

0 1 0 0 0 1 0 1
1 0 1 0 0 0 1 0
0 1 0 0 1 0 0 1
0 0 0 0 1 1 0 1
0 0 1 1 0 0 1 0
1 0 0 1 0 0 1 0
0 1 0 0 1 1 0 0
1 0 1 1 0 0 0 0

Number of vertices: 8
Number of edges: 12
Triangles in graph: 0
Squares in graph: 6
Symmetric: true
Anti-reflexive: true
Tree: false



Graph info:
ag22.txt

0 0 0 1 0 1 1 0
0 0 0 1 1 0 1 0
0 0 0 1 1 1 0 0
1 1 1 0 0 0 0 0
0 1 1 0 0 0 0 1
1 0 1 0 0 0 0 1
1 1 0 0 0 0 0 1
0 0 0 0 1 1 1 0

Number of vertices: 8
Number of edges: 12
Triangles in graph: 0
Squares in graph: 6
Symmetric: true
Anti-reflexive: true
Tree: false

The Graphs are Isomorphic!
Our program only checked 40320 combinations instead of 40320
That's roughly 1 times faster
The Isomorphisms are:
1
V1 --> V8
V2 --> V6
V3 --> V3
V4 --> V2
V5 --> V4
V6 --> V7
V7 --> V1
V8 --> V5

2
V1 --> V8
V2 --> V6
V3 --> V1
V4 --> V2
V5 --> V4
V6 --> V5
V7 --> V3
V8 --> V7
.
.
.

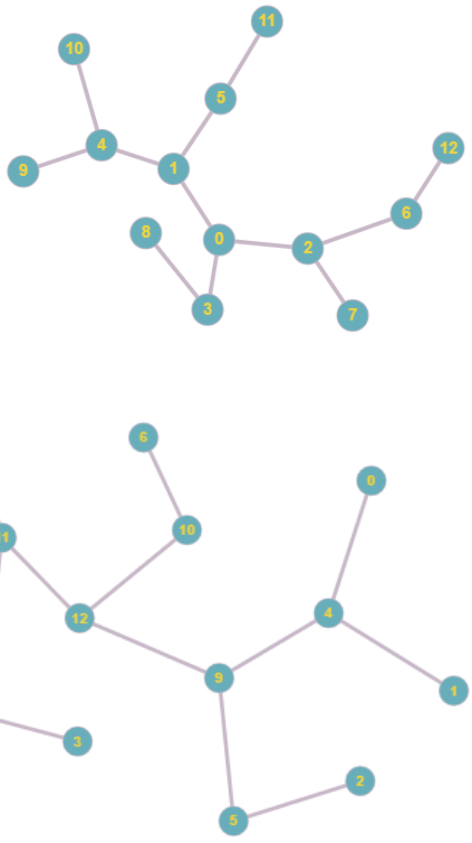
.
.
47
V1 --> V1
V2 --> V4
V3 --> V2
V4 --> V8
V5 --> V5
V6 --> V6
V7 --> V3
V8 --> V7

48
V1 --> V1
V2 --> V4
V3 --> V3
V4 --> V8
V5 --> V5
V6 --> V7
V7 --> V2
V8 --> V6

Example 6.3 – Trees

Tree info:
at.txt
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 1 1 1 0

Number of vertices: 13
Number of edges: 12
Tree Height: 3



Symmetric: true
Anti-reflexive: true
Tree: true

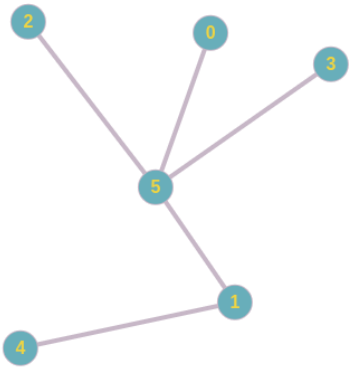
Tree info:
bt.txt
0 1 1 1 0 0 0 0 0 0 0 0 0
1 0 0 0 1 1 0 0 0 0 0 0 0
1 0 0 0 0 0 1 1 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 1 1 0 0
0 1 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0

Number of vertices: 13
Number of edges: 12
Tree Height: 3
Symmetric: true
Anti-reflexive: true
Tree: true

The Trees are Isomorphic!
The Isomorphism is:
V13 --> V1
V11 --> V4
V7 --> V9
V12 --> V3
V9 --> V8
V8 --> V7
V4 --> V13
V10 --> V2
V6 --> V6
V3 --> V12
V5 --> V5
V1 --> V10
V2 --> V11

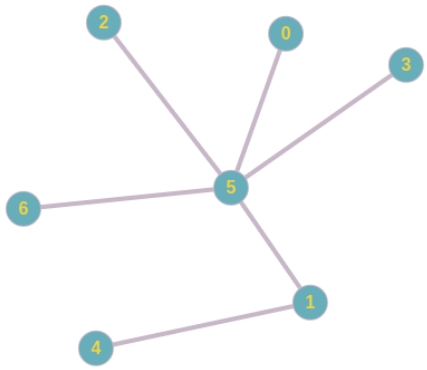
Example 6.4 – different trees

Tree info:
atq1.txt
0 0 0 0 0 1 0
0 0 0 0 1 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 1 0 0 0 0 0
1 1 1 1 0 0 1
0 0 0 0 0 1 0



Number of vertices: 7
Number of edges: 6
Tree Height: 2
Symmetric: true
Anti-reflexive: true
Tree: true

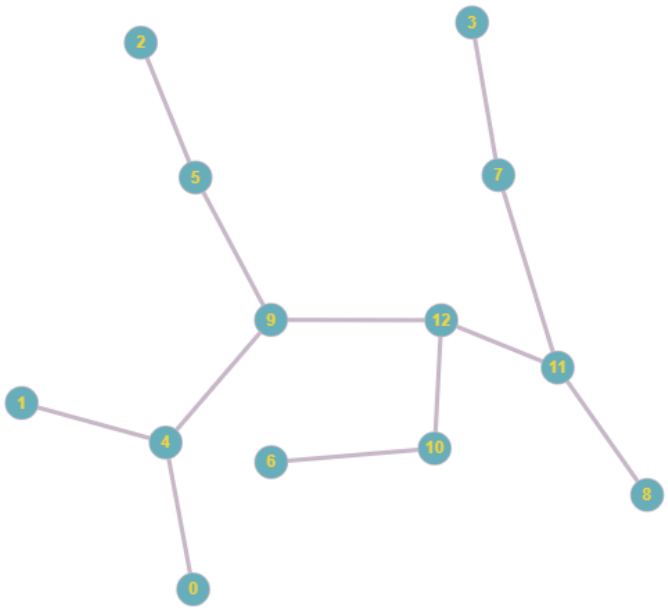
Tree info:
atq.txt
0 0 0 0 0 1
0 0 0 0 1 1
0 0 0 0 0 1
0 0 0 0 0 1
0 1 0 0 0 0
1 1 1 1 0 0



Number of vertices: 6
Number of edges: 5
Tree Height: 2
Symmetric: true
Anti-reflexive: true
Tree: true

The Trees are not Isomorphic!

Example 6.5 – same tree



Tree info:
at.txt
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 1 1 1 0

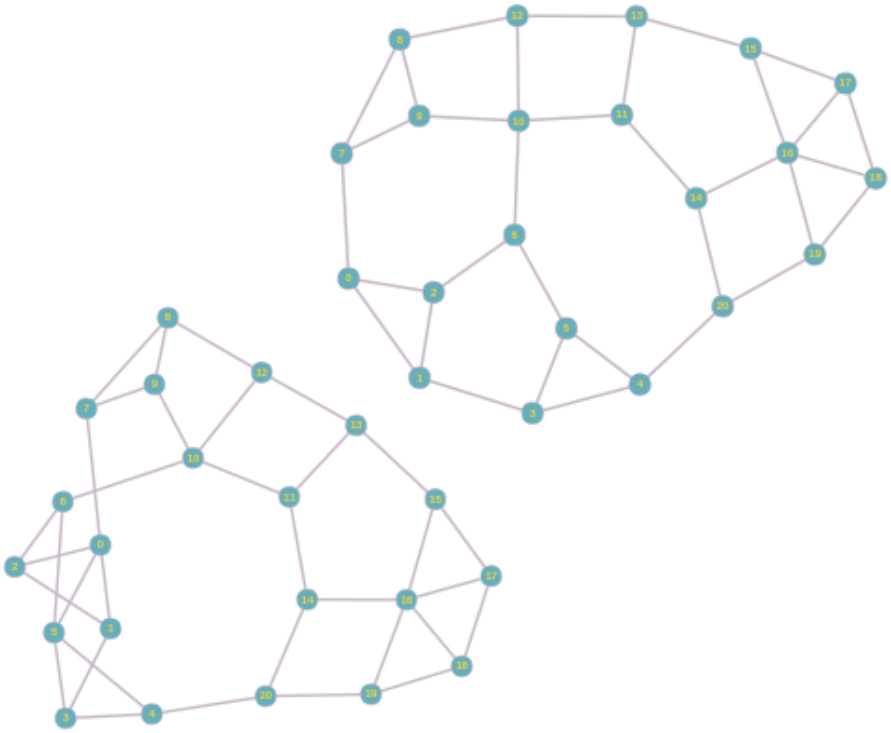
Number of vertices: 13
Number of edges: 12
Tree Height: 3
Symmetric: true
Anti-reflexive: true
Tree: true

Tree info:
at.txt
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 1 1 1 0

Number of vertices: 13
Number of edges: 12
Tree Height: 3
Symmetric: true
Anti-reflexive: true
Tree: true

The Trees are Isomorphic!
The Isomorphism is:
V13 --> V13
V11 --> V11
V7 --> V7
V12 --> V12
V9 --> V9
V8 --> V8
V4 --> V4
V10 --> V10
V6 --> V6
V3 --> V3
V5 --> V5
V1 --> V1
V2 --> V2

<u>Example 6.6 – different graphs</u>
Graph info:
ag1111.txt
0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0
Number of vertices: 21
Number of edges: 34
Triangles in graph: 6
Squares in graph: 7
Symmetric: true
Anti-reflexive: true
Tree: false



Graph info:

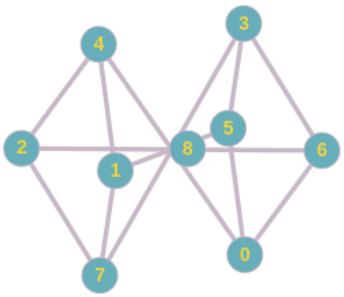
ag111.txt

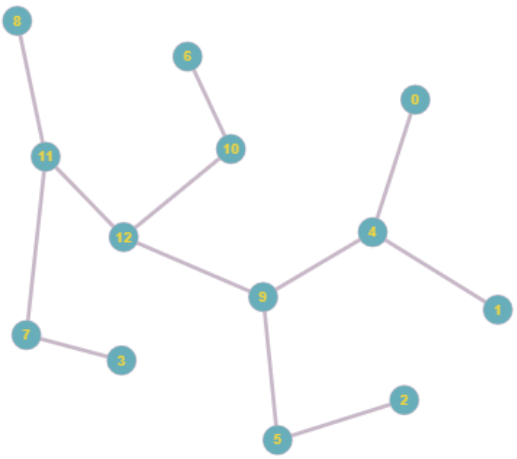
0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0

Number of vertices: 21
Number of edges: 33
Triangles in graph: 6
Squares in graph: 5
Symmetric: true
Anti-reflexive: true
Tree: false

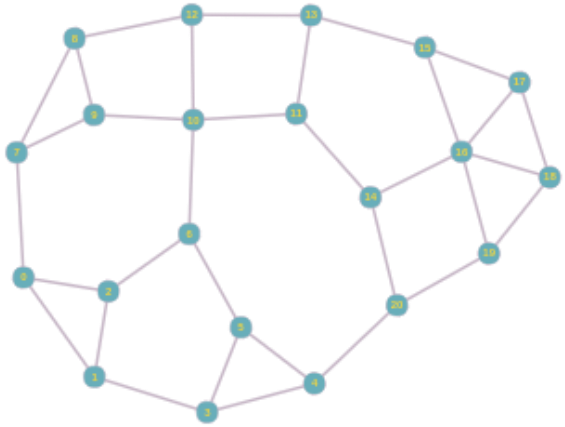
The Graphs are not Isomorphic!

Example 6.7 – tree and graph
Graph in File: ag.txt Is not a Tree





Example 6.8 – big graph
Graph info:
ag1111.txt



0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0

Number of vertices: 21
Number of edges: 34
Triangles in graph: 6
Squares in graph: 7
Symmetric: true

Anti-reflexive: true
Tree: false

The Graphs are Isomorphic!
Our program only checked 1 combinations instead of 51090942171709440000
That's roughly 51090942171709440000 times faster
The Isomorphism is:
V1 --> V1
V2 --> V2
V3 --> V3
V4 --> V4
V5 --> V5
V6 --> V6
V7 --> V7
V8 --> V8
V9 --> V9
V10 --> V10
V11 --> V11
V12 --> V12
V13 --> V13
V14 --> V14
V15 --> V15
V16 --> V16
V17 --> V17
V18 --> V18
V19 --> V19
V20 --> V20
V21 --> V21

References

V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

K. Fukuda and T. Matsui, “Finding All the Perfect Matthings in Bipartite Graphs,” Appl. Math. Lett. 7 1 (1994) 15–18

http://wwwmayr.in.tum.de/konferenzen/Jass08/courses/1/smal/Smal_Paper.pdf

http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/HW5+.pdf