Exploring the theoretical and practical aspects of the graph isomorphism problem.

BY MARTIN GROHE AND PASCAL SCHWEITZER

The Graph Isomorphism Problem

Deciding whether two graphs are structurally identical, or isomorphic, is a classical algorithmic problem that has been studied since the early days of computing. Applications span a broad field of areas ranging from chemistry (Figure 1) to computer vision. Closely related is the problem of detecting symmetries of graphs and of general combinatorial structures. Again this has many application domains, for example, combinatorial optimization, the generation of combinatorial structures, and the computation of normal forms. On the more theoretical side, the problem is of central interest in areas such as logic, algorithmic group theory, and quantum computing.

Graph isomorphism (GI) gained prominence in the theory community in the 1970s, when it emerged as one of the few natural problems in the complexity class NP that could neither be classified as being hard (NP-complete) nor shown to be solvable with an efficient algorithm (that is, a polynomial-time algorithm). It was mentioned numerous times as an open problem, in particular already in Karp’s seminal 1972 paper on NP-completeness as well as in Garey and Johnson’s influential book on computers and intractability. Since then, determining the precise computational complexity of GI has been regarded a major open problem in theoretical computer science.

In a recent breakthrough, Babai proved that GI is solvable in quasipolynomial time. This means that on $n$-vertex input graphs, Babai’s algorithm runs in time $n^{p \log n}$ for some polynomial $p(X)$. This can be interpreted as the problem being almost efficiently solvable—-theoretically.

In this paper, we will survey both theoretical and practical aspects of the graph isomorphism problem, paying particular attention to the developments that led to Babai’s result.

Historical development. Graph isomorphism as a computational problem first appears in the chemical documentation literature of the 1950s (for example, Ray and Kirsch) as the problem of matching a molecular graph (see Figure 1) against a database of such graphs. The earliest computer science reference we are aware of is due to Unger, incidentally also in the Communications of the ACM. Maybe the first important step on the theoretical side was Hopcroft and Tarjan’s $O(n \log n)$ isomorphism algorithm for planar graphs. As the question whether GI is NP-complete gained prominence, it was realized that GI has aspects that distinguish it from most NP-complete problems.

key insights

- With its many practical and theoretical applications the graph isomorphism problem remains one of the most important unresolved problems in theoretical computer science.
- A recent breakthrough by László Babai shows that the problem is almost efficiently solvable— theoretically.
- We are only starting to explore the potential of the wealth of new ideas the recent advances bring to the field.
In the mid-1980s, another fascinating facet of the complexity of GI was discovered. Using the newly developed machinery of interactive proof systems, it was shown that the complement of GI has short zero-knowledge proofs\(^ {16} \) and this was seen as another indication that GI is not NP-complete. (If GI is NP-complete, then the so-called polynomial hierarchy of complexity classes above NP collapses to its second level, which is regarded as unlikely.) On the other hand, Torán\(^ {18} \) proved that GI is hard to solve when the available memory is quite limited (specifically it is hard for nondeterministic logarithmic space under logspace reductions), which gives us at least some complexity theoretic lower bound.

As the group theoretic methods have been introduced in the early 1980s, they have been continually refined. The underlying group theory has progressed (for example, Babai et al.\(^ {5,9} \)), the complexity of the group theoretic problems has been analyzed in detail (for example, Luks\(^ {27} \) and Seress\(^ {37} \)), and the scope of the methods has been expanded to other structures (for example, Babai et al.\(^ {4} \)). Another active strand of research has been the design of efficient algorithms for GI restricted to graphs with specific properties (for example, Babai et al.\(^ {5,9} \), Grohe and Marx\(^ {19} \), Lokshtanov et al.\(^ {25} \) Ponomarenko\(^ {34} \)). More recently, there has also been work on memory restricted algorithms (for example, Datta et al.\(^ {14} \)).

But no real progress on the general isomorphism problem was made until—out of the blue—Babai published his quasipolynomial-time algorithm in 2015.

After this historical overview, let us get slightly more concrete.

**Isomorphisms, automorphisms, and canonical forms.** An isomorphism from a graph \( G = (V,E) \) to a graph \( H = (W,F) \) is a one-to-one mapping \( \pi \) from the vertices of the first graph \( V \) onto the vertices of the second graph \( W \) that preserves adjacency and nonadjacency, that is, \( uv \in E \) if and only if \( \pi(u) \pi(v) \in F \) for all pairs \( uv \) of vertices in \( V \) (Figure 2).

An automorphism, or a symmetry, of a graph \( G \) is an isomorphism from \( G \) to \( G \) itself. For example, all \( n! \) permutations of the vertex set of a complete graph \( K_n \) on \( n \) vertices are automorphisms. By comparison, an (undirected) path of length \( n \) only has two automorphisms, the trivial identity mapping, and the mapping that flips the ends of the path. The collection of all automorphisms of \( G \) forms a mathematical structure known as a (permutation) group. As the example of the complete graph shows, automorphism groups can get very large, exponentially large in the number of vertices, but fortunately every permutation group has a generating set linear in the size of the permutation domain (that is, the set of objects being permuted). This allows us to work with automorphism groups efficiently as long as they are represented by sufficiently small generating sets. The problem GI of deciding whether two graphs are isomorphic and the problem of computing a generating set for the automorphism group of a graph (AUT) have the same computational complexity, or more precisely, can be reduced to each other by polynomial-time reductions (see Mathon\(^ {29} \)).

Another important related problem is the graph canonization problem. A canonical form \( \gamma \) maps each graph \( G \) to an isomorphic graph \( \gamma(G) \) in such a way that if graphs \( G \) and \( H \) are isomorphic then the graphs \( \gamma(G) \) and \( (H) \) are identical (not just isomorphic).
Observe that a canonical form $\gamma$ yields an isomorphism test: given $G, H$, compute $\gamma(G)$ and $\gamma(H)$ and check if they are identical. In practical applications, canonical forms are often preferable over isomorphism tests. It is an open problem whether these two problems are actually equivalent (for example, whether the existence of a polynomial-time isomorphism algorithm would yield the existence of a polynomial-time computable canonical form). However, typically for graph classes for which we know a polynomial-time isomorphism algorithm, we also have a polynomial-time canonization algorithm. Sometimes, the extension from isomorphism testing to canonization is straightforward; sometimes it requires extra work (for example, Babai, Babai and Luks, Schweitzer and Wiebking).

**Combinatorial Algorithms**

To establish that two graphs are isomorphic, we can try to find an isomorphism. To establish that the graphs are nonisomorphic, we can try to find a “certificate” of nonisomorphism. For example, we can count vertices, edges, and triangles in both graphs; if any of these counts differ, the graphs are nonisomorphic. Or we can look at the degrees of the vertices. If there is some $d$ such that the two graphs have a different number of vertices of degree $d$, the graphs are nonisomorphic.

The **Weisfeiler-Leman algorithm** provides a systematic approach to generate such certificates of nonisomorphism in an efficient way. Actually, it is a whole family of algorithms, parameterized by a positive integer, the *dimension*. The refinement is repeated until the coloring is stable, then the stable coloring is returned.

**Color refinement.** We start by describing the 1-dimensional version, which is commonly known as *color refinement* or *naive vertex classification*. It is one of the most basic ideas in graph isomorphism testing that has been reinvented several times; the oldest published version that we are aware of can be found in Morgan. Color refinement is an important subroutine of almost all practical graph isomorphism tools, and it is also a building block for many theoretical results.

The color refinement algorithm, displayed in Figure 3, iteratively computes a coloring of the vertices of a graph. The actual colors used are irrelevant, what matters is the partition of the vertices into color classes. The final coloring has the property that any two vertices of the same color have the same number of neighbors in each color class. Figure 4 shows an example.

The coloring computed by the algorithm is *isomorphism invariant*, which means that if we run it on two isomorphic graphs, the resulting colored graphs will still be isomorphic and in particular have the same numbers of nodes of each color. Thus, if we run the algorithm on two graphs and find that they have distinct numbers of vertices of some color, we have produced a certificate of nonisomorphism. If this is the case, we say that color refinement distinguishes the two graphs.

Unfortunately, color refinement does not distinguish all nonisomorphic graphs. Figure 5 shows a simple example. But, remarkably, color refinement does distinguish almost all graphs, in a precise probabilistic sense. This, together with its efficiency, is what makes color refinement so useful as a subroutine of practical isomorphism tools.

The reader may have noticed that color refinement is very similar to other partitioning algorithms, in particular the standard algorithm for minimizing deterministic finite automata. Borrowing ideas from Hopcroft’s DFA minimization algorithm, color refinement can be implemented to run in time $O(nm\log n)$, where $n$ is the number of vertices and $m$ is the number of edges of the input graph. Thus, color refinement is indeed very efficient.

**Weisfeiler-Leman.** We have seen that color refinement is not a complete isomorphism test: it fails to distinguish extremely simple nonisomorphic graphs such as those shown in Figure 5. The $k$-dimensional Weisfeiler-Leman algorithm ($k$-WL) is based on the same iterative-refinement idea as color refinement, but is significantly more powerful. Instead of vertices, $k$-WL colors $k$-tuples of vertices of a graph. Initially, each $k$-tuple is “colored” by the isomorphism type of the subgraph it induces. Then in the refinement rounds, the color information is propagated between “adjacent” tuples that only differ in one coordinate (details can be found in Cai et al.). The 2-dimensional version of the algorithm is due to Weisfeiler and Leman; the generalization to higher dimensions is due to Faradžev, Zemlyachenko, Babai, and Mathon (see Cai et al.). If implemented using similar ideas as for color refinement, $k$-WL runs in time $O(n^{k+1}\log n)$.

Higher-dimensional WL is very powerful. Indeed, it is highly nontrivial to find nonisomorphic graphs that are not distinguished by 3-WL. It took a
now seminal paper, by Cai et al., to prove that for every $k$, there are nonisomorphic graphs $G_k, H_k$ that are not distinguished by $k$-WL. Indeed, these graphs, known as the CFI graphs, have size $O(k)$ and are 3-regular.

It turns out that many natural graph classes do not admit the CFI graph construction and a low-dimensional WL is a complete isomorphism test. In particular, for all graph classes $\mathcal{C}$ that exclude some fixed graph as a minor, there is a constant $k$ such that $k$-WL distinguishes all nonisomorphic graphs in $\mathcal{C}$. This includes the class of planar graphs, for which 3-WL suffices.

The Weisfeiler-Leman algorithm is remarkably robust. It not only subsumes most combinatorial ideas for graph isomorphism testing but also has a natural characterization in terms of logic. Surprisingly, it also correlates graph isomorphism testing but also assumes most combinatorial ideas for the latter. Starting with a suitable group of permutations, we want to compute within it the automorphism group $A$ of interest (technically we want to compute a certain set-stabilizer or the solution to a string isomorphism problem, on which we will not elaborate here). We continually maintain an encasing group $\Gamma \supseteq A$ containing all automorphisms as a subgroup. Our strategy is to iteratively shrink $\Gamma$ until it agrees with $A$.

To shrink the group $\Gamma$, in case the permutation group $\Gamma$ has more than one orbit (see Figure 6), we process orbits sequentially. If the group has only one orbit, we exploit so-called blocks whenever they exist. A block of a permutation group $\Gamma \leq \text{Sym}(\Omega)$ is a subset always mapped to itself or somewhere else entirely, that is, a set $B \subseteq \Omega$ of the permutation domain $\Omega$ such that for all $\gamma \in \Gamma$ we have $\gamma(B) = B$ or $\gamma(B) \cap B = \emptyset$. The set of images of the block $\{ \gamma(B) \mid \gamma \in \Gamma \}$ partitions the domain $\Omega$ into blocks of equal size, which together form a so-called block system. The group $\Gamma$ permutes the blocks of the system and we can consider the induced permutation group $\Gamma'$ on the blocks.

As GI and the automorphism group problem AUT are polynomially equivalent (see Mathon), it suffices to solve the latter. Starting with a suitable group of permutations, we want to compute within it the automorphism group $A$ of interest (technically we want to compute a certain set-stabilizer or the solution to a string isomorphism problem, on which we will not elaborate here). We continually maintain an encasing group $\Gamma \supseteq A$ containing all automorphisms as a subgroup. Our strategy is to iteratively shrink $\Gamma$ until it agrees with $A$.

For general graphs, the bottleneck of this procedure occurs when $\Gamma$ is large. In that case, $\Gamma'$ is a large primitive group. Such groups are called Cameron groups and a precise classification is known. However, this is not a new insight and the fact that Cameron groups form the bottleneck to improving Luks’s method was already known in the 1980s.

**Babai’s Quasipolynomial-Time Algorithm**

Attacking exactly this bottleneck, 35 years later, it was Babai who improved the running time of the theoretically fastest general graph isomorphism algorithm. He showed that graph isomorphism can be solved in quasipolynomial time $n^{O((\log n)^c)}$, that is, $n^{\log^{\log^c}(n)}$ for some constant $c$. Doing his algorithmic ideas justice is difficult not only because they span 80 pages in his original manuscript but also because the algorithm contains several major, very distinct new ideas that combine smoothly to an overall algorithm. Here, we can...
thus only sketch the underlying ideas of the various puzzle pieces and how they are combined.

The first step, at quasipolynomial cost, is to reduce the bottleneck of Cameron groups further to what Babai calls Johnson groups. They are groups abstracly isomorphic to a symmetric or an alternating group, but do not necessarily act in their natural action of permuting elements in some ground set, and rather consist of permutations of the \( t \)-element subsets of the ground set. An example is the automorphism group of the Petersen graph (see Figure 7).

As next step, to make Luks’s framework more flexible for his recursive algorithm, Babai does not only maintain an encasing group \( \Gamma \) containing the automorphism group, but also a homomorphism \( \varphi: \Gamma \rightarrow \Gamma' \) into a permutation group \( \Gamma' \leq \text{Sym}(\Omega') \) over an ideal domain \( \Omega' \). This allows the algorithm to make progress by decreasing the size of the ideal domain.

Initially, for Johnson groups, we can choose as ideal domain the abstract ground set mentioned here. This way, the image of \( \varphi \) contains almost all permutations, that is, it is the symmetric group or the alternating group on the ideal domain. These two groups are by far the largest primitive groups and therefore called the giants. Accordingly, we speak of a giant homomorphism.

The general strategy of the algorithm is to reduce a problem instance to quasipolynomially many instances that are all smaller than the original instance by at least a constant factor. This is continued until the recursive instances are sufficiently small to be resolved with brute force, leading to an overall quasipolynomial-time algorithm.

If the permutation group induced by the homomorphism \( \varphi \) on the ideal domain is intransitive or imprimitive, we can use the strategies of Luks to process orbits sequentially or to consider the actions on blocks, respectively. For this, some nontrivial group theory is required to pull back information from the ideal domain to the original domain.

In case the subgroup \( A \) of \( \Gamma \) that we are interested in maps onto the alternating group of the ideal domain, the computation of \( A \) is comparatively easy, so let us focus on the case that the image of the subgroup is not a giant.

We then use a local to global approach. We first collect local certificates by testing, for all logarithmic-size subsets \( T \) of the ideal domain, whether the homomorphism \( \varphi \) applied to the sought-after automorphism group \( A \) and restricted to \( T \) is a giant homomorphism. We call these sets test sets. A test set is full if the said restriction is a giant homomorphism. As the test set size is only logarithmic and there are only quasipolynomially many such test sets, we can test all test sets for fullness using recursion.

If a test set is full, which certifies high local symmetry, there must be global symmetries certifying this and quite surprisingly such global symmetries can be efficiently constructed. At the core of this statement lies the Unaffected Stabiliser Lemma, a central insight proven by Babai.

If there are a lot of full test sets, the global symmetries allow for efficient recursion. On the other hand, if only few test sets are full, the graph must have a nontrivial structural invariant. Furthermore, we can use the logarithmic-dimensional Weisfeiler-Leman algorithm to construct such a structural invariant in the form of a relational structure of logarithmic arity. This breaks the apparent symmetry.

With the design lemma, we can reduce the relational structure of logarithmic arity to a structure with a binary relation. We obtain a uniprimitive coherent configuration, a particular structure important in algebraic graph theory closely related to the 2-dimensional Weisfeiler-Leman algorithm.

The final puzzle piece is the Split-Or-Johnson combinatorial partitioning algorithm which, from a uniprimitive coherent configuration either produces a split or finds a large canonically embedded Johnson graph, a graph whose automorphism group is a Johnson group. In fact splits, which are invariant partitions of the ideal domain akin to the blocks of a permutation group, can also occur during other parts of the algorithm. They are handled with the techniques similar to the imprimitive case of Luks’s algorithm.

We are left with the case in which a large canonically embedded Johnson graph has been produced. After all, this case had to occur at some point because we know that the resilient Johnson groups exist. But now the Johnson graph is in fact a blessing because we can exploit the well-understood structure of the Johnson graphs to dramatically decrease the size of the ideal domain.

Overall we obtain a quasipolynomial-time algorithm solving the general graph isomorphism problem. Besides the original manuscript, there is also a detailed explanation of the algorithm in the Bourbaki series by Helfgott (see Helfgott et al. for an English translation). In fact, Helfgott detected an error in the Split-or-Johnson routine which however was quickly fixed by Babai.

Babai’s algorithm depends on the classification of the finite simple groups, an enormous theorem spanning several hundred journal articles written by numerous authors. Many group theorists prefer to avoid the theorem and indeed Pyber modified Babai’s algorithm to give an alternative analysis that does not depend on the classification.

In further advances, Babai recently extended his result to a canonization algorithm that runs in quasipolynomial time, and there is an improvement on Luks’s original result for graphs of maximum degree \( d \) testing isomorphism in time \( n^{(\log d)\cdot \log(\log d)} \).

**Practical Graph Isomorphism**

In practice it is excessive to even run the 2-dimensional Weisfeiler-Leman algorithm, let alone some version of increasing dimension as in Babai’s algorithm. Current isomorphism packages rather use color refinement, that is, the 1-dimensional version. As mentioned, this is already sufficient for almost all graphs. If it turns out not to be sufficient, the algorithms take the route of branching by using the concept of individualization.

Specifically, the individualization-refinement paradigm, which is adopted by virtually all modern competitive isomorphism tools, one by one artificially assigns a different color to all the vertices in a color class. This breaks the symmetry and subsequently color refinement can be potentially applied again to produce a more refined partition of the vertices. In a backtracking manner, the tools continue until a discrete color (a coloring in which all color classes are a singleton) has been reached. The tools use various pruning techniques, such as invariants and pruning with
automorphisms, discovered with intricate methods, to drastically improve their performance.

The tools actually compute a canonical form, which also solves the isomorphism problem (as explained earlier). This highly practical method was originally pioneered by McKay with his famous software tool Nauty. There are now various extremely efficient packages such as Bliss, Conuato, Nauty, Saucy, and Traces freely available. Recently many new ideas, responsible for their efficiency, such as the use of the trace for early abortion of color refinement in Traces, have found their way into the tools. We refer the reader to an extensive survey. In contrast to Babai’s quasipolynomial-time algorithm, there are, however, graphs on which the running time of all individualization-refinement algorithms scale exponentially.

Applications

Graph isomorphism tools can in practice be used to find symmetries of combinatorial objects and as such they have numerous applications in miscellaneous domains. In the context of optimization, for example, in SAT solving, symmetries are exploited to collapse the search space, as parts equivalent under symmetries only need to be explored once. An alternative way of exploiting symmetries is to add symmetry breaking constraints to the original input again drastically improving performance.

Another application domain exploits canonical labeling to store graph structured data in a database. For example, when molecules are stored in a chemical database, the idea is to store only a canonical representative. To look up a given molecule in the database, we compute its canonical representative and find the result in the database. This way, no isomorphism tests against the elements in the database are required. Other application domains include machine learning, computer graphics, software verification, model checking, and mathematical programming.

Concluding Remarks

With Babai’s quasipolynomial-time algorithm, we have seen a breakthrough on one of the oldest and best studied algorithmic problems. Undoubtedly, this algorithm and its underlying mathematical framework rank among the most important contributions to theoretical computer science in a long time. We are only starting to explore the potential of the wealth of new ideas they bring to the field.

Current challenges include the group isomorphism problem, one of the core obstacles to even faster graph isomorphism tests. On the practical side, emerging applications in areas such as machine learning demand a better understanding of approximate versions of isomorphism and similarity measures between graphs.

Yet the question whether graph isomorphism is solvable in polynomial time remains open, and we can expect further deep, exciting insights until it will finally be settled.

References

1. Alon, N., Friedman, J. Combinatorial, geometric, and graph-theoretic methods for testing group isomorphism. J. ACM 38, 1 (1991), 214–296.
2. Babai, L. Graph isomorphism in quasipolynomial time. In Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC’19), 2019, pp. 1–99.
3. Babai, L., Luks, E.M. Permutation groups and polynomial-time isomorphism test for graphs of bounded valence. In Proceedings of the 20th Annual ACM Symposium on Theory of Computing (1988), pp. 172–177.
4. Babai, L. Canonical form for graphs in quasipolynomial time. In Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (2019).
5. Babai, L., Kantor, W.M., Luks, E.M. Computations in the symmetric group. In Proceedings of the 31st Annual ACM Symposium on Theory of Computing (1999), pp. 202–210.
6. Babai, L., Kantor, W.M., Luks, E.M. Local isomorphism of nonlinear structures. In Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (2000), pp. 299–307.
7. Babai, L., Luks, E. Canonical labeling of combinatorial objects. In Proceedings of the 1983 Conference on Computational Complexity, pp. 17–25.
8. Babai, L., Maletti, A. Two approaches to graph isomorphism. In Proceedings of the 1984 Conference on Computational Complexity, pp. 12–20.
9. Babai, L., Luks, E.M. Isomorphism of permutation groups. In Proceedings of the 1985 Conference on Computational Complexity, pp. 13–29.
10. Babai, L., Luks, E.M. Group isomorphism in quasipolynomial time. In Proceedings of the 51st Annual ACM Symposium on Theory of Computing (2019), pp. 1–99.
11. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
12. Babai, L., Luks, E.M. A new approach to the graph isomorphism problem. In Proceedings of the 1985 Conference on Computational Complexity, pp. 2–19.
13. Babai, L., Luks, E.M. Isomorphism of graphs of bounded valence can be tested in polynomial time. J. Comput. Syst. Sci. 30, 1 (1985), 1–25.
14. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
15. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
16. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
17. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
18. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
19. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
20. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
21. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
22. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
23. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
24. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
25. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
26. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
27. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
28. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
29. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
30. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
31. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
32. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
33. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
34. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
35. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
36. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
37. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.
38. Babai, L., Luks, E.M. Canonical labeling of graphs. In Proceedings of the 1984 Conference on Computational Complexity, pp. 1–25.