12 Network Comparison

Michael Baur and Marc Benkert

A fundamental question in comparative network analysis is whether two given networks have the same structure. To formalize what to relate to structural equivalence, the following definition was made:

Definition 12.0.1. *Two undirected simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are* isomorphic *(denoted by $G_1 \simeq G_2$) if there is an edge-preserving bijective vertex mapping $\phi : V_1 \longrightarrow V_2$, i.e. a bijection ϕ with*

$$\forall u, v \in V_1 : \ \{u, v\} \in E_1 \Longleftrightarrow \{\phi(u), \phi(v)\} \in E_2.$$

The graph isomorphism problem (GI) is to determine whether two given graphs are isomorphic. Figure 12.1 shows an example of two – differently embedded – isomorphic graphs. However, in practice it will be extremely rare that two graphs are isomorphic. We can deal with this fact, as in most cases it is comparatively easy to recognize two graphs as non-isomorphic. We simply have to check necessary conditions: trivially, the number of vertices and edges has to match. For each degree value the number of vertices having this degree has to match, the two graphs must form the same number of connected components, the diameter has to match, and so on. We can also use more complicated properties like those from other chapters in this book: if it should be possible that two graphs are isomorphic, their spectra should be equal, all centrality indices have to match, etc. One could give an ever increasing list of necessary conditions, but thus far no one has succeeded in giving a sufficient condition that is polynomially computable. More details are given in Section 12.1, especially in the overview and in Section 12.1.3.

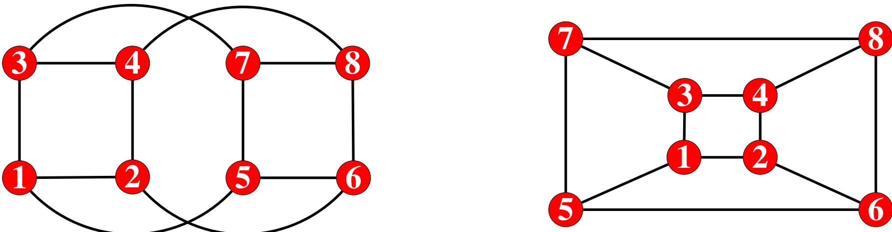


Fig. 12.1. Two isomorphic graphs. The labeling indicates a possible isomorphism; it is not part of the graph

U. Brandes and T. Erlebach (Eds.): Network Analysis, LNCS 3418, pp. 318–340, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Indeed, even in the case of two graphs being non-isomorphic, we want to make a statement as to how similar the graphs are. For example, in chemistry it is often desired to determine the similarity between two molecular structures. Several approaches were made to give such similarity measures; we will present the important ones in Section 12.2. One can also ask if one graph is a part of another; this leads to the *Subgraph Isomorphism Problem*: Determine for two given graphs H and G whether there is a subgraph $H' \subset G$ with $H \simeq H'$. This problem is $\mathcal{NP}$–complete [240] and can probably only be solved in time exponential in the number of vertices of the subgraph.

12.1 Graph Isomorphism

Although the graph isomorphism problem has been studied since the Seventies [489], its complexity status is still unknown. Clearly GI$\in \mathcal{NP}$, but it is not known that GI is polynomially solvable or that it is $\mathcal{NP}$-complete. Also the relationship of GI to co-$\mathcal{NP}$ is not known. Unless $\mathcal{P} = \mathcal{NP}$ there are problems whose complexity status is intermediate, that means their complexity class lies between $\mathcal{P}$ and $\mathcal{NPC}$. The widespread conjecture is that GI is such an intermediate problem. Indications for that conjecture are on one hand that, in spite of enduring research, no polynomial algorithm has been found. And on the other hand, it is known that the counting version of the problem (determine the number of all isomorphisms) is equally difficult as the decision version itself [409]. This together with a theorem proven by Boppana, Håstad and Zachos [80] and Schöning [504] indicates that GI is unlikely to be $\mathcal{NP}$-complete. The theorem makes a statement on the collapse of the polynomial time hierarchy as a consequence of GI$\in \mathcal{NPC}$, which is considered to be very unlikely. One approach in complexity theory to follow this conjecture is to define a special complexity class *isomorphism-complete* which contains GI and all problems as hard as GI. However, Lubiw described in 1981 $\mathcal{NP}$-complete problems similar to GI [398].

Nevertheless, GI is in $\mathcal{P}$ for many graph classes, and graph classes for which GI is really difficult seem to be rare. GI is in $\mathcal{P}$ for trees [4], planar graphs [313], [314], graphs with bounded degree [402], circular-arc graphs [316] and interval graphs (as a subclass of circular-arc graphs). Recently, Cvetkovic, Rowlinson and Simic [136] showed that GI is in $\mathcal{P}$ for graphs with eigenvalues of bounded multiplicity. On the other hand, isomorphism-completeness is maintained on bipartite graphs, line graphs [4], chordal graphs [77] and regular graphs [76]. Most of the positive results are mainly of theoretical interest as the introduced algorithms are of little practical use.

A powerful approach to solve GI is to consider the automorphism group $\mathrm{Aut}(G_1)$ of a given graph G_1, or at least the computable information about $\mathrm{Aut}(G_1)$. Clearly, if $\mathrm{Aut}(G_1)$ is known, $G_1 \simeq G_2$ can be decided by testing $\phi(G_1) = G_2$ for all $\phi \in \mathrm{Aut}(G_1)$. Even if we cannot compute $\mathrm{Aut}(G_1)$ explicitely, we can restrict the number of possible isomorphisms between two graphs by grouping their vertices in equivalence classes. For this, vertex invariants are used. A vertex invariant is a function *inv* defined on the vertex set of a graph

with the following property: if there is an isomorphism between G_1 and G_2 that maps v to w then $inv(v) = inv(w)$. The simplest vertex invariant, and in many cases the most powerful, is the degree of a vertex. We can immediately recognize two graphs to be non-isomorphic if their degree sequences are different. If the sequences are equal, but the cardinality of equivalence classes is small, the number of possible isomorphisms is restricted, e.g. if in each graph there are only three vertices of degree d and all other degrees appear only once, the number of possible isomorphisms is 6. In general, we can solve GI polynomially on graph classes for which the automorphism group is polynomially computable, or at least the vertices can be grouped in equivalence classes such that the number of possible isomorphisms between two graphs is polynomial. This raises the question for which graph classes this approach does not work, as these might be difficult to solve. The degree sequence, for example, yields no restriction for regular graphs, but also more elaborate properties may fail. In Section 12.1.3 we give two comparatively small examples that show the hardness of solving GI polynomially using invariants. It gets difficult if the graph does not allow a meaningful vertex grouping because the graph structure is very regular. For more details see Section 4 of [221].

To solve the problem (on general graphs) in practice, there are mainly two methods. Naturally, the direct one: take the two graphs that are to be compared and try to compute an isomorphism. This has the advantage that if there are many isomorphisms, only one has to be found. The second method is to define – independently from the comparison of two specific graphs – a canonical label C, which is a function on the set of all graphs, such that G_1 and G_2 are isomorphic if and only if $C(G_1) = C(G_2)$. This has the advantage that already computed information can be recycled for new comparisons. *McKay's nauty algorithm* grabs this second idea and has become the most practical algorithm for GI. We will elaborate on it later in Section 12.1.2, but refer to [415] for full details. However, first we will have a look at a simple backtracking algorithm that follows the first method.

12.1.1 A Simple Backtracking Algorithm

For the first method we give an algorithm that uses vertex invariants in order to find an isomorphism. The more powerful the invariant, the less the number of functions tested to be isomorphisms from the $n!$ possible ones. Let $\mathcal{R}$ be a set with a linear order '<'. Let $inv : V \to \mathcal{R}$ denote some vertex invariant, e.g. $inv(v) = d(v)$ and $\mathcal{R} = \mathbb{N}$. Let $\Pi(V, inv) = \big(V_1, \ldots, V_k\big)$ be the ordered vertex partition of V with respect to inv, i.e. $\forall v, w \in V_i : inv(v) = inv(w)$ and for all $v \in V_i, w \in V_j$ with $i < j : inv(v) < inv(w)$.

Let $G_1 = \big(V = \{v_1, \ldots, v_n\}, E_1\big)$ and $G_2 = \big(W = \{w_1, \ldots, w_n\}, E_2\big)$ denote the two graphs that are checked for isomorphism. The output of the algorithm will be a permutation ϕ of $\{1, \ldots, n\}$, such that $v_i \to w_{\phi(i)},\ 1 \leq i \leq n$, is an isomorphism between G_1 and G_2, or 'NON-isomorphic', if no isomorphism exists. The algorithm will extend isomorphisms between subgraphs of G_1 and G_2 step-by-step and either stop if an isomorphism can be extended to the whole graphs or

Algorithm 26: ISOMORPH$(G_1, G_2, (V_1, \ldots, V_m), (W_1, \ldots, W_m), \phi')$

Input: Graphs $G_1 = \big(V = \{v_1, \ldots, v_n\}, E_1\big)$, $G_2 = \big(W = \{w_1, \ldots, w_n\}, E_2\big)$, vertex partitions $(V_1, \ldots, V_m), (W_1, \ldots, W_m)$ with $\bigcup V_i \subset V$, $\bigcup W_i \subset W$ and $|V_i| = |W_i|$, and an isomorphism ϕ' between the subgraphs induced by $V \setminus \bigcup V_i$ and $W \setminus \bigcup W_i$.

Output: ϕ, if ϕ' is extensible to an isomorphism ϕ between G_1 and G_2, 'NON-isomorphic' otherwise.

```
if (V_1, ..., V_m) = ∅ then return φ'
compute V_i ∈ { V_j | |V_j| ≤ |V_ℓ|, 1 ≤ ℓ ≤ m }
let V_i = {v_i1, v_i2, ...}, W_i = {w_i1, w_i2, ...}
for j = 1, ..., |V_i| do
    if φ' extended by i1 → ij is an isomorphism between the subgraphs induced
    by V \ ⋃V_k ∪ {v_i1} and W \ ⋃W_k ∪ {w_ij} then
        branch = ISOMORPH(G_1, G_2, (V_1, ..., V_i \ v_i1, ... V_m),
                          (W_1, ..., W_i \ w_ij, ... W_m), φ' ∪ {i1 → ij})
        if branch ≠ 'NON-isomorphic' then return branch
return 'NON-isomorphic'
```

if all possibilities have been checked unsuccessfully. Isomorphisms on subgraphs will be denoted by ϕ'. Note that any ϕ' is a bijection between two subsets of $\{1, \ldots, n\}$. Initially $\Pi(V, inv) = (V_1, \ldots, V_k)$ and $\Pi(W, inv) = (W_1, \ldots, W_{k'})$ are computed. If $k \neq k'$ or $|V_i| \neq |W_i|$ for any $1 \leq i \leq k$, the two graphs cannot be isomorphic because each possible mapping does not preserve *inv*. Let us assume that we have checked $k = k'$ and $|V_i| = |W_i|$ successfully in the preprocessing; then ISOMORPH$\big(G_1, G_2, (\Pi(V, inv), \Pi(W, inv), \emptyset\big)$ is called, see Algorithm 26.

First, the vertex subset V_i with minimum cardinality among all subsets of the partition is determined; obviously W_i has the same cardinality. Any isomorphism ϕ between G_1 and G_2 has to map the vertices of V_i to the vertices of W_i. Thus it is sufficient to fix a mapping between a vertex of V_i and W_i and to go on. The smallest cell is chosen in the hope of detecting 'NON-isomorphism' as fast as possible. Now, in the *for*–loop we determine the mapping. If there is an isomorphism ϕ, then $\phi(v_{i1}) \in W_i$, and checking all mappings $v_{i1} \to w_{ij}$ is sufficient in order to obtain an isomorphism. If it is now still possible to extend $\phi' \cup \{v_{i1} \to w_{ij}\}$ to an isomorphism ϕ, we check the mappings of the remaining unmapped vertices. This is done by a recursive call of ISOMORPH.

12.1.2 McKay's Nauty Algorithm

An example of the approach to compute a canonical label that has been implemented is *McKay's nauty algorithm.* In which *nauty* stands for NO AUTomorphisms YET?

We first explain McKay's idea to define a canonical label. For an undirected graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$ let $\text{Adj}(G, \delta)$ be the adjacency matrix of G with respect to the vertex order $v_{\delta(1)}, v_{\delta(2)}, \ldots, v_{\delta(n)}$, where δ is a

permutation of $\{1, \dots, n\}$. Then C_{adj} defined by

$$C_{\text{adj}}(G) = \min_{\delta \in S_n} \text{Adj}(G, \delta)$$

is a canonical label, where $\text{Adj}(G, \delta)$ is interpreted as a n^2-bit binary number derived by concatenation of all rows. Two labels $C_{\text{adj}}(G_1)$and $C_{\text{adj}}(G_2)$ are equal if and only if G_1 and G_2 are isomorphic. This is because the minimum adjacency matrix is uniquely defined, and two graphs are isomorphic only if there are vertex orders that yield equal adjacency matrices. The naive approach to compute $C_{\text{adj}}(G)$ would look at all $n!$ vertex orders and for each order compare two adjacency matrices of size $n \times n$. However, even for comparatively small values of n this would not be feasible in acceptable time. To speed up this approach McKay uses various techniques in his nauty algorithm in order to compute a label $C(G)$. In general, $C(G)$ will be different from $C_{\text{adj}}(G)$ as the nauty algorithm does not look at all $n!$ orders but at a special sample and computes the minimum matrix among them. The *Refinement Procedure* will determine these samples. The number of samples depends on the structure of the graph, but usually the sample size is significantly smaller than $n!$. To compute all vertex orders that will be checked, the nauty algorithm uses a search tree $\mathcal{T}$ in which each leaf corresponds to a vertex order. The algorithm traverses $\mathcal{T}$ and examines all adjacency matrices that are induced by the vertex orders of visited leaves. Now, the next trick comes into play: not all leaves are visited. Group theory, more precisely the information about the automorphism group $\text{Aut}(G)$ already known, allows to exclude subtrees of $\mathcal{T}$ from the traversal. A subtree is pruned if it is known that it contains only vertex orders that lead to adjacency matrices not smaller than the best one found so far. Using algebra, mainly group theory, it is shown that the label C derived by this approach is indeed canonical, see Theorem 12.1.2. There is another technique to prune $\mathcal{T}$, but, as it is very abstract, we will mention it only briefly in the chapter notes (Section 12.3). Next, we introduce some basics from group theory that we need in the sequel.

Basics in Group Theory

We denote the permutation group of n elements by S_n. An element $\delta \in S_n$ is simply a bijection between the sets $\{1, \dots, n\}$ and $\{1, \dots, n\}$. Obviously, there are $n!$ such bijections. The product of two elements f and g in a group of functions is defined by composition, i.e. $f \cdot g = f \circ g$. For a finite group $\mathcal{G}$ and a subset of elements $\mathcal{F} \subseteq \mathcal{G}$ the group product of $\mathcal{F}$ in $\mathcal{G}$ is the subgroup $\langle \mathcal{F} \rangle$ defined by

$$\langle \mathcal{F} \rangle = \{ f \in \mathcal{G} \mid \exists m \exists f_1, \dots, f_m \in \mathcal{F} : f = f_1 \cdot \dots \cdot f_m \}.$$

The elements of $\mathcal{F}$ are called *generators* of $\langle \mathcal{F} \rangle$.

A group $\mathcal{G}$ *operates* on a set $\mathcal{M}$ with respect to a function $\sigma : \mathcal{G} \times \mathcal{M} \to \mathcal{M}$, if for the neutral element $e \in \mathcal{G}$ and all $f, g \in \mathcal{G}$ and $x \in \mathcal{M}$ it holds that $\sigma(e, x) = x$ and $\sigma(f \cdot g, x) = \sigma(f, \sigma(g, x))$. Then, $\mathcal{G}$ and σ induce an equivalence relation on $\mathcal{M}$ in the following way:

$$x \sim y \iff \exists f \in \mathcal{G} : \ \sigma(f,\ x) = y.$$

We call the equivalence class of x, i.e. the set $\{\sigma(f,\ x) \mid f \in \mathcal{G}\}$, the *orbit* of x. The set of all equivalence classes of $\mathcal{M}$ with respect to $\mathcal{G}$ and σ is called the *orbit partition*. In our case a subgroup $\Phi \subseteq \mathrm{Aut}(G)$ of the automorphism group of a graph $G = (V, E)$ will operate on V. For an automorphism $\phi \in \Phi$ and a vertex $v \in V$ the function σ is simply defined by $\sigma(\phi,\ v) = \phi(v)$.

The Search Tree $\mathcal{T}$

In the following $G = (V, E)$ is the undirected graph whose label $C(G)$ we want to compute. Let the cardinality of V be n. We first fix an initial indexing of the vertices $V = \{v_1, \dots, v_n\}$. We now give a formal definition of what we mean by *vertex partition*.

Definition 12.1.1 (Vertex partition). *A* vertex partition *of G is an ordered list $\Pi = \big(V_1, \dots, V_r\big)$ of vertex subsets $V_i \subseteq V$, the so-called* cells, *with*

1. $V_i \cap V_j = \emptyset$, $1 \le i \ne j \le r$
2. $\bigcup_{i \in \{1,\dots,r\}} V_i = V$
3. $|V_i| \ge 1$, $1 \le i \le r$.

The number r of vertex subsets of Π is denoted by $|\Pi|$. A vertex partition Π is called unit partition *if $r = 1$ and* discrete partition *if $r = n$.*

From now on, by vertex partition we will always mean a vertex partition of G. Any node of $\mathcal{T}$ corresponds to a vertex partition with which we will identify that node. To specify these vertex partitions, we have to introduce a refinement procedure f in advance. For a vertex partition Π, $f(\Pi)$ will be a refinement of Π, i.e. for each cell V' in $f(\Pi)$ there will be a cell V in Π with $V' \subseteq V$. The refinement is arranged such that vertices that have 'equal' adjacencies are grouped together. For a vertex $v \in V$ and a vertex set $W \subset V$ let $d(v, W)$ be the number of vertices in W that are adjacent to v. For simplicity assume that we want to compute the refinement $f(\Pi)$ of the unit partition $\Pi = (V)$. In the first refinement step, the number $d(v, V)$ is computed for each vertex v, which simply means the degree of v. Then, the vertices are partitioned according to their degrees, i.e. the result of this first refinement step is a partition $\Pi_1 = (W_1, \dots, W_j)$ in which any two vertices of each cell are of same degree and for a vertex $v \in W_k$ and a vertex $w \in W_\ell$ it holds that $d(v, V) < d(w, V)$ if and only if $k < \ell$. Next, each cell of Π_1 is refined with respect to Π_1. We proceed in basically the same manner as before. For each vertex v of a cell W_i its number $\eta(v) = \big(d(v, W_1), \dots, d(v, W_j)\big)$ is computed and the vertices of W_i are partitioned according to these numbers. (Two vectors are compared according to their lexicographical order.) Doing this for all cells results in a refined partition Π_2. Partition Π_3 is then the refined partition of Π_2 and so on. This is done as long as Π_{i+1} is a real refinement of Π_i, see Algorithm 27.

Note that the partition $f(\Pi) = (V_1, \dots, V_{r'})$ fulfills the following property: for any two (not necessarily distinct) cells V_i, V_j of $f(\Pi)$ and for any two vertices

Algorithm 27: Refinement procedure $f(\Pi)$

Input: A vertex partition $\Pi = (V_1, \ldots, V_r)$.
Output: The refined vertex partition $f(\Pi)$.

```
Π_new = Π
repeat
    Π_old = Π_new
    let Π_old = (V_1, ..., V_r')
    for i = 1 to r' do
        for each v ∈ V_i do
            compute η(v) = (d(v, V_1), ..., d(v, V_r'))
        partition V_i into W_1, ..., W_j such that for v ∈ W_k, w ∈ W_ℓ:
                                η(v) < η(w) ⟺ k < ℓ
        replace V_i in Π_new by W_1, ..., W_j
until Π_new = Π_old
return Π_new
```

$v, w \in V_i$ it holds that $d(v, V_j) = d(w, V_j)$. A partition that satisfies this property has been called *equitable* in Section 9.3.1, where the same method has been discussed. We say that two vertices are *structurally equivalent* (w.r.t. $f(\Pi)$) if they lie in the same cell of $f(\Pi)$.

Now, we can precisely describe the nodes of $\mathcal{T}$. All nodes will correspond to equitable partitions. The root $\Pi = \big(V_1, \ldots, V_r\big)$ corresponds to the refinement of the unit partition $f\big((V)\big)$. If Π is already a discrete partition, Π has no descendants and $\mathcal{T}$ consists of just one node, otherwise the descendants of Π are derived as follows: let $V_i = \{v'_1, \ldots, v'_m\}$ be the first *non-trivial* cell of Π, i.e. the first cell that contains more than one vertex. Then, Π has m descendants, namely $f(\Pi \setminus v'_1), \ldots\ , f(\Pi \setminus v'_m)$, where $f(\Pi \setminus v'_j)$ is short hand for $f(\big(V_1, \ldots, V_{i-1}, \{v'_j\}, V_i \setminus \{v'_j\}, V_{i+1}, \ldots V_r\big))$. This means that we take each vertex $v' \in V_i$ out of V_i once, define $\{v'\}$ artifically as new cell and refine this partition in order to get the descendant $f(\Pi \setminus v')$. This makes sense as Π was equitable before, i.e. any two vertices of one cell of Π were structurally equivalent, and we now check each possibility to refine Π by removing each vertex v' out of V_i and make it an artificial cell.

For any other node $\Pi' \in \mathcal{T}$ that does not correspond to a discrete partition, the descendants are derived in exactly the same manner as for Π. Hence, all leaves of $\mathcal{T}$ correspond to discrete partitions. The order of such a discrete partition $\big(\{v_{\delta(1)}\}, \ldots, \{v_{\delta(n)}\}\big), \delta \in S_n$ determines the adjacency matrix of the corresponding leaf. Recall that the purpose of f is to make $\mathcal{T}$ as small as possible by means of structurally equivalent vertices with respect to the current partition. However, the real size of $\mathcal{T}$ depends on the structure of G. For the example graph in Figures 12.2, $\mathcal{T}$ has only three nodes, while the search tree $\mathcal{T}$ of the example graph in Figure 12.3, which contains somewhat more regular structures than the graph of Figure 12.3, is much bigger.

McKay now defines the label $C(G)$ as minimum adjacency matrix found among all leaves of $\mathcal{T}$. This is indeed a canonical label:

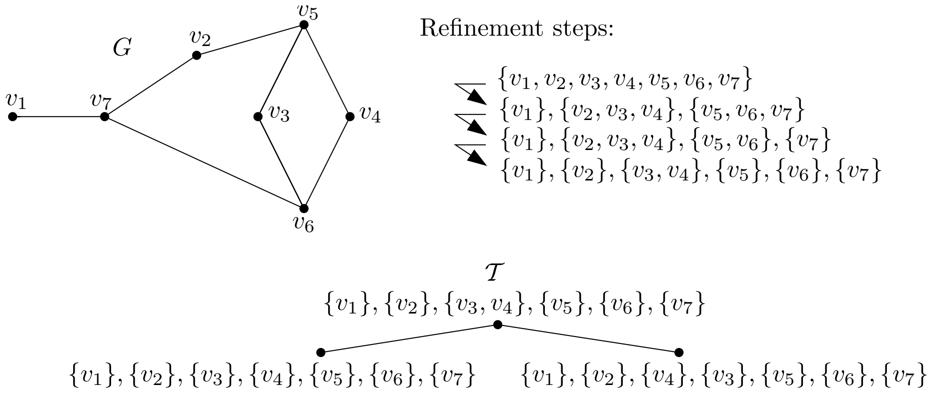


Fig. 12.2. A graph G and the corresponding search tree $\mathcal{T}$. At the beginning only v_3 and v_4 are structurally equivalent

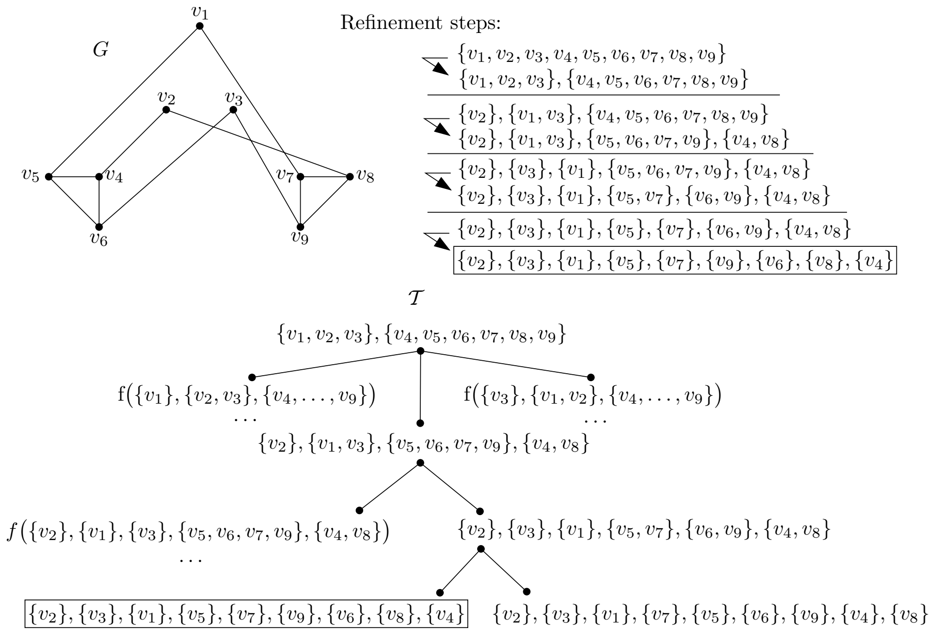


Fig. 12.3. A graph G, an extract of the corresponding search tree $\mathcal{T}$ and the refinement steps for the emphasized path in $\mathcal{T}$. At the beginning the vertices in $\{v_1, v_2, v_3\}$ and in $\{v_4, \ldots, v_9\}$ are structurally equivalent

Theorem 12.1.2. *Let G_1 and G_2 be two undirected graphs. Let $C(G_1)$ and $C(G_2)$ be the labels that were derived from the corresponding search trees. It holds that*

$$C(G_1) = C(G_2) \Longleftrightarrow G_1 \simeq G_2.$$

On one hand it is clear that non-isomorphic graphs G_1 and G_2 cannot have the same label as each adjacency matrix of G_1 is different from each adjacency matrix of G_2 (otherwise the graphs would be isomorphic). In the other direction the clear prescript to generate the search trees gives a hint that two isomorphic graphs really get the same label. Of course, this has to be proven exactly. However, the proof is very technical. We refer the interested reader to [415, Theorem 2.19].

Using Automorphisms to Prune $\mathcal{T}$

The nauty algorithm does not compute $\mathcal{T}$ explicitly. Instead the algorithm parses $\mathcal{T}$ in a special early-to-late order and tries to exclude as many subtrees from the search. Actually, the partition that corresponds to a node is not computed until the node is visited by the search. When the algorithm reaches a leaf ℓ, the adjacency matrix A_ℓ induced by ℓ is computed. During the traversal the algorithm maintains the minimum adjacency matrix $A_{\min}$ it has found so far. When the algorithm reaches the first leaf ℓ_1, $A_{\min}$ is initialized by A_{ℓ_1}. When another other leaf ℓ is reached, it is tested whether $A_\ell < A_{\min}$ and, if so, $A_{\min}$ is set to A_ℓ. Thus, at the end $A_{\min}$ contains the label $C(G)$. Additionally, the algorithm maintains the subgroup $\Phi_t(G)$ of the automorphism group of G that has been computed so far. We will denote this group by $\Phi_t(G)$. It holds that $\Phi_t(G) = \langle \phi_1, \ldots, \phi_{i(t)} \rangle$, where $\phi_1, \ldots, \phi_{i(t)}$ are all automorphisms that we know at time t. An automorphism ϕ is found when two leaves induce equal adjacency matrices: let $w_1, \ldots, w_n$ and $w'_1, \ldots, w'_n$ be the vertex orders of the two leaves. Then, $\phi : w_i \to w'_i$ for $i = 1, \ldots, n$ is an automorphism, see Figure 12.4.

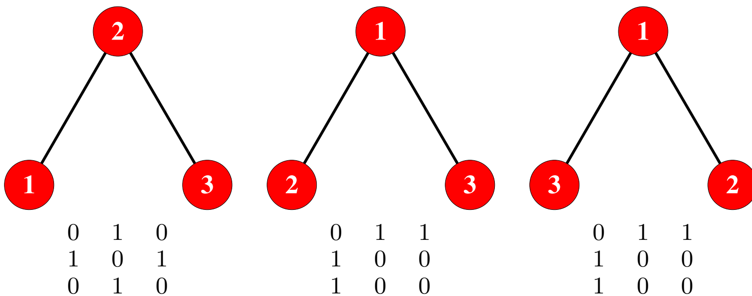


Fig. 12.4. Recognizing automorphisms: the matrix below each graph is the adjacency matrix induced by the particular labeling. If and only if two matrices are equal, the mapping that matches identically labeled vertices is an automorphism

To see how $\mathcal{T}$ can be pruned, we need some more definitions. First a linear order on the nodes in $\mathcal{T}$ is introduced to establish the early-to-late order. Let Π be an inner vertex of $\mathcal{T}$. We denote the subtree rooted at a descendant $f(\Pi \setminus v_i)$ of Π by $\mathcal{T}(\Pi \setminus v_i)$.

Definition 12.1.3 (Linear order on the nodes of $\mathcal{T}$). *Let Π_1, Π_2 be two different nodes of $\mathcal{T}$ and let Π be the least common ancestor of Π_1 and Π_2 in $\mathcal{T}$. We define Π_1 '$<$' Π_2 if $\Pi_1 = \Pi$ or if for the vertices v_i and v_j in the first non-trivial cell of Π with $\Pi_1 \in \mathcal{T}(\Pi \setminus v_i)$ and $\Pi_2 \in \mathcal{T}(\Pi \setminus v_j)$ it holds that $i < j$. Otherwise Π_2 '$<$' Π_1.*

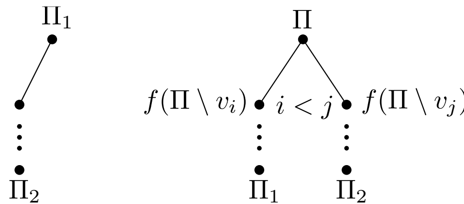


Fig. 12.5. Linear order on the nodes of $\mathcal{T}$: the two cases where Π_1 '$<$' Π_2

It is easy to see that the relation '$<$' is a linear order, see also Figure 12.5. The nauty algorithm traverses the nodes of $\mathcal{T}$ with respect to this order. Next, we need an equivalence relation on the nodes of $\mathcal{T}$.

Definition 12.1.4 (Equivalence relation on the nodes of $\mathcal{T}$).
Let $\Pi_1 = (V_1, \ldots, V_m) \in \mathcal{T}$ and $\Pi_2 = (W_1, \ldots, W_m) \in \mathcal{T}$. Then $\Pi_1 \sim \Pi_2$ if and only if there is an automorphism $\phi \in \mathrm{Aut}(G)$ and a permutation $\delta \in S_m$ such that $\phi(V_i) = W_{\delta(i)}$ for $i = 1, \ldots, m$. We say that ϕ witnesses *$\Pi_1 \sim \Pi_2$.*

Automorphisms that witness the equivalence of two partitions can be thought of as color-preserving automorphisms. For each cell V_i of Π_1 color its vertices in a distinct color and color the vertices of cell $W_{\delta(i)}$ in the same color. Then there exists an automorphism ϕ that preserves the color of each vertex. We can now state the first of two important theorems on the way to prune $\mathcal{T}$.

Theorem 12.1.5. *Let $\Pi_1 \sim \Pi_2 \in \mathcal{T}$ and let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the subtrees of $\mathcal{T}$ rooted at Π_1 and Π_2, respectively. Then for each node $\Pi_1' \in \mathcal{T}_1$ there is a node $\Pi_2' \in \mathcal{T}_2$ with $\Pi_1' \sim \Pi_2'$.*

For the proof we refer to [415], Theorem 2.14. As an immediate consequence of Theorem 12.1.5 we can discard the subtree $\mathcal{T}_2$ rooted at a node $\Pi_2 \in \mathcal{T}$ if we know that there is a node Π_1 with Π_1 '$<$' Π_2 and $\Pi_1 \sim \Pi_2$. This is due to the fact that each leaf of $\mathcal{T}_2$ is equivalent to a leaf of the subtree rooted at Π_1. Thus, we have already seen all adjacency matrices that would be induced by the leaves of $\mathcal{T}_2$. We have to see how $\Phi_t(G)$ is applied to find equivalent inner nodes. For a vertex $v \in V$, $\{\phi(v) \mid \phi \in \Phi_t(G)\}$ is the orbit of v with respect to

$\Phi_t(G)$. Let Θ_t be the orbit partition of V at time t. The algorithm has access to Θ_t at any time. Initially Θ_t is the discrete partition, i.e. $\Theta_0 = \{v_1\}, \ldots, \{v_n\}$. Every time a new automorphism is discovered, Θ_t is updated. This means Θ_t is getting coarser as the new automorphism can enlarge $\Phi_t(G)$ and thus vertices can become equivalent (w.r.t. $\Phi_t(G)$) that were not equivalent before. We can now detect equivalent descendants of a node $\Pi \in \mathcal{T}$ by means of the following theorem which corresponds to Theorem 2.15. in [415].

Theorem 12.1.6. *Let $\Pi = \big(V_1, \ldots, V_r\big) \in \mathcal{T}$ and $V_i = \{v'_1, \ldots, v'_m)$ be the first non-trivial cell of Π. If there are $v'_i, v'_j \in V_i$ that lie in the same orbit of Θ_t, there is an automorphism $\phi \in \Phi_t(G)$ that witnesses $f(\Pi \setminus v'_i) \sim f(\Pi \setminus v'_j)$.*

This theorem is used to prune $\mathcal{T}$ in two ways. The first is obvious: assume the algorithm reaches a node $\Pi \in \mathcal{T}$ whose first non-trivial cell is V_i. Then, Θ_t induces a partition of V_i into cells such that any two vertices of each cell lie in the same orbit. We denote this partition by $\Theta_t \wedge V_i$. According to Theorem 12.1.6, we have to consider only one descendant $\mathcal{T}(\Pi \setminus v')$ for each cell $\Theta_t \wedge V_i$. Namely v' is the vertex that is minimal in its cell, i.e. has the lowest initial index out of all vertices in its cell. In other words, we have to consider the descendants that are derived by the minimal cell representatives of $\Theta_t \wedge V_i$.

The second way is a bit trickier. Assume that the algorithm reaches a node $\Pi \in \mathcal{T}$ at time t_1. Again let V_i be the first non-trivial cell of Π, and let $v_i, v_j \in V_i$ be vertices that do not lie in the same orbit w.r.t. Θ_{t_1}. This means that the algorithm will examine $\mathcal{T}(\Pi \setminus v_i)$ and $\mathcal{T}(\Pi \setminus v_j)$ by the information that it gets from Θ_{t_1}. W.l.o.g. let v_i have the smaller initial index than v_j, and thus $\mathcal{T}(\Pi \setminus v_i)$ will be examined before $\mathcal{T}(\Pi \setminus v_j)$. The algorithm proceeds, and at time t_2 it finds a new automorphism ϕ' such that now there is an automorphism $\phi \in \Phi_{t_2}(G)$ with $\phi(v_i) = v_j$. Hence, v_i and v_j lie in the same orbit w.r.t. Θ_{t_2}. (Note that ϕ is not necessarily the new automorphism ϕ' itself but a composition of ϕ' and automorphisms that have been found before.) Now, the algorithm has the information that $\mathcal{T}(\Pi \setminus v_j)$ can be pruned. Of course, this cannot be taken into account anymore if t_2 is after the examination of $\mathcal{T}(\Pi \setminus v_j)$ has been completed. Otherwise this examination can be discarded or (if $\mathcal{T}(\Pi \setminus v_j)$ is already being examined) aborted. If the algorithm indeed aborts the examination of a subtree $\mathcal{T}(\Pi \setminus v_j)$ and jumps back to Π, a new automorphism has just been found such that Θ_t allows this step. Now, it might even be possible to jump back to an ancestor of Π because Θ_t now also allows to abort the examination of a subtree in which Π is contained. Actually, when a new automorphism is found, the algorithm immediately checks how far it can jump back in $\mathcal{T}$ by means of the new information.

A challenge is to determine an appropriate number of adjacency matrices to be stored. Storing and comparing adjacency matrices needs a lot of time and space. However, if the algorithm maintains a large number of adjacency matrices, the number of detected automorphisms will also be higher. Thus $\mathcal{T}$ can be pruned more efficiently which in turn will again decrease the running time. McKay claims that the storage of only two adjacency matrices has stood

Algorithm 28: NautyAlgorithm $\big(G = (V, E), V = \{v_1, \dots, v_n\}\big)$

Input: A graph $G = (V, E)$ and an initial vertex indexing $V = \{v_1, \dots, v_n\}$.
Output: The label $C(G)$.

adj.matrix A_{ℓ_1}, vertex_order(A_{ℓ_1}) = nil
adj.matrix $A_{\min}$, vertex_order($A_{\min}$) = nil
$\Phi(G) = \{\mathrm{id}\}$
$\Theta = \{v_1\}, \dots, \{v_n\}$
process$\big(f\big((V)\big)\big)$
return $A_{\min}$.

---

process$\big(\Pi = (V_1, \dots, V_r)\big)$
if $r = n$ **then**
  identify $V_1 = \{v'_1\}, \dots, V_n = \{v'_n\}$ with vertex order $v'_1, \dots, v'_n$
  compute adj. matrix A_Π induced by $v'_1, \dots, v'_n$
  if A_{ℓ_1} = nil **then**
    $A_{\ell_1} = A_\Pi$, vertex_order(A_{ℓ_1}) $= v'_1, \dots, v'_n$
    $A_{\min} = A_\Pi$, vertex_order($A_{\min}$) $= v'_1, \dots, v'_n$
  else
    if $A_{\min} > A_\Pi$ **then** $A_{\min} = A_\Pi$, vertex_order($A_{\min}$) $= v'_1, \dots, v'_n$
    else
      ϕ = nil
      if $A_{\ell_1} = A_\Pi$ **then**
        compute automorphism ϕ induced
        by vertex_order(A_{ℓ_1}) and $v'_1, \dots, v'_n$
      if $A_{\min} \neq A_{\ell_1}$ and $A_{\min} = A_\Pi$ **then**
        compute automorphism ϕ induced
        by vertex_order($A_{\min}$) and $v'_1, \dots, v'_n$
      if $\phi \neq$ nil **then**
        $\Phi(G) = \langle\{\Phi(G) \cup \phi\}\rangle$
        update Θ
        check jump back
else
  let $V_i = \{v'_1, \dots, v'_m\}$ be the first non-trivial cell of Π
  let $v''_1, \dots, v''_{m'}$ be the minimum cell representatives of $\Theta \wedge V_i$
  for $j = 1$ **to** m' **do** process$\big(f(\Pi \setminus v''_j)\big)$

the test in practice. At any time, the nauty algorithm stores two adjacency matrices, the matrix A_{ℓ_1} of the first visited leaf and $A_{\min}$. We summarize the nauty algorithm in Algorithm 28. For simplification we have omitted a detailed, rather complicated description of the jump-back steps.

12.1.3 The Difficulty of GI or 'How to Trick Nauty'

Recall that the complexity status of GI is not yet known. Assume we strongly believe that there is a polynomial algorithm and we want to try to solve GI polynomially (many people have in fact tried to derive such an algorithm). The obvious way to do it would be to use an idea similar to that of McKay. This section tries to illustrate why it seems to be hard to succeed in solving GI like this; we show that even elaborate approaches fail. In principle we want to proceed as in the nauty algorithm, but to use a different refinement procedure and compute only *one* leaf of the search tree. The label $C(G)$ is then again defined as the adjacency matrix induced by the vertex order of this leaf. As stated before two non-isomorphic graphs G_1 and G_2 are never recognized as isomorphic because each adjacency matrix of G_1 is different from any adjacency matrix of G_2. To make sure that two isomorphic graphs G_1 and G_2 are recognized as isomorphic we want to ensure the following: Let Π_k be the leaf in the search tree of G_1 that has been computed and that defines $C(G_1)$. Let Π'_k be the leaf in the search tree of G_2 that defines $C(G_2)$. Let $\Pi_1, \ldots, \Pi_k$ and $\Pi'_1, \ldots, \Pi'_{k'}$ be the vertex partitions that have been computed in order to get to Π_k and $\Pi'_{k'}$, respectively. Then it should hold that $k = k'$ and for $i = 1, \ldots, k$ each vertex partition Π_i matches Π'_i in terms of number of cells and cardinality of each cell. Finally we need that for $\Pi_i = (V_1, \ldots, V_r)$ and $\Pi'_i = (V'_1, \ldots, V'_r)$ and for each pair $V_j = \{v_1, \ldots, v_m\}, V'_j = \{v'_1, \ldots, v'_m\}$ of cells the following holds: for all $(v, v') \in V_j \times V'_j$ there is an isomorphism ϕ between G_1 and G_2 with $\phi(v) = v'$. This last condition justifies our computing only one leaf of the search tree. Then it is irrelevant which vertices v and v' we take out of the first non-trivial cells of Π_i and Π'_i, define artificially as new equivalence classes and refine according to these new partitions. To see this, note that if later $C(G_1)$ really equals $C(G_2)$, the corresponding vertex orders of the two leaves induce an isomorphism ϕ between G_1 and G_2. Defining $\{v\}$ and $\{v'\}$ as new equivalence classes simply means that we fix $\phi(v) = \phi(v')$ and refine with respect to this information. And if there is really an isomorphism ϕ that maps v onto v', which is guaranteed by the last condition, we will still find it.

To illustrate that it seems difficult to solve GI polynomially in the way described above, we give two counterexamples. First, we look at the refinement procedure f used in the nauty algorithm. The 3-regular graph G in Figure 12.6 proves that f does not help to solve GI polynomially.

For this graph $G = (V, E)$, it holds that $d(v, V)$ equals $d(w, V)$ for any two vertices $v, w \in V$, since G is regular. Thus, the unit partition is not further refined by f. If we now have two copies G_1 and G_2 of G and take v_1 out of V to derive $C(G_1)$ while we take v_2 out of V to derive $C(G_2)$, we will come to the false conclusion that G_1 and G_2 are non-isomorphic. There is no isomorphism

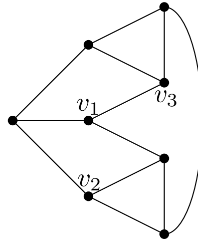


Fig. 12.6. The 3-regular graph G

that maps v_1 onto v_2: from v_1 the distance to any other vertex is 2, while the distance from v_2 to v_3 is 3.

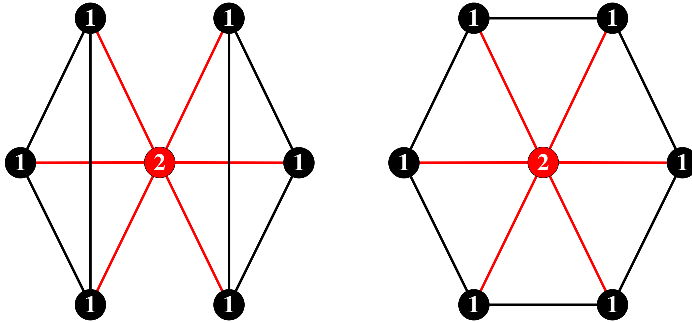


Fig. 12.7. Graph with two components

We now want to see what happens if we apply a different refinement procedure that uses more information than adjacencies to other cells. Recall that the idea of f was to partition the vertex set in equivalence classes as long as it holds that any two vertices of one cell have the same number of neighbors in each other cell. For $v \in V, W \subseteq V$ and $i \in \mathbb{N}$ let now $d_i(v, W)$ be the number of vertices in W of distance i to v. We try to improve f and refine as long as the following holds: for any two vertices v and w of one cell the numbers $d_i(v, W)$ and $d_i(w, W)$ are equal w.r.t. each cell W and each $i \in \mathbb{N}$. The 3-regular graph is no longer a counterexample for this refinement procedure. However, the new method also fails, as the graph in Figure 12.7 shows. The label of each vertex v corresponds to the equivalence class to which v belongs after the refinement of the unit partition. Each 1–vertex has two 1–vertices and one 2–vertex at distance 1 and three 1–vertices at distance 2, while each 2–vertex has six 1–vertices at distance 1. Obviously, there is no isomorphism that maps the 2–vertex of the left component onto the 2–vertex of the right component. For simplification the graph consists of two components, but the graph can be extended, resulting in a connected graph which yields the same result.

12.2 Graph Similarity

The graph isomorphism problem asks if two graphs have identical structure. As this is a very restrictive criterion, one may consider the natural relaxation which tries to specify how similar two graphs are. *Graph similarity*, often called *graph matching*, compares two graphs to give a measure for the similarity, or distance, between them.

There are various applications of this problem, i.e., CAD/CAM, computer vision, and molecule matching. An important advantage of graph similarity over isomorphism is its ability to cope with errors and distortions in the input data, which often occurs when collecting real world data. These errors can change isomorphic graphs to non-isomorphic ones, so a rigorous check for isomorphism is inappropriate. The alternative is an imprecise matching using a graph similarity measure.

Many applications imply a labeling of the vertices or edges, i.e., in molecule matching the labeling is defined by the types of the elements. When labels are present vertices and edges with different labels are either penalized or even not allowed to match. Since we are interested in structural similarity, all graphs are regarded as unlabeled in the following.

There are certain properties a meaningful similarity measure should fulfill. For example, the distance from graph G_1 to graph G_2 should be the same as from G_2 to G_1, and the distance of isomorphic graphs should be 0. An common formalization of such properties is a *graph distance metric*.

Definition 12.2.1. *Let G_1, G_2, and G_3 be graphs. A function $d : G_1 \times G_2 \to \mathbb{R}_0^+$ is called a* graph distance metric *if the following properties hold:*

$$\textbf{\textit{reflexivity:}} \quad d(G_1, G_2) = 0 \Leftrightarrow G_1 \cong G_2 \tag{12.1}$$

$$\textbf{\textit{symmetry:}} \quad d(G_1, G_2) = d(G_2, G_1) \tag{12.2}$$

$$\textbf{\textit{triangle inequality:}} \quad d(G_1, G_2) + d(G_2, G_3) \geq d(G_1, G_3) \tag{12.3}$$

On the other hand, all graph distance metrics are hard to compute since the reflexivity property implies a solution for graph isomorphism. Thus, in practice, one may either relax these properties, or compute an approximation of the measure.

For simplicity, only undirected connected graphs are considered in the following. All statements can be extended to unconnected graphs by considering their connected components, and also to directed (strongly connected) graphs.

We present three types of similarity measures. Two are metrics: one is based on the size of a maximum common subgraph, and the other on the difference in the length of corresponding paths. Another approach defines the distance between two graphs in terms of edit operations needed to transform one into the other. Finally we give a short overview of other methods from literature

12.2.1 Edit Distance

A general and flexible method for matching structural objects is the concept of edit distance. Given a set of allowed edit operations on the objects, the distance

between two objects is defined as the minimal number of operations needed to transform one into the other. A well-known example is string edit distance.

In graph edit distance typical operations include the insertion, deletion, and substitution of vertices and edges. There is no general agreement on the set of allowed operations. Instead, a good selection of allowed operations is very application-dependent. Furthermore, non-negative costs can be assigned to operations to better fit special requirements. In this case the distance is defined as the minimum cost taken over all sequences of operations that transform one graph into the other.

Intuitively speaking, for reasonable and meaningful specifications of operations and costs, the problem is hard to solve. For certain combinations of operations and costs the metric properties are satisfied. Recall this implies the problem is at least as hard to solve as GI. On the other hand the distance is efficiently computable only for simple sets of allowed operations. In this case the resulting distance is less significant.

Example 1. The first example illustrates a specification which is easy to handle but does not lead to very meaningful results. The following edit operations are allowed:

– vertex insertion - a new (isolated) vertex is added to the graph
– vertex deletion - a (isolated) vertex is deleted from the graph
– edge insertion - a new edge is added between arbitrary vertices of the graph
– edge deletion - an edge is deleted from the graph

The costs of both vertex operations are one, of both edge operations zero. It is easy to see that the distance defined by this specification is equal to the difference of the number of vertices of the two graphs:

$$d_{\mathrm{exp1}} = ||V(G_1)| - |V(G_2)|| \ .$$

This means, for example, a path, a star, and a clique of the same number of vertices are equal in terms of this distance.

Example 2. This specification was introduced by Papadopoulos and Manolopoulos [464]. They propose to use three operations, all with cost one:

– vertex insertion – a new (isolated) vertex is added to the graph
– vertex deletion – a (isolated) vertex is deleted from the graph
– edge update – one endvertex of an edge is changed

Insertion or deletion of an edge requires two edge updates in this model. Using these operations on the graphs of Figure 12.8, two operations are required to match G_1 with G_2, namely two edge updates, whereas three operations are required to match G_1 with G_3, namely one vertex insertion and two edge updates. Thus, in this specification, G_1 is more similar to G_2 than to G_3.

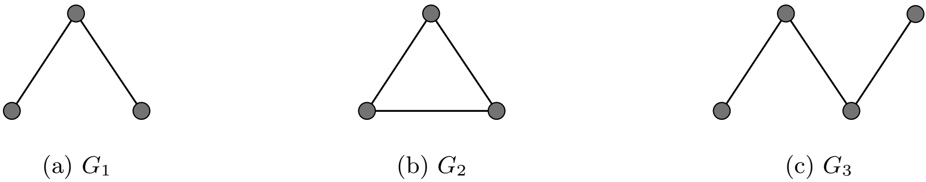


Fig. 12.8. Similarity among graphs: in Example 2, G_1 is more similar to G_2 than to G_3

As already mentioned, the computation of a meaningful graph edit distance is hard. Therefore, the matching condition is relaxed: given two graphs G_1 and G_2, instead of transforming G_1 into G_2, G_1 is transformed into a graph with the same number of vertices and edges and the same degree sequence as G_2. In other words, only the size and the degree sequence of the graphs are considered.

A *degree vector* $x = (x_1, \ldots, x_n)$ of a graph $G = (\{v_1, ..., v_n\}, E)$ with n vertices is defined by $x_i := d(v_i)$. A *graph histogram* is a degree vector whose entries are incremented by one and sorted in decreasing order. Given two graphs G_1 and G_2, the distance according to the L_1 metric of the corresponding graph histograms gives the minimum number of operations required to transform G_1 into a graph with the same number of vertices and edges and the same degree sequence as G_2. If the number of vertices of the two graphs differs, zeros are added to the smaller graph histogram.

12.2.2 Difference in Path Lengths

The next similarity measure we present is an example of a graph distance metric [116]. Hence, while the definition is quite simple, its computation is hard. Roughly speaking, the sum of differences of the lengths of corresponding paths for all pairs of vertices is considered. Since this measure is reasonable only for graphs of the same number of vertices, only such graphs are compared in the following.

Definitions. Let G_1, G_2 be isomorphic connected graphs with isomorphism $\phi : V(G_1) \to V(G_2)$. Two vertices of G_1 are adjacent iff their isomorphic vertices in G_2 are adjacent, in other words:

$$\forall u, v \in V(G_1) : \{u, v\} \in E(G_1) \Leftrightarrow \{\phi(u), \phi(v)\} \in E(G_2) .$$

An equivalent formulation extends this connection property from distance-one vertices to arbitrary pairs of vertices:

$$\forall u, v \in V(G_1) : d_{G_1}(u, v) = d_{G_2}(\phi(u), \phi(v)) . \tag{12.4}$$

Now, let G_1, G_2 be two arbitrary connected graphs of the same number of vertices and $\sigma : V(G_1) \to V(G_2)$ a bijection. Then, Equation 12.4 does not

necessarily hold anymore. Instead, we can use the differences of the path lengths to define the similarity of two graphs with respect to σ.

Definition 12.2.2. *For two connected graphs G_1, G_2 of the same number of vertices and a bijection $\sigma : V(G_1) \to V(G_2)$ we define the* σ-distance d_σ *by*

$$d_\sigma(G_1, G_2) = \sum_{\{u,v\} \in V(G_1) \times V(G_1)} |d_{G_1}(u, v) - d_{G_2}(\sigma(u), \sigma(v))| ,$$

where the sum is taken over all unordered pairs of vertices of G_1.

Since the similarity of two graphs can not depend on a specific mapping between the sets of vertices, the distance is defined as the minimum over all possible bijections between $V(G_1)$ and $V(G_2)$.

Definition 12.2.3. *For two connected graphs G_1, G_2 of the same number of vertices, we define the* path distance d_{path} *by*

$$d_{path}(G_1, G_2) = \min_{\sigma \in \Lambda} d_\sigma(G_1, G_2) ,$$

where Λ is the set of all bijections between $V(G_1)$ and $V(G_2)$.

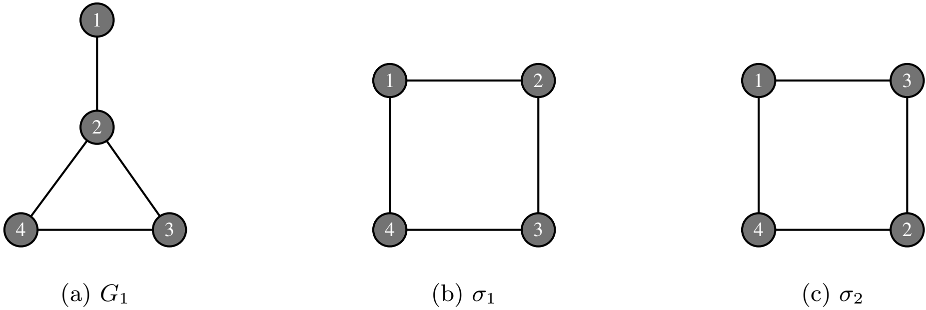
Example. Let G_1 be the graph shown in Figure 12.9 and let G_2 be a cycle of 4 vertices. At first sight there are $4! = 10$ bijective mappings from $V(G_1)$ to $V(G_2)$. However, because of the highly symmetric structure of the graphs, there are only two inequivalent mappings with respect to path distance. These are depicted in Figure 12.9, where the mappings σ_1, $\sigma_2 : V(G_1) \to V(G_2)$ are defined by $\sigma_i j = j$ for $j = 1, \ldots, 4$. Now we determine for each pair of vertices the difference between distance in G_1 and distance of the corresponding images in G_2 and find that

$$d_{\sigma_1}(G_1, G_2) = 2 \quad \text{and} \quad d_{\sigma_2}(G_1, G_2) = 4 .$$

Thus $d_{\text{path}}(G_1, G_2) = 2$.

Path Distance Is a Metric. $d_{\text{path}}(G_1, G_2) = d_{\text{path}}(G_2, G_1)$ follows directly from the definition. From Equation 12.4 we get immediately $d_{\text{path}}(G_1, G_2) = 0$ for isomorphic graphs. On the other hand, $d_{\text{path}}(G_1, G_2) = 0$ implies the existence of an isomorphism $\phi : V(G_1) \to V(G_2)$.

The triangle inequality remains to be verified. Let G_1, G_2, and G_3 be connected graphs with $|V(G_1)| = |V(G_2)| = |V(G_3)|$, and $\alpha : V(G_1) \to V(G_2)$ and $\beta : V(G_2) \to V(G_3)$ bijections with $d_\alpha(G_1, G_2) = d_{\text{path}}(G_1, G_2)$ and $d_\beta(G_2, G_3) = d_{\text{path}}(G_2, G_3)$ respectively. Then $\beta \circ \alpha : V(G_1) \to V(G_3)$ is also a bijection and



(a) G_1 (b) σ_1 (c) σ_2

Fig. 12.9. Two different mappings of G_1 to a cycle of 4 vertices

$$\begin{aligned}
d_{\text{path}}(G_1, G_3) &\le d_{\beta\circ\alpha}(G_1, G_3) \\
&= \sum_{\{u,v\}\in V(G_1)\times V(G_1)} |d_{G_1}(u,v) - d_{G_3}((\beta\circ\alpha)(u), (\beta\circ\alpha)(v))| \\
&\le \sum_{\{u,v\}\in V(G_1)\times V(G_1)} |d_{G_1}(u,v) - d_{G_2}(\alpha(u), \alpha(v))| \\
&\quad + \sum_{\{u,v\}\in V(G_1)\times V(G_1)} |d_{G_2}(\alpha(u), \alpha(v)) - d_{G_3}((\beta\circ\alpha)(u), (\beta\circ\alpha)(v))| \\
&= d_\alpha(G_1, G_2) + d_\beta(G_2, G_3) \\
&= d_{\text{path}}(G_1, G_2) + d_{\text{path}}(G_2, G_3)\ .
\end{aligned}$$

Therefore, the triangle inequality holds and d_{path} is a graph similarity metric.

Computation of Path Distance. The computation of path distance of two graphs consists of three steps. First, we compute the distance of all pairs of vertices in both graphs. This is exactly the all-pairs shortest path problem (see Section 2.2.2). Then, we can compute the σ-distance for a given bijection σ in time $\mathcal{O}(n^2)$. Finally, we must identify the minimum bijection with respect to path distance.

12.2.3 Maximum Common Subgraphs

In this section we look at a similarity measure based on the size of a maximum common subgraph. The idea to use similar substructures of graphs for graph matching was introduced by Horaud and Skordas [315] and Levinson [391], and refined by Bunke and Shearer [106].

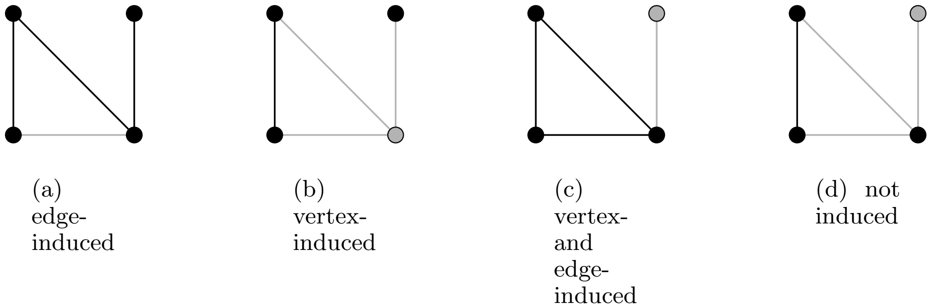
Recall the definition of induced subgraphs in Section 2.1. A graph $G' = (V', E')$ is a *subgraph* of the graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. It is an *induced subgraph* if E' contains all edges $e \in E$ that join vertices in V'.

Definition 12.2.4. *Let G_1, G_2 be undirected graphs. An injective function $\phi : V(G_1) \rightarrow V(G_2)$ is a* subgraph isomorphism *from G_1 to G_2 if there exists an induced subgraph $G_2' \subseteq G_2$ such that ϕ is a graph isomorphism between G_1 and G_2'.*

Definition 12.2.5. *Let G_1, G_2 be undirected graphs. A graph S is a* common induced subgraph *of G_1 and G_2 if there exist subgraph isomorphisms from S to G_1 and G_2.*

Definition 12.2.6. *Let G_1, G_2 be undirected graphs. A common induced subgraph S of G_1 and G_2 is* maximum *if there exists no other common subgraph with more vertices than S. We denote such a maximum common induced subgraph (MCIS) by $mcis(G_1, G_2)$.*

A concept closely related to (vertex-)induced subgraphs are edge-induced subgraphs. A graph $G' = (V', E')$ is a *edge-induced subgraph* of the graph $G = (V, E)$ if $E' \subseteq E$ and V' contains only the incident vertices of edges in E'. Note that edge-induced subgraphs contain no isolated vertices. Figure 12.10 shows a comparison of vertex- and edge-induced subgraphs of a simple graph. The prior definitions for induced subgraphs are easily carried over to edge-induced subgraphs.



(a) edge-induced

(b) vertex-induced

(c) vertex- and edge-induced

(d) not induced

Fig. 12.10. Comparison of vertex- and edge-induced subgraphs

Definition 12.2.7. *Let G_1, G_2 be undirected graphs. An injective function $\phi : V(G_1) \rightarrow V(G_2)$ is an* edge subgraph isomorphism *from G_1 to G_2 if there exists an edge-induced subgraph $S \subseteq G_2$ such that ϕ is a graph isomorphism between G_1 and S.*

Definition 12.2.8. *Let G_1, G_2 be undirected graphs. A graph S is a* common edge subgraph *of G_1 and G_2 if there exist edge subgraph isomorphisms from S to G_1 and to G_2.*

Definition 12.2.9. *Let G_1, G_2 be undirected graphs. A common edge subgraph S of G_1 and G_2 is* maximum *if there exists no other common edge subgraph with more vertices than S. We denote such a maximum common edge subgraph (MCES) by $mces(G_1, G_2)$.*

Note that maximum common subgraphs are neither unique nor connected by definition. Note also that the MCIS or MCES of non-empty graphs consist at least of one vertex or one edge, respectively. Next, induced subgraphs are used to define distance measures for graphs.

Definition 12.2.10. *Let G_1, G_2 be undirected graphs, not both empty. We define the* MCIS distance d_{mcis} *by*

$$d_{mcis}(G_1, G_2) = 1 - \frac{|V(mcis(G_1, G_2))|}{\max(|V(G_1||, |V(G_2)|)} \tag{12.5}$$

and the MCES distance d_{mces} *by*

$$d_{mces}(G_1, G_2) = 1 - \frac{|V(mces(G_1, G_2))|}{\max(|V(G_1)|, |V(G_2)|)} \ . \tag{12.6}$$

MCIS and MCES Distance Are Metrics. Two properties of a graph similarity metric, reflexivity and symmetry, follow directly from the definition. The proof of the triangle inequality consists of a longish case differentiation, so we only give a sketch for MCIS. The complete proof for MCIS is given in [106].

Let G_1, G_2, and G_3 be undirected graphs. For notational convenience, let $n_i = V(G_i)$, $\text{mcis}(i, j) = |V(\text{mcis}(G_i, G_j))|$, and $\max(i, j) = \max(n_i, n_j)$ for $i, j \in \{1, 2, 3\}$. Using this notation, the triangle inequality is equivalent to

$$1 - \frac{\text{mcis}(1,3)}{\max(1,3)} \leq 1 - \frac{\text{mcis}(1,2)}{\max(1,2)} + 1 - \frac{\text{mcis}(2,3)}{\max(2,3)} \ .$$

Next, consider a maximum common subgraph of $\text{mcis}(G_1, G_2)$ and $\text{mcis}(G_2, G_3)$ and denote its number of vertices by $\text{mcis}(12, 23)$. Clearly

$$\begin{aligned} \text{mcis}(12, 23) &\leq \text{mcis}(1, 3) \ , \\ \text{mcis}(12, 23) &\leq \text{mcis}(1, 2) \ , \\ \text{mcis}(12, 23) &\leq \text{mcis}(2, 3) \ , \end{aligned}$$

and

$$\text{mcis}(1, 2) + \text{mcis}(2, 3) - \text{mcis}(12, 23) \leq n_2 \ .$$

Now distinguish six cases by the possible orderings of n_1, n_2, and n_3 and get the result by combining the above inequalities.

Computation of MCIS and MCES. The detection of a maximum common subgraph is an $\mathcal{NP}$-complete problem [240]. Nevertheless a few exact algorithms have been proposed, based either on an exhaustive search for all subgraphs or on the relation of maximum common subgraph and maximum clique detection.

The first method was proposed by McGregor [414] and is very similar to the search-and-backtrack approach to graph isomorphism. The algorithm identifies common subgraphs by starting from single vertices in each graph and iteratively adding vertices (and incident edges) which do not violate the common subgraph condition. If it is impossible to add any new vertex, the size of the current subgraph is compared to the one previously found and a backtracking is done to test other branches of the search tree. Finally, a largest common subgraph is reported.

The second approach is based on the fact that a MCIS of two graphs corresponds to a maximum clique in their modular product graph. Recall a clique is a completely connected subgraph. A *maximum clique (MC)* is a clique with the largest number of vertices. Note that a MC is not necessarily unique. The *modular product graph* $G_1 \diamond G_2$ of G_1 and G_2 is defined on the vertex set

$$V(G_1 \diamond G_2) = V(G_1) \times V(G_2)$$

and two vertices $(u_i, v_i), (u_j, v_j) \in G_1 \diamond G_2$ being adjacent if either

$$(u_i, u_j) \in E(G_1) \text{ and } (v_i, v_j) \in E(G_2)$$

or

$$(u_i, u_j) \notin E(G_1) \text{ and } (v_i, v_j) \notin E(G_2) \,.$$

Accordingly, a MCES of two graphs corresponds to a maximum clique in the modular product graph of their line graphs [450].

Exact algorithms for clique detection are based on exhaustive search strategies [240]. This approach is similar to algorithms for MCIS, but takes advantage of a number of upper and lower bounds to prune the search space (e.g., see [466]). Also, many approximation algorithms have been proposed, see [70] for an extensive survey.

12.2.4 Other Methods

RASCAL. This is not a single method but a combination of a fast initial screening process followed by a rigorous MCES detection algorithm [488]. In the initial screening the degree sequence and vertex and edge labels are considered for computing a first approximation of the similarity. Only if it is above a certain threshold is the costly MCES detection executed. The idea is that one does not care about quite different graphs but only about very similar ones. Other benefits of this paper, beside the two phase approach, are a detailed description of the MCES computation including some minor improvements and a good readability.

Motifs. In Section 11.6 the concept of motifs is introduced. Motifs are small connected subgraphs in a graph G that occur in G significantly more often than in a random graph of the same size and degree distribution. Characteristics and quantity of motifs in graphs can be used as indicators for their similarity.

12.3 Chapter Notes

Besides the detection of explicit algorithms (by finding equal adjacency matrices), an automorphism can sometimes be inferred by a special structure of a vertex partition in $\mathcal{T}$. However, this occurs rarely, for details see [415, Lemma 2.25].

McKay uses another trick in order to prune the search tree $\mathcal{T}$: let Λ be a function defined on the set of all vertex partitions. The goal is now to define an indicator function Λ^* on the nodes of $\mathcal{T}$. In the nauty algorithm a node $\Pi_m \in \mathcal{T}$ actually stores all vertex partitions of its ancestors, i.e. the list of refined partitions $f\big((V)\big) = \Pi_1, \ldots, \Pi_m$ that were derived in order to get to Π_m. Identify the node from now on with $[\Pi_1, \ldots, \Pi_m]$. The function Λ^* is defined by $\Lambda^*([\Pi_1, \ldots, \Pi_m]) = (\Lambda(\Pi_1), \ldots, \Lambda(\Pi_m))$. McKay's algorithm actually searches the minimum adjacency matrix among the leaves that maximize Λ^*. The algorithm can then prune subtrees as soon at it is clear that all their leaves have a Λ^*–value below the current maximum. This is due to the lexicographical order of Λ^*. The benefit of this method depends eminently on the quality of Λ. For example, if Λ is the identity, Λ has no effect. McKay uses information from the computation of $f(\overline{\Pi}) = \Pi$ to define $\Lambda(\Pi)$, see again [415].