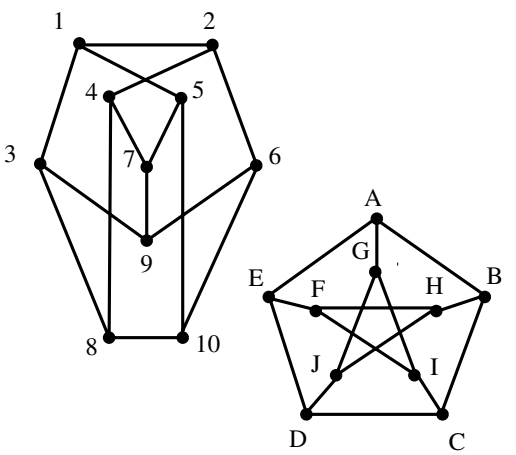


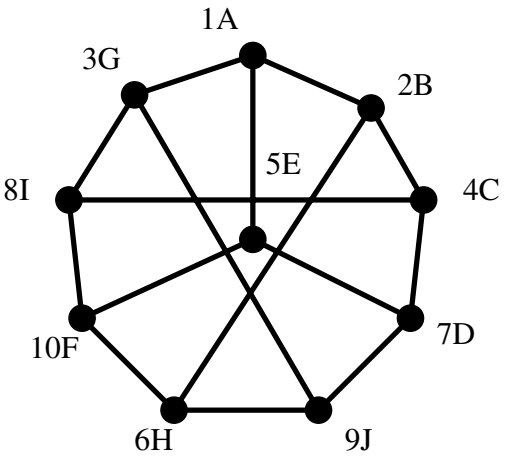


INPUT OUTPUT

## 16.9 Graph Isomorphism

**Input description**: Two graphs, $G$ and $H$.

**Problem description**: Find a (or all) mapping $f$ from the vertices of $G$ to the vertices of $H$ such that $G$ and $H$ are identical; i.e. , $(x, y)$ is an edge of $G$ iff $(f(x), f(y))$ is an edge of $H$.

**Discussion**: Isomorphism is the problem of testing whether two graphs are really the same. Suppose we are given a collection of graphs and must perform some operation on each of them. If we can identify which of the graphs are duplicates, we can discard copies to avoid redundant work.

Certain pattern recognition problems are readily mapped to graph or subgraph isomorphism detection. The structure of chemical compounds are naturally described by labeled graphs, with each atom represented by a vertex. Identifying all molecules in a structure database containing a particular functional group is an instance of subgraph isomorphism testing.

We need some terminology to settle what is meant when we say two graphs are the same. Two labeled graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$ are *identical* when $(x, y) \in E_g$ iff $(x, y) \in E_h$. The isomorphism problem consists of finding a mapping from the vertices of $G$ to $H$ such that they are identical. Such a mapping is called an *isomorphism*; the problem of finding the mapping is sometimes called *graph matching*.

Identifying symmetries is another important application of graph isomorphism. A mapping of a graph to itself is called an *automorphism*, and the collection of

automorphisms (the automorphism *group*) provides a great deal of information about symmetries in the graph. For example, the complete graph $K_n$ has $n!$ automorphisms (any mapping will do), while an arbitrary random graph is likely to have few or perhaps only one, since $G$ is always identical to itself.

Several variations of graph isomorphism arise in practice:

- *Is graph $G$ contained in graph $H$?* – Instead of testing equality, we are often interested in knowing whether a small pattern graph $G$ is a *subgraph* of $H$. Such problems as clique, independent set, and Hamiltonian cycle are important special cases of subgraph isomorphism.

  There are two distinct graph-theoretic notions of "contained in." *Subgraph isomorphism* asks whether there is a subset of edges and vertices of $H$ that is isomorphic to a smaller graph $G$. *Induced subgraph isomorphism* asks whether there is a subset of vertices of $H$ whose deletion leaves a subgraph isomorphic to a smaller graph $G$. For induced subgraph isomorphism, (1) all edges of $G$ must be present in $H$, and (2) no *non-edges* of $G$ can be present in $H$. Clique happens to be an instance of both subgraph isomorphism problems, while Hamiltonian cycle is only an example of vanilla subgraph isomorphism.

  Be aware of this distinction in your application. Subgraph isomorphism problems tend to be harder than graph isomorphism, while induced subgraph problems tend to be even harder than subgraph isomorphism. Some flavor of backtracking is your only viable approach.

- *Are your graphs labeled or unlabeled?* – In many applications, vertices or edges of the graphs are *labeled* with some attribute that must be respected in determining isomorphisms. For example, in comparing two bipartite graphs, each with "worker" vertices and "job" vertices, any isomorphism that equated a job with a worker would make no sense.

  Labels and related constraints can be factored into any backtracking algorithm. Further, such constraints significantly speed up the search, by creating many more opportunities for pruning whenever two vertex labels do not match up.

- *Are you testing whether two trees are isomorphic?* – Faster algorithms exist for certain special cases of graph isomorphism, such as trees and planar graphs. Perhaps the most important case is detecting isomorphisms among trees, a problem that arises in language pattern matching and parsing applications. A parse tree is often used to describe the structure of a text; two parse trees will be isomorphic if the underlying pair of texts have the same structure.

  Efficient algorithms for tree isomorphism begin with the leaves of both trees and work inward toward the center. Each vertex in one tree is assigned a label representing the set of vertices in the second tree that might possibly

be isomorphic to it, based on the constraints of labels and vertex degrees. For example, all the leaves in tree $T_1$ are initially potentially equivalent to all leaves of $T_2$. Now, working inward, we can partition the vertices adjacent to leaves in $T_1$ into classes based on how many leaves and non-leaves they are adjacent to. By carefully keeping track of the labels of the subtrees, we can make sure that we have the same distribution of labeled subtrees for $T_1$ and $T_2$. Any mismatch means $T_1 \neq T_2$, while completing the process partitions the vertices into equivalence classes defining all isomorphisms. See the references below for more details.

- *How many graphs do you have?* – Many data mining applications involve searching for all instances of a particular pattern graph in a big database of graphs. The chemical structure mapping application described above falls into this family. Such databases typically contain a large number of relatively small graphs. This puts an onus on indexing the graph database by small substructures (say five to ten vertex each), and doing expensive isomorphism tests only against those containing the same substructures as the query graph.

No polynomial-time algorithm is known for graph isomorphism, but neither is it known to be NP-complete. Along with integer factorization (see Section 13.8 (page 420)), it is one of the few important algorithmic problems whose rough computational complexity is still not known. The conventional wisdom is that isomorphism is a problem that lies between P and NP-complete if P $\neq$ NP.

Although no worst-case polynomial-time algorithm is known, testing isomorphism is *usually* not very hard in practice. The basic algorithm backtracks through all $n!$ possible relabelings of the vertices of graph $h$ with the names of vertices of graph $g$, and then tests whether the graphs are identical. Of course, we can prune the search of all permutations with a given prefix as soon as we detect any mismatch between edges whose vertices are both in the prefix.

However, the real key to efficient isomorphism testing is to preprocess the vertices into "equivalence classes," partitioning them into sets of vertices so that two vertices in different sets cannot possibly be mistaken for each other. All vertices in each equivalence class must share the same value of some invariant that is independent of labeling. Possibilities include:

- *Vertex degree* – This simplest way to partition vertices is based on their degree—the number of edges incident on the vertex. Two vertices of different degrees cannot be identical. This simple partition can be a big win, but won't do much for regular (equal degree) graphs.

- *Shortest path matrix* – For each vertex $v$, the all-pairs shortest path matrix (see Section 15.4 (page 489)) defines a multiset of $n-1$ distances representing the distances between $v$ and each of the other vertices. Any two identical vertices must define the exact same multiset of distances, so we can partition the vertices into equivalence classes defining identical distance multisets.

- *Counting length-k paths* – Taking the adjacency matrix of $G$ and raising it to the $k$th power gives a matrix where $G^k[i,j]$ counts the number of (nonsimple) paths from $i$ to $j$. For each vertex and each $k$, this matrix defines a multiset of path-counts, which can be used for partitioning as with distances above. You could try all $1 \leq k \leq n$ or beyond, and use any single deviation as an excuse to partition.

Using these invariants, you should be able to partition the vertices of most graphs into a large number of small equivalence classes. Finishing the job off with backtracking should then be short work. We assign each vertex the name of its equivalence class as a label, and treat it as a labeled matching problem. It is harder to detect isomorphisms between highly-symmetric graphs than it is with random graphs because of the reduced effectiveness of these equivalence-class partitioning heuristics.

**Implementations**: The best known isomorphism testing program is `nauty` (No AUTomorphisms, Yes?)—a set of very efficient C language procedures for determining the automorphism group of a vertex-colored graph. Nauty is also able to produce a canonically-labeled isomorph of the graph, to assist in isomorphism testing. It was the basis of the first program to generate all 11-vertex graphs without isomorphs, and can test most graphs with fewer than 100 vertices in well under a second. Nauty has been ported to a variety of operating systems and C compilers. It is available at *http://cs.anu.edu.au/∼bdm/nauty/*. The theory behind `nauty` is described in [McK81].

The `VFLib` graph-matching library contains implementations for several different algorithms for *both* graph and subgraph isomorphism testing. This library has been widely used and very carefully benchmarked [FSV01]. It is available at *http://amalfi.dis.unina.it/graph/*.

`GraphGrep` [GS02] (*http://www.cs.nyu.edu/shasha/papers/graphgrep/*) is a representative data mining tool for querying large databases of graphs.

Valiente [Val02] has made available the implementations of graph/subgraph isomorphism algorithms for both trees and graphs in his book [Val02]. These C++ implementations run on top of LEDA (see Section 19.1.1 (page 658)), and are available at *http://www.lsi.upc.edu/∼valiente/algorithm/*.

Kreher and Stinson [KS99] compute isomorphisms of graphs in addition to more general group-theoretic operations. These implementations in C are available at *http://www.math.mtu.edu/∼kreher/cages/Src.html*.

**Notes**: Graph isomorphism is an important problem in complexity theory. Monographs on isomorphism detection include [Hof82, KST93]. Valiente [Val02] focuses on algorithms for tree and subgraph isomorphism. Kreher and Stinson [KS99] take a more group-theoretic approach to isomorphism testing. Graph mining systems and algorithms are surveyed in [CH06]. See [FSV01] for performance comparisons between different graph and subgraph isomorphism algorithms.

Polynomial-time algorithms are known for planar graph isomorphism [HW74] and for graphs where the maximum vertex degree is bounded by a constant [Luk80]. The all-pairs

shortest path heuristic is due to [SD76], although there exist nonisomorphic graphs that realize the exact same set of distances [BH90]. A linear-time tree isomorphism algorithm for both labeled and unlabeled trees is presented in [AHU74].

A problem is said to be *isomorphism-complete* if it is provably as hard as isomorphism. Testing the isomorphism of bipartite graphs is isomorphism-complete, since any graph can be made bipartite by replacing each edge by two edges connected with a new vertex. Clearly, the original graphs are isomorphic if and only if the transformed graphs are.

**Related Problems**: Shortest path (see page 489), string matching (see page 628).