University of Alberta

The Graph Isomorphism Problem

by

Scott Fortin

Technical Report TR 96–20
July 1996

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Abstract

The graph isomorphism problem can be easily stated: check to see if two graphs that look differently are actually the same. The problem occupies a rare position in the world of complexity theory, it is clearly in NP but is not known to be in P and it is not known to be NP-complete. Many sub-disciplines of mathematics, such as topology theory and group theory, can be brought to bear on the problem, and yet only for special classes of graphs have polynomial-time algorithms been discovered. Incongruently, this problem seems very easy in practice. It is almost always trivial to check two random graphs for isomorphism, and fast hardware implementations exists for application domains such as image processing. This paper is mostly a survey of related work in the graph isomorphism field. We examine the problem from many angles, mirroring the multi-faceted nature of the literature. We survey complexity results for the graph isomorphism problem, and discuss some of the classes of graphs which have known polynomial-time algorithms. Many of these algorithms have constants that are so large as to negate their usefulness in actual applications. Thus a number of practical techniques, which do not have a polynomial worst case, are also examined. We end the paper by discussing some graphs which are "hard" for practical algorithms, and present experimental results which demonstrate a class of graphs on which practical algorithms run into difficulty.

1 Introduction

The graph isomorphism problem has a long history in the fields of mathematics, chemistry, and computing science. Much research has been devoted to this subject, so much in fact that in 1977 Read and Corneil christened it "The Graph Isomorphism Disease" [29]. Work has continued unabated on the graph isomorphism problem however, due to the many practical applications of the problem, and its unique complexity properties.

The graph isomorphism problem can be simply stated: given two graphs, does there exist a 1-to-1 mapping of the vertices in one graph to the vertices of the other such that adjacency is preserved? In symbols, given G_1 and G_2, does there exist an f such that $\forall a, b \in V_1,\ (a, b) \in E_1 \iff (f(a), f(b)) \in E_2$. The graphs in Figure 1 are examples of isomorphic graphs.

As a first attempt at understanding graph isomorphism, we may wish to know in which complexity class the problem lies. Somewhat surprisingly, the problem is known to be in NP, but is not known to be in P or NP-complete. Factoring is another problem which is not thought to be in P or NP-complete, however factoring is known to be in co-NP, which is not the case with graph isomorphism. This ambiguity, about the exact location of graph isomorphism within the conventional complexity classes, has led to many research papers; in Section 2 we survey some of these results.

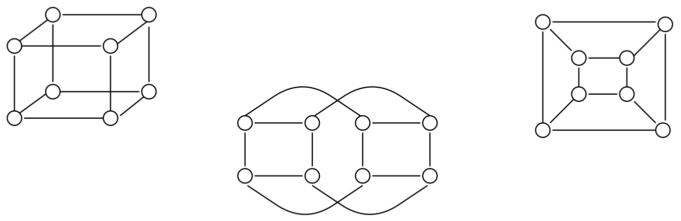


Figure 1: Isomorphic graphs

While the position of graph isomorphism in complexity analysis is more than enough to justify its study, it turns out that there are many practical applications which desire a fast isomorphism algorithm. For example, organic chemists routinely deal with graphs which represent molecular links, and would like some system to quickly give each graph a unique name. Thus many papers on graph isomorphism discuss how to build fast, practical, isomorphism checkers. Some of these routines come from special polynomial-time algorithms for a restricted class of graphs, but most practical algorithms do not have a polynomial time worst case. In Section 3 we examine in detail some of the practical results.

Although no practical algorithm which can be applied to all graphs has a provably polynomial worst case, the significance of this fact is not entirely clear. Finding graphs which are "hard" for the strongest practical algorithms is a non-trivial task. Graphs which are randomly generated, by making each pair of vertices an edge with fixed probability p, are almost always easy to solve by a simple backtrack routine with a very naive heuristic [16]. In Section 4 we discuss the problem of finding hard graphs for the isomorphism problem, and give experimental results which demonstrate a class of difficult graphs.

For the most part, we try to hide the reader from the complicated group theory that underlies many graph isomorphism results. However, some group theoretic concepts cannot be avoided, and we give a brief summary here. An *automorphism* of a graph G can be defined as an isomorphism between two copies of G; i.e. it is a mapping f between pairs of vertices such that $(a, b) \in E \iff (f(a), f(b)) \in E$. A *trivial automorphism* is an automorphism where $f(a) = a$ for all $a \in V$. An *automorphism partition* of a graph G is a sequence of disjoint subsets $V_1, \ldots, V_k$, such that for all pairs of vertices $a, b \in V_i$ there exists an automorphism where $f(a) = b$[1]. Thus the automorphism partition divides V into sets consisting of all the vertices that can be mapped onto one another.

[1]Alternatively, we could say that V_i is the orbit of a in the automorphism group of G.

2 Complexity Results

The graph isomorphism problem (GI) occupies an important position in the world of complexity analysis. It is one of the few problems which is in NP but is not known to be in either P or NP-complete; further, GI is not known to be in co-NP. It is well known that if P $\neq$ NP then there exists problems which are of intermediate status, and it has been hypothesized that GI may be one of these problems.

One early result on the complexity of GI showed that there is a moderately exponential algorithm for solving the problem [3]. Moderately exponential means that on a problem of size n, the running time t obeys the relation: $n^k < t < a^n$, where $k, a > 1$ are arbitrary real numbers. Typically this means the exponent in the running time is a function of n which grows slower than n. For example, the one upper bound for general graph isomorphism is $\exp(n^{1/2+o(1)})$ [24].

Babia, Erdos, and Selkows showed that for almost all graphs X, isomorphism to any other graph Y could be computed in quadratic time [5]. As well, it has been shown that there is a linear expected time algorithm for solving graph isomorphism [6]. These results show that under a metric applied to other graph theory problems, namely random graphs, the isomorphism problem is not very difficult.

Since GI has resisted efforts to classify it in either P or NP-complete, researchers have taken other approaches to determining its complexity. One approach is to define it as being its own complexity class, *isomorphism-complete*, and see which other problems fall within its class. Another approach is to generalize the notion of NP into more esoteric notions of complexity, and study the location of GI in these new complexity classes. A third approach is to try to determine classes of graphs such that the isomorphism problem is in P when restricted to graphs of that type.

2.1 Isomorphism Complete

Mathon [25] showed that a number of problems are polynomially equivalent to GI. Most significantly, he showed that the problem of counting the number of isomorphisms between two graphs is isomorphism-complete. This result is further evidence that GI is not NP-complete, since it seems that the counting versions of NP-complete problems are much harder than their related decision problems. Indeed even for some problems in P, the counting version is (conjectured to be) much harder than the decision problem.

Mathon also demonstrated the close ties between GI and automorphism problems. He showed that GI is equivalent to the problem of counting the number of non-trivial automorphisms of a graph, which is in turn equivalent to the problem of determining the automorphism partition of a graph. The graph automorphism decision problem (GA), which asks if there exists a non-trivial automorphism, is an interesting exception; while GA is reducible to GI, no such reduction in the other direction is known to exist.

Many restricted isomorphism problems are also known to be polynomially equivalent to GI. We simply list, without proof, that the graph isomorphism problem confined to any of the following types of graphs is isomorphism-complete: bipartite graphs, line graphs, rooted acyclic digraphs [2], chordal graphs, transitively orientable graphs [10], and regular graphs [9]. These results are not terribly surprising since many of these graph classes exhibit a large amount of regularity, which is one of the factors that makes graph isomorphism difficult (see Section 4).

The term equality problem was also shown to be isomorphism-complete by Basin [7]. In this problem, we are given two terms which can have functions which are associative, commutative, or both, and can have commutative variable binding operators. We then must decide if the two terms are equal; a special case of this problem is deciding if two predicate calculus terms are equal. This problem is different from the previous isomorphism-complete problems, in that it does not occur naturally as a graph theoretic problem. It lends credence to the notion of isomorphism-complete as a separate and useful complexity class.

2.2 Generalizations of NP

NP is typically characterized as those problems where there is a polynomially checkable proof certificate. It can also be thought of as an interaction between two participants, an all-powerful prover and a verifier. The prover looks at the input, and computes a proof certificate which he sends to the verifier; the verifier then checks the certificate in polynomial time and either accepts or rejects the input. P can be thought of as those problems where there is no communication between the prover and the verifier.

The class IP is a generalization of the above scenario. Instead of no messages (for P) or 1 message (for NP), we allow a polynomial number of messages between the prover and the verifier. As well, we let the prover and verifier have access to a stream of private random numbers and only require that over all inputs the verifier accepts positive instances with probability $\geq 3/4$, and accepts negative instances with probability $\leq 1/4$. IP is known to be equal to PSPACE, and is thus not very interesting. However if we restrict the number of messages to a constant k, we get the class IP(k), which is known to be equal to IP(2) for all $k > 1$ and is not known (and believed not) to be equal to PSPACE. In this notation we thus have three classes which may not be equal, IP(0), IP(1), and IP(2). Note that P and NP are special cases of IP(0) and IP(1) respectively, where the probability of accepting postive instances is 1 and the probability of accepting negative instances 0.

Coming back to graph isomorphism, we have that since GI is in NP, GI is in IP(1) and hence GI is in IP(2). Much more surprising is the result of Goldwasser, Micali, and Rackoff [19] that the graph non-isomorphism problem (which is not known to be in NP) is in IP(2). The algorithm is quite simple, and demonstrates the basic idea of an interactive proof:

proof that $\overline{\mathbf{GI}} \in \mathbf{IP}(2)$

```
1. let the input graphs be $G_1$ and $G_2$
2. the verifier picks (at random) $a, b \in \{1, 2\}$
3. $I \leftarrow$ some isomorphism of $G_a$, $J \leftarrow$ some isomorphism of $G_b$
4. the verifier sends $I, J$ to the prover in parallel
5. the prover returns 2 numbers $i, j$ such that $G_i$ is isomorphic to $I$,
     and $G_j$ is isomorphic to $J$
6. if $a = i$ and $b = j$ the verifier accepts, otherwise it rejects
```

If G_1 and G_2 are not isomorphic, then clearly the verifier will always accept. If they are isomorphic, the prover has no way to know which graph I was derived from, and so with probability 1/2 it will return i such that $i \neq a$. Similarly for graph J, there is probability 1/2 that $j \neq b$. Thus if G_1 and G_2 are isomorphic, the verifier accepts with probability $\leq 1/4$. □

This result, combined with the lack of results placing GI in P or NP-complete, gives a complexity diagram like that of Figure 2. This significance of this classification is not solely the increased perspective it gives on the GI problem, but it has also been used to give strong evidence that GI is not NP-complete.

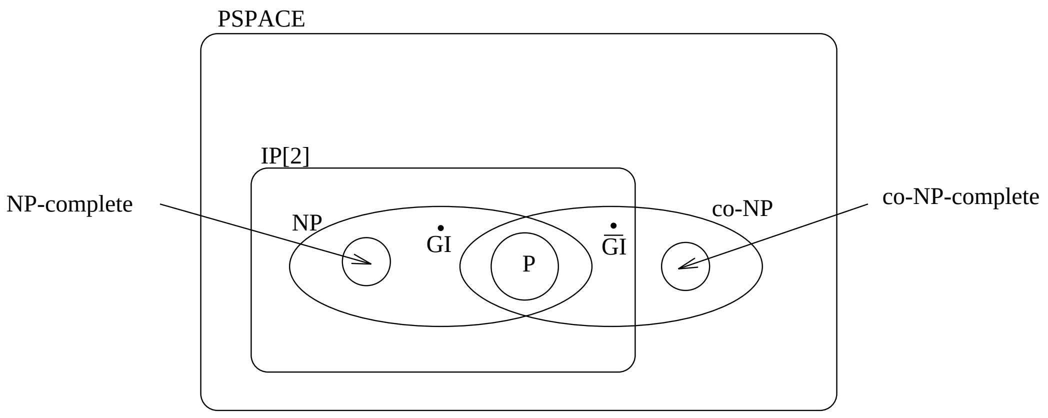


Figure 2: Complexity of GI and $\overline{\text{GI}}$

The polynomial hierarchy (PH) is a hierarchy of complexity classes $PH_0 \subseteq PH_1 \subseteq PH_2 \subseteq \ldots$, based on existential quantifiers, universal quantifiers, and polynomial time algorithms. It too is a generalization of NP, i.e. $PH_0 = \mathrm{P}$ and $PH_1 = \mathrm{NP}$. We will not discuss the details of PH, except to say that it is generally believed to be a proper hierarchy [23]. That is to say, it is believed there does not exist an i such that for $j \geq i$, $\mathrm{P}H_i = \mathrm{P}H_j$. Boppana, Hastad and Zachos [11] and Schoning [31] prove (in different ways) this main result of graph isomorphism structural complexity:

> If the graph isomorphism problem is NP-complete, the polynomial hierarchy collapses to $PH_2 = \mathrm{IP}(2)$

2.3 Graphs where GI is in P

The first major result in this field was a paper by Luks in 1978 [24]. He showed that for graphs which have bounded degree (i.e. the maximum degree $\leq$ some constant k), there exists a polynomial time algorithm to check isomorphism. In the paper, GI is reduced to the problem of finding the number of automorphisms in a graph with a special labeled edge. A high level description of the algorithm proceeds in three steps. First an edge e is chosen, and the graph is partitioned into $X_1, \ldots, X_{n-1}$ where X_i is the induced subgraph on those vertices which are i steps or fewer away from e. Next, in every X_i, each vertex is mapped to its set of neighbours in X_{i+1}, and vertices with the same mapping are grouped together. Finally each of these groups is decomposed into primitive groups where the number of automorphisms is known, and these numbers are summed for all groups. Using powerful ideas from group theory, Luks proved that each step is always possible and runs in polynomial time. The algorithm presented in the paper was not meant to give a tight upper bound on the running time, and Luks claims that using more sophisticated techniques the running time is in $\mathrm{O}(n^{ck \log k})$, where $c > 1$ is a constant and k is the maximum degree. It is clear that even if the maximum degree is only 10, this technique does not yield a practical algorithm.

Other graph properties have been studied with respect to graph isomorphism, one such property is the genus of a graph. If we wish to draw (more commonly *embed*) a graph on some surface such that no two lines cross, we may need a surface with a fairly complicated structure. The genus is a measure of how complicated such a surface must be; and the genus distribution of a graph is defined as a sequence of numbers $g_1, g_2, \ldots$ such that g_i is the number of ways to draw a graph on a surface of genus i. Filotti and Mayer [18] show that if we fix k, then those graphs which have $g_k > 0$ can be checked for isomorphism in polynomial time. This algorithm is not very practical, as its running time is $\mathrm{O}(n^{ck})$ where $c > 300$ is a constant. Chen obtains a better result [13], showing that if you take the average of the g_i's, and require that to be bounded by a constant k, then there is a linear time isomorphism test. However, the hidden constant in this algorithm is $(512k^3)!$ which is very large no matter what the value of k . Planar graphs are in some sense a special case of this result, as they can be drawn on a plane (or a sphere) and hence have $g_k > 0$ for all $k \geq 1$. It was first shown by Hopcroft and Wong [20] that planar graphs could be checked for isomorphism in linear time, although their algorithm also has a large constant which makes it unsuitable in practice.

Another large class of graphs over which GI can be solved efficiently is the circular-arc graphs [21]. Circular arc graphs are those graphs which can be created by drawing a number of arcs on a circle, creating a vertex for each arc, and joining two vertices if their arcs overlap; a subclass of circular arc graphs is the interval graphs, where our "arcs" are confined to the real number line. Hsu [21] has shown that there is a $O(mn)$ algorithm for testing isomorphism of these graphs, where m is the number of edges, and n is the number of vertices. This result is interesting in that it does not require some explicit parameter to be constant, and seems to apply to a large and practical group of graphs.

There are several other restricted classes of graphs for which graph isomorphism can be solved in polynomial time. For example, Cogis and Guinaldo [14] show that isomorphism for conceptual graphs which have a linear ordering function can be solved in polynomial time. This is particularly of interest to the artificial intelligence community, where conceptual graphs are used as a technique for knowledge representation. Other graph classes such as trees [2], permutation graphs [15], chordal (6,3) graphs [4] and partial k-trees [8] all have isomorphism problems which are in P.

These results taken together provide an interesting counterpoint to the discussion of random graphs. It seems that the isomorphism problem is easy for graphs which are not related in any way (random graphs) and for graphs which are structurally related (circular arc graphs for example). This notion will be explored more fully in Section 4.

3 Practical Isomorphism Algorithms

One of the factors contributing to the large amount of work on the graph isomorphism problem is undoubtably the many practical applications of the technique. There are two broad approaches to solving the isomorphism problem. The first approach is to take the two graphs which are to be compared, and try to directly find an isomorphism between them. This has the advantage that if there are many isomorphisms between two graphs, then only the first one need be considered.

The second approach is to take a single graph G and compute some function $C(G)$ which returns a *canonical label*; where a canonical label means that $C(G) = C(H)$ iff G and H are isomorphic. Such a labeling not only solves the isomorphism problem, but is possibly of independent interest. For example, organic chemists typically deal with "graphs" such as the one if Figure 3. It is a fundamental problem in the world of chemical documentation to give each such structure a unique name.

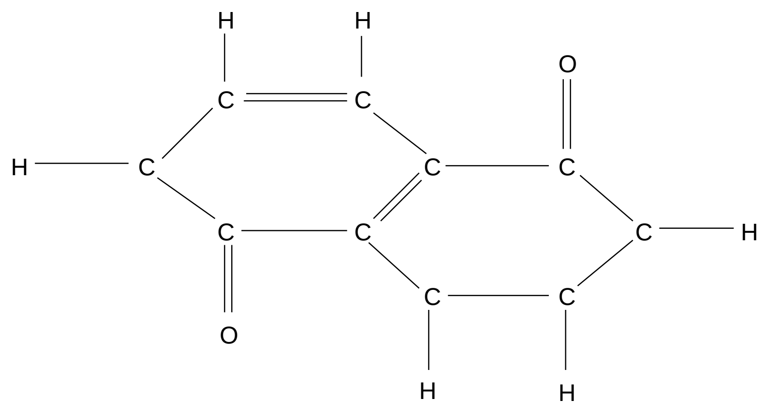


Figure 3: A typical organic chemistry graph

It is not hard to describe a naive canonical labeling system. Suppose that we represent graphs by an adjacency matrix M, where $M_{ij} = 1$ if (i, j) is an edge, and 0 otherwise. We can construct a label for G by concatenating the rows of M to form a $N \times N$ binary number. We can then compute a canonical label for G by looking at the label (as defined above) of every automorphism of G, and returning the smallest one. Of course finding every automorphism of G is isomorphism-complete, so this method has to be slightly modified to be used in practice.

The rest of this section is organized as follows. We first briefly look at non-traditional and special case approaches to graph isomorphism. We then look at vertex invariants which are an important practical tool in solving isomorphism problems, and finally we look at algorithms for solving the graph isomorphism problem using each of the two approaches outlined above.

3.1 Non-traditional Approaches to GI

There has been some work on non-traditional approaches to the graph isomorphism. McGregor [26] discusses using arc and path consistency algorithms on the isomorphism problem. The algorithm in the paper converts a graph isomorphism problem into a constraint satisfaction problem, and then uses specially tuned constraint algorithms. This approach does not seem intuitively inviting, as the constraint satisfaction problem is known to be NP-hard, and the graph isomorphism problem could very possibly be in P. The results of the paper seem to fit the intuitive notions, where only a slight increase was observed over previous (relatively simple) backtracking algorithms.

Agusa, Fujita, Yamashita, and Ae [1] present a Hopfield network for solving isomorphism problems. It is unclear to what problem domain this type of solution is suited. Theoretically, this approach cannot prove a polynomial worst case, since it is not possible to guarantee that it will always return the correct result. In practice, we may be willing to live with a small degree of uncertainty, however this approach certainly cannot be used unmodified in a practical setting. The reason for this is the fact that the model restricts the maximum number of vertices allowed in a graph to 12. Clearly this paper proves that neural network solutions to the isomorphism problems (for small graphs) do exist, although it seems to do little to advance the theory or practice of graph isomorphism.

Image processing ranks with chemistry as a large application domain for isomorphism checking. The graphs which arise in this application domain are planar graphs which represent the interrelationships between objects in a scene. As we discussed in Section 2, Hopcroft and Wong showed planar isomorphism checking can be solved in linear time [20]. JaJa and Kosaraju extend this result by giving a fast, parallel planar isomorphism algorithm [22]. Given the real time constraints of image processing, this algorithm was taken one step further by Demarchi, Masera, and Piccinin [17], who implemented planar isomorphism checking in a VLSI processor array. While these results have little bearing on the isomorphism problem as a whole, they do show that some practical implementations have become very efficient.

3.2 Vertex Invariants

One very practical technique, which is independent of the algorithm used to actually solve the isomorphism problem, is the use of vertex invariants. A vertex invariant is some number $i(v)$ assigned to a vertex such that if there is some isomorphism that maps v onto v' then $i(v) = i(v')$. The typical example of a vertex invariant is the degree of a vertex: if v and v' map to each other under some isomorphism, then certainly they must have the same degree. Note that the converse does not hold, if two vertices have the same degree, then there is not necessarily an isomorphism which maps the two vertices onto one another. In fact, if there is a polynomially computable vertex invariant for which the converse does hold, then there is a polynomial time solution to the isomorphism problem.

There are many vertex invariants which have been proposed in the literature. Below we list some invariants which are present in an isomorphism program called **nauty** [27].

twopaths
: assign a number to v based on the vertices reachable along a path of length 2

adjacency triangles
: assign a number to v based on the size of the common neighbourhood of

those neighbours of v who are adjacent

k-cliques
: assign a number to v based on the number of different cliques of size k that contain v

independent k-sets
: assign a number to v based on the number of different independents sets of size k that contain v

distances
: assign a number to v based on the number of vertices at each distance $1, \ldots, n$ from v

The use of invariants is something of a black art. Corneil and Kirkpatrick [16] show that using vertex invariants may not result in any decrease of the search space. More practically, vertex invariants are typically used by combining the results of many invariants. For example, if a vertex has degree 3, is in 2 cliques of size 4, and can reach 9 vertices on paths of length 2, then a composite invariant might be 329. In fact, this ability to compose invariants has led to the case where practical algorithms always use degree as the "base" invariant, and other variants get appended to the base. These other invariants, while they can be computed in polynomial time[2], can be quite expensive to calculate, compared to the total execution time. It turns out that many isomorphism problems can be directly solved in less time than it takes to calculate the more powerful invariants [27], and it is very difficult to determine which (if any) invariant is the best for a particular graph. Instead, what is typically done is to leave the decision of if and when to use a vertex invariant up to the user (except for degree which is always used). This assumes that if the user happens to come across a number of very hard graphs, he would then take the time to experimentally determine what invariant to use. Another way of justifying this, is that the degree invariant is sufficient for most graphs encountered in practice, and the user only has to intervene if the graph has a very regular structure.

3.3 Solving the Isomorphism Problem Directly

Perhaps the most natural way to try and solve the graph isomorphism problem is to try and directly discover a mapping between the vertices of two graphs G_1 and G_2. The algorithm presented in this section is an adaptation of the one in [30]. We start by defining $G_1(k)$ as the subgraph of G_1 induced on the vertices $1, \ldots, k$. The basic idea is to build an isomorphism from $G_1(k)$ to a subgraph of G_2, and attempt to extend this to an isomorphism on $G_1(k+1)$ by adding an unused vertex of G_2.

[2]Actually invariants have been proposed which do not run in polynomially time, but they are clearly not of practical use

The main routine of the algorithm looks something like:

```
**Input:** graphs $G_1$ and $G_2$
**Output:** array $f$ such that $i \rightarrow f_i$ is an isomorphism between $G_1$ and $G_2$,
    or NONE if no such $f$ exists
**Procedure Main:**
    for $i \leftarrow 1, ..., n$ do
        $inv1_i \leftarrow$ some vertex invariant on the $i$-th vertex of $G_1$
        $inv2_i \leftarrow$ some vertex invariant on the $i$-th vertex of $G_2$
    endfor
    if $inv1$ sorted in ascending order $\neq inv2$ sorted in ascending order
        return NONE
    reorder the vertices of $G_1$ (and reorder $inv1$ as well)
    if ISOMORPH(emptySet,1,$f$) then
        return $f$
    else
        return NONE
```

We can see that this is one of the places in the algorithm where heuristics are used. The calculation of $inv1$ and $inv2$ allows us to reject many non-isomorphic graphs, providing our chosen vertex invariant is powerful enough. The second optimization is to reorder the vertices of G_1, with the hope that there will not be many vertices in G_2 that can map to the first few vertices G_1. Several ordering methods may be used, for example the vertices in G_1 may be simply sorted based on the number of vertices which share a common invariant value (the invariant multiplicity), or perhaps a vertex with minimum invariant multiplicity is chosen and a depth-first search rooted at that vertex is used to re-order the vertices. The simple sort ensures us that as we enlarge the isomorphism there is a minimum number of candidates to extend the solution at each step. The depth first search approach is attractive in that adjacent vertices are placed near one another, and thus we would hope that if we making an incorrect mapping decision, we would quickly get a contradiction.

The ISOMORPH procedure takes a set S, and number k, and an array f as input. It returns true if the isomorphism between $G_1(k-1)$ and the subgraph of G_2 induced by S can be extended to a complete isomorphism on G_1 and G_2.

```
**Input:** set $S$, integer $k$, and an array $f$
**Output:** TRUE if $f$ can be extended to the entire input graphs
**Procedure ISOMORPH:**
    if $k = n + 1$ then return TRUE
    foreach $j \in (V_2 \ / \ S)$
        if $(inv1_k \neq inv2_j) \vee \neg$CAN_MATCH$(k, j, f)$
            go to next loop iteration
        $f_k \leftarrow j$
        if ISOMORPH$(k + 1, S \cup \{j\})$ then
```

```
            return TRUE
        endfor
        return FALSE
```

It is not too hard to see what this routine is doing. When it is called, there already exists an isomorphism between the first k vertices of G_1 and the vertices in S. If it has already mapped all possible vertices it immediately returns TRUE, and an isomorphism f has been found. Otherwise, it looks at all the unmapped vertices in G_2 and tries to map them to the next vertex in G_1. It uses the vertex invariant heuristic so that it will never try a mapping where the invariant is different. This is the means by which much of the search tree can be pruned, provided we have a good invariant. Note that although this invariant is shown as a cached value on this example, it is possible to dynamically calculate invariants, in which case we could use the partial mapping f to help us create a more discriminating invariant.

The only function left to describe is CAN_MATCH, which returns true if the given mapping would not conflict with previous mappings. This works by simply scanning the isomorphism already discovered (which is contained in f) and checking the (i, k) is an edge of the first graph iff (f_i, j) is an edge in the second one.

3.4 Determining a Canonical Labeling

In this section we present a canonical labeling algorithm, which is actually a high level description of the algorithm presented by McKay called **nauty** [28, 27]. This is a very powerful algorithm, and is currently the preferred method for solving the graph isomorphism problem. The reason this method out performs (in general) the algorithm in the previous section, is that by concentrating on one graph at a time, powerful ideas from the realm of group theory can be brought to bear on the problem, significantly decreasing the running time. As well, this algorithm provides more information than the previous algorithm, listing (for example) all of the automorphisms of a graph.

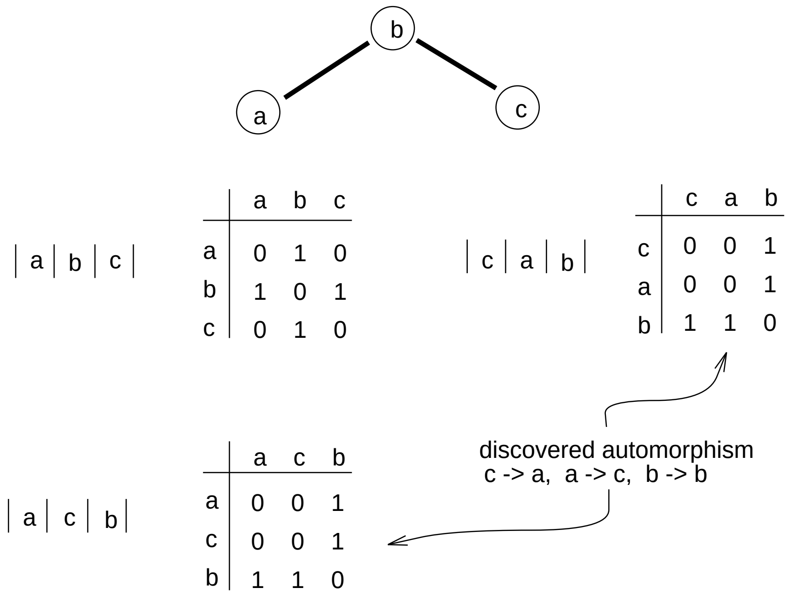
We will not give as detailed an algorithm as that in the previous section, as it would require a large amount of group theory background on the part of the reader. Instead we will concentrate on the main ideas, and refer the interested reader to [28] for more details.

Partitions

Central to the operation of nauty is the idea of an ordered partition. A partition π divides the vertices of a graph into non-empty, disjoint subsets of V (called cells): $V_1, V_2, \ldots, V_m$. A partition with only singleton sets, will be called a *leaf partition*[3]. A key point is that when we find a leaf partition, it defines a

[3] In nauty these are referred to as discrete partitions, however the term leaf partition hilights their role as leaves of the search tree

relabeling of the graph where the vertex in cell V_i gets relabeled to i.



$$\begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \end{array} \qquad \begin{array}{c|ccc} & c & a & b \\ \hline c & 0 & 0 & 1 \\ a & 0 & 0 & 1 \\ b & 1 & 1 & 0 \end{array} \qquad \begin{array}{c|ccc} & a & c & b \\ \hline a & 0 & 0 & 1 \\ c & 0 & 0 & 1 \\ b & 1 & 1 & 0 \end{array}$$

Figure 4: Discovering an automorphism by comparing leaf partitions

For example, consider the graph in Figure 4. Three different leaf partitions are given for the vertices in the graph, and associated with each partition is the adjacent matrix where the order of the vertices in the rows and columns corresponds to the order of the vertices in the partitions. Automorphisms are recognized by finding two distinct leaf partitions which (after relabeling) give rise to the same graph. In the example $|c|a|b|$ and $|a|c|b|$ give the same matrices and thus define an automorphism.

There are two major operations that nauty performs on partitions, refining a partition and generating the children of a partition. Refinement is where vertex invariants can be used. The invariant is computed for each vertex over the entire graph to create an initial partition, and then the invariant restricted to the cells of the partition to try to further distinguish them. For example assume $d(v, S)$ returns the number of neighbours of v that are in the set S. We could then define a refinement operation as:

```
1) $\pi \leftarrow$ the initial partition $V_1, \ldots, V_m$ where for all vertices $x, y \in V_i$
        we have $d(x, V) = d(y, V)$
2) select a $V_i \in \pi$ such that $V_i$ has more than one element
3) for each $v \in V_i$, compute a sequence $a_v = (d(v, V_1), \ldots, d(v, V_m))$
```

4) split V_i into a number of subsets, so the vertices in each subset have the same value for a_v

In actuality, we do not just select a single V_i but starting at V_1 we loop through all the V_i (including ones created after a split) until we cannot split any more cells.

We generate the children of a partition by choosing the first cell V_i which has more than one member, and then for each vertex $v \in V_i$ we create a child partition which is $V_1, \ldots, V_{i-1}, \{v\}, V_i/\{v\}, V_{i+1}, \ldots, V_m$. That is to say, we create a new child for vertex v by splitting the cell V_i into two cells, one which contains only v and one which is V_i without v.

Basic Algorithm

Here is the basic algorithm used by nauty. We have omitted some of the pruning operations, which we will talk about in the next section. It is basically a depth first search of the space of partitions, with the added optimization that we refine each partition before expanding its children.

```
Input: a graph $G$
Output: a canonical graph $C$
    $\pi \leftarrow$ the partition containing a single cell $V$
    $S \leftarrow$ stack containing $\pi$
    while ($S$ is not empty)
        $x \leftarrow$ pop the top of the stack $S$
        if ($x$ is a leaf partition)
            update($C, x$)
        else
            refine($x$)
            append the children of $x$ to $S$
        endif
    endwhile
    return $C$
```

Canonical Labeling

Conceptually, nauty examines every automorphism of a graph and computes a canonical label. This label is simply the adjacency matrix of the "smallest" automorphism. The scheme used to define the smallest automorphism an extension of the scheme we discussed previously, where graphs are ordered based on the binary number that is creating by concatenating the rows of the adjacency matrix. The main addition is an *indicator function* Λ, which takes a partition as input, and computes a hash function in the hope that different automorphism will hash to different values.

Thus if we reach a leaf partition x in our search tree, we define a sequence Λ_x by starting at the root of the search tree and transversing down to x, at each point the result of the indicator function is used to define a member of the sequence. Thus a leaf partition α is defined as smaller than β if $\Lambda_\alpha < \Lambda_\beta$[4], and the equivalent binary number of each automorphism is used to break ties. McKay proves that a canonical labeling defined in this fashion is the same for any two isomorphic graphs [28].

It is easy to see how this information could be used to prune the tree. Suppose we have some partition x which is the smallest partition discovered so far. Then we can compare some node v to x, and if v is bigger we can prune v. By defining the indicator function over all ancestors of x, we may be able to prune nodes which are not leaf nodes, since they may be bigger than the ancestors of x. Note that if we simply order automorphisms based on their binary representation, then we can only prune nodes which are leaf partitions.

Using Automorphisms

There is only one last main technique employed by nauty, the recognition and use of automorphisms. As mentioned earlier, nauty recognizes an automorphism if two different leaf partition result in the same adjacency matrix after relabeling the vertices. Nauty stores two graphs, which it uses to check for automorphisms. One graph corresponds to the "smallest" graph found so far, as defined above, and will eventually become the canonical label; the other graph corresponds to the first leaf node discovered. When a new leaf node is reached, its relabeled graph is compared to the stored copies, and if they have exactly the same adjacency matrix, then an automorphism has been found. At the same time, if nauty discovers the new graph is smaller than smallest so far, then the new graph becomes the new canonical graph. Note that this will not discover all automorphisms, since we are only storing two graphs to compare against. There is a tradeoff — more stored graphs means more matches, but it also means the time to check each graph is increased. In addition to these automorphisms, an automorphism can sometime be inferred by the structure of a partition; however this technique is less important and we do not discuss it further.

Once nauty has found an automorphism, it puts it to work to try to prune the search space. One useful result is that if there is an automorphism between a leaf partition α and some stored partition β, then nauty can jump all the way back to the first common ancestor of α and β and continue from there. Additionally, nauty maintains a special partition θ, where two vertices are in the same cell in θ if nauty has discovered some automorphism which maps the two onto one another. Thus θ is an approximation of the automorphism partition, and gets updated whenever a new automorphism is found. The significance of θ comes into play when we generate the children of some partition π. Recall that this

[4]The two sequences are compared in lexographic order, i.e. in the same manner as English words are ordered

is done by choosing a cell $W \in \pi$, and generating a new child for every element of W. Given θ, we now only create a child for each element of W which is in a unique cell of θ. Thus as we find more and more automorphisms, less and less children get expanded from each node. The proof that these prunings are valid depends on some further ideas about partitions and group theory, which we will not develop here. The interested reader can see [28].

3.5 Discussion

Practical experience seems to indicate that other general paradigms, such as constraint satisfaction and neural networks, are not very useful for graph isomorphism. A fundamental weakness of these approaches is the fact that once the problem is transformed into that paradigm, it is no longer possible to use group theory ideas to help prune the search space. In some sense, the semantics of the data is being lost in the translation.

Another interesting note is the success of a program like nauty, over a more direct approach to isomorphism. The differences here are not as conclusive, but it is not difficult to speculate why nauty does better. Both programs make use of vertex invariants, nauty uses them in its refinement process, and the direct algorithm uses them when deciding whether to extend an isomorphism. The main difference between the two techniques seems to be their use of partial results. Whenever nauty finds an automorphism, it immediately puts the automorphism to work, pruning away sections of the search tree. Thus when nauty explores a portion of the search tree, it always attempts to use these results. Contrast this with the direct scheme. Suppose we have extended the isomorphism to all but one vertex, and then realize that we have to backtrack to a point which is a number of levels back. There is no attempt to use the information gained in the "almost isomorphism" to prune subsequent attempts, and thus we are essentially discarding the knowledge we learned in that portion of the search.

4 Hard Graphs

While the graph isomorphism problem is not known to be in P, one might wonder if this has any practical significance. Put another way, we may wish to know if there are any really "hard" graphs, and if there are, get some handle on what makes them difficult.

The search for hard graphs is not a trivial one. As we have already seen, if we confine ourselves to random graphs, then the isomorphism problem is almost always very easy. If we instead focus on planar graphs, there exists a parallel hardware implementation that will find the isomorphisms of very large graphs in real time. The list of properties that our "hard" graphs cannot have goes on and on, and even if our graph does not have one of those properties, practical algorithms have a dazzling array of heuristics that can be used to quickly solve the isomorphism problem.

Practical algorithms do not get off that easily however. Many of the algorithms that solve a certain class of graphs in polynomial time have constants that make them prohibitive for actual use. Further, even if there is a fast algorithm for a special subclass of graphs, it is often non-trivial to determine if an input graph belongs to such a subclass. Finally, the array of heuristics available to practical algorithms is indeed large, but that also serves to make it non-trivial for the algorithm to decide if and when to apply each one.

In this section we look at the nauty program introduced in the previous section, and attempt to find graphs which are difficult to process. Our approach is not to directly construct these hard graphs, but instead to transform types of "easy" graphs into hard ones. In doing so we gain a better understanding of the power and limitations of practical graph isomorphism.

4.1 Finding Hard Graphs

Our previous discussion of the isomorphism algorithms gives us some hints as to where to look for hard graphs. As we noted, all practical algorithms use degree as a means to prune out obvious non-isomorphisms (or non-automorphisms in nauty's case). Thus the first decision we can make is to restrict ourselves to regular graphs. A graph is *regular* if every vertex has the same degree. We may wish to increase this notion of regularity, by requiring that range in sizes of the common neighbourhood of any two vertices be very small.

Unfortunately, once we make these decisions we reach a practical difficulty; there is no known algorithm to uniformly generate random graphs with the above properties. Instead, we have to try to find "naturally occurring" graphs with our desired regularity properties.

One such source of regular graphs is the p-regular map embeddings of trivalent graphs. These graphs are generated by creating many (conceptually, infinitely many) rings of regular p-gons, and defining a function which maps these polygons onto one another – for example, start at any vertex and map the three polygons which share it to the three polygons that share the vertex that is 14 vertices to the right. After performing this mapping, either the rings of polygons remain infinite (or at least beyond the size that can be handled by the computer's memory), or all polygons are mapped to the single polygon at the center, or there is some finite map of polygons which is produced. The interested reader is referred to [12] for more details on this class of graphs. The later case is of particular interest to us, since it generates a highly symmetric p-regular graph.

Once we have these graphs in hand, we can test how easily programs like nauty can assign them a canonical label. It turns out these graphs are in some sense too regular, the number of automorphisms is very large, which can be exploited to prune the search space. The key idea in making these graphs hard is to decrease the amount of self-similarity by choosing two edges and swapping their endpoints. The edges are chosen at random, but are constrained so that when they are swapped, the degree of each vertex does not change. As the experimental results in the next section show, doing only one of these swaps can make the problem several hundred times harder for nauty.

4.2 Experimental Results

Four graphs were chosen to examine the effect of random swaps on the difficulty of the graph that results. The graphs are entitled **v190**, **v348**, **v486**, and **v864**, where the number in each case represents the number of vertices. All graphs were regular graphs with degree 8, and were created using a generator program written by Joe Culberson [12] based on the construction mentioned in the previous section.

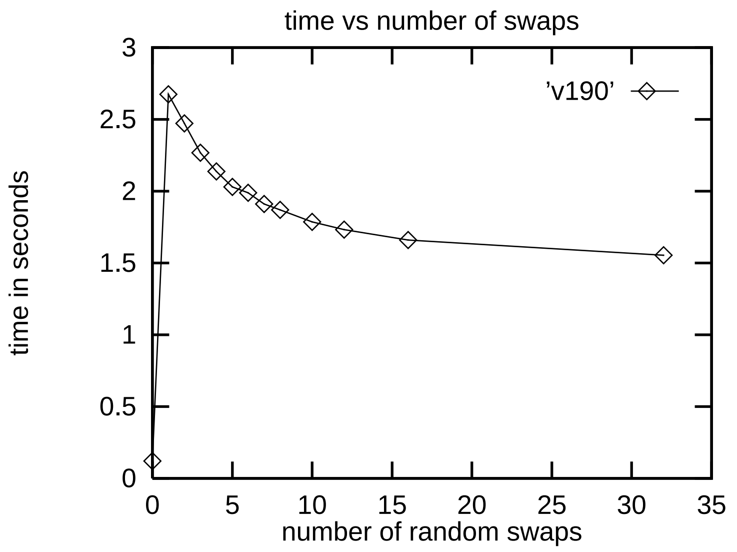


Figure 5: Results for **v190**

For each graph we made measurements for 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 16, and 32 random swaps. Each measurement was obtained by creating 10 graphs with the required number of swaps, and then averaging the running time of nauty over those graphs. We do not explicitly list the variances for each measurement, except to mention that they were very low in all but one case, which we will discuss shortly.

Figures 5 through 8 show a uniform picture of the effect of random swaps.

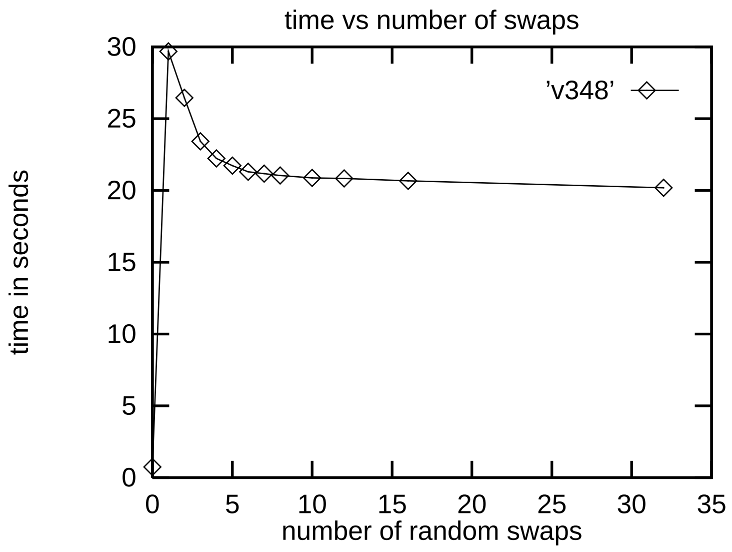


Figure 6: Results for **v348**

In each case nauty was very fast if there was no random swaps, and then much slower when random swaps were introduced. Further, as more swaps were introduced, nauty began to decrease its running time.

There are a few plausible explanations for this behaviour. One factor which may explain the large increase in time between 0 and 1 swaps, is the number of automorphisms in the resulting graph. When there were no swaps, the numbers of automorphisms of v190, v348, v486, and v864 were 3420, 6144, 7776, and 6912 respectively. Based on our previous analysis of the nauty algorithm, it is not hard to imagine that these large group sizes help nauty to aggressively prune the search space. After doing 1 or more random swaps, the number of automorphisms was almost always 1; out of 520 graphs that contained random swaps, only 5 had an automorphism group of size 2. In fact this is the cause of the one measurement with large variance: it contained two graphs with an automorphism group of size 2, and these were able to complete in approximate 2/3 the amount of time as the other runs. Clearly in the graphs with random swaps, nauty cannot use automorphisms to help prune the search tree.

This does not explain the fact the time decreased as the number of swaps increased, as all of these graphs had the approximately the same number of automorphisms (1 or 2). Intuitively we can argue that nauty is speeding up on the graphs with many swaps because it is using its refinement procedure more effectively. With each random swap we essentially "break" the structure of the

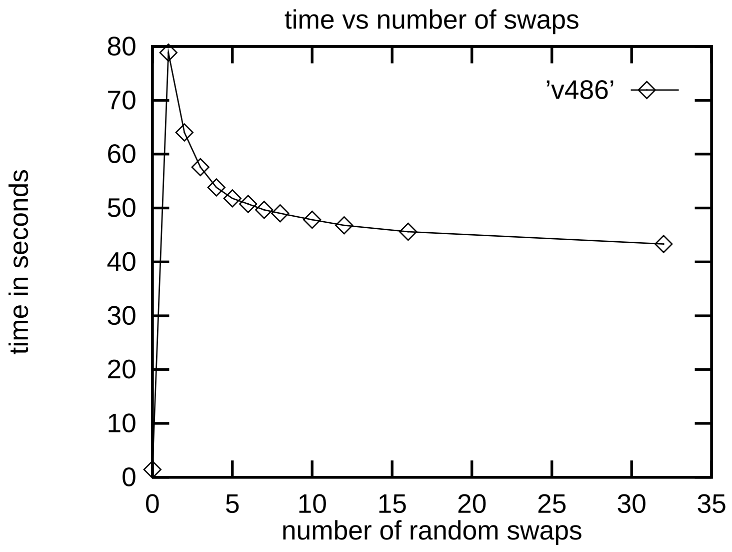


Figure 7: Results for **v486**

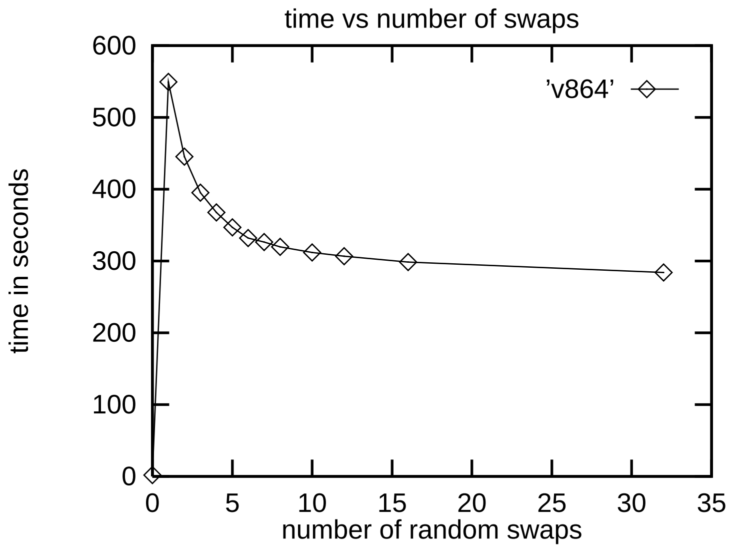


Figure 8: Results for **v864**

graph, and thus with many swaps the number of breaks increase, and hence nauty is more likely to find and exploit one of those "breaks".

One final test we performed was to try and measure the amount of increase, as it related to graph size. For each graph we plot the blowup ratio $r =$ (time with one swap)/(time with no swaps). For this test we included a number of graphs of various size, generated in a similar fashion as those in the previous test. As well we include a graph **v900** which is a 900 vertex graph generated by interlocking latin squares of size 10. The results are shown in Table 1. Not many conclusions can be made about these results, except to say that more work is necessary to discover how to predict these ratios.

n	degree	no swaps (sec)	one swap (sec)	blowup ratio
190	8	0.12	2.7	22
216	8	0.05	3.9	78
348	8	0.75	29.7	40
486	8	1.44	78.8	55
864	8	1.85	549.4	297
870	7	7.69	595.8	78
900	683	34.60	429.8	12
1140	9	16.50	963.5	58
1536	7	37.46	5683	152
1966	8	74.72	12091	162

Table 1: The effect of graph parameter settings on the blowup ratio

5 Conclusion

The theory of NP-completeness has allowed researchers to conclude that many problems are either polynomially solvable, or are probably intractable. Graph isomorphism is one of the few problems for which NP-completeness does not seem to guide the researcher. One is unsure of whether to search for a polynomial time solution, or to be content with algorithms that may have very bad worst-case behaviour. As the large amount of research on the complexity of graph isomorphism shows, researchers are trying to remedy that problem, believing that a deeper understanding of graph isomorphism will yield insights into the larger realm of complexity analysis.

However, the graph isomorphism problem has so many interesting applications that the computing community is not content to wait for its complexity to be completely determined. Instead a large body of work has concentrated on solving restricted versions of graph isomorphism very quickly, or solving the general problem very fast on average. When the graphs being averaged over are generated in a completely random fashion, it is trivial to be fast on average. However practical algorithms are very fast when averaged over the nebulous notion of real world graphs, by using powerful group theory tools to help prune the search space.

As we showed, it is not too difficult to create graphs which current algorithms find hard. Granted that these graphs were rather contrived, but it seems plausible that they might arise in an application where very regular graphs are generated, which might have a small amount of noise. One point to note is that even for "hard" instances, 1000 vertex graphs were solved in less than 10 minutes. This is quite a contrast with problems like graph coloring, were 10 minutes on a 1000 node problem would not be considered a long time, and may be a further indication of the "almost polynomial" nature of the graph isomorphism problem.

References

[1] K. Agusa, S. Fujita, M. Yamashita, and T. AE. On neural networks for graph isomorphism problem. In *The RNNS/IEEE Conference on Neuroinformatics and Neurocomputers*, pages 1142–1148, Rostov-on-Don, Russia, October 1992.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[3] L. Babai. Moderately exponential bound for graph isomorphism. *Proceedings of the Fundamentals of Computing Science, Lecture Notes in Computing Science 117*, pages 34–50, 1981.

[4] L. Babel. Isomorphism of chordal (6,3) graphs. *Computing*, 54:303–306, 1995.

[5] L. Babia, P. Erdos, and S. Selkows. Random graph isomorphism. *SIAM Journal of Computing*, 9(3):628–635, August 1980.

[6] L. Babia and L. Kucera. Canonical labeling of graphs in linear average time. In *Proceedings of the 20th IEEE Symposium on Foundations of Computing Science*, pages 39–46, 1979.

[7] D. Basin. A term equality problem equivalent to graph isomorphism. *Information Processing Letters*, 51:61–66, 1994.

[8] H. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11:631–643, 1990.

[9] K. Booth. Problems polynomially equivbalent to graph isomorphism. Technical Report CS-77-04, Univerisity of Waterloo, 1979.

[10] K. Booth and G. Lueker. Linear algorithms to recognize interval graphs and test for consectuative ones property. In *Proceedings of the 7th Annual ACM Symposium on the Theory of Computing*, pages 255–265, May 1975.

[11] R. Boppana, J. Hastad, and S. Zachos. Does co-np have short interactive proofs? *Information Processing Letters*, 25:27–32, 1987.

[12] I. Bouwer. *The Foster Census: R.M. Foster's Census of Trivalent Graphs*. Babbage Reaserch Company, Winnipeg, 1988.

[13] J. Chen. A linear time algorithm for isomorphism of graphs with bounded average genus. *Proceedings of the 18th Internation Workshop on Graph-Theoretic Concepts in Computer Science*, pages 103–113, 1992.

[14] O. Cogis and O. Guinaldo. A linear descriptor for conceptual graphs and a class for polynomial isomorphism test. In *Proceedings of the 3rd International Conference on Conceptual Structures*, pages 263–277, Santa Cruz, USA, August 1995.

[15] G. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11:13–21, 1981.

[16] D. Corneil and D. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM Journal of Computing*, 9(2):281–297, May 1980.

[17] D. Damarchi, G. Masera, and G. Piccinini. A VLSI processor array for graph isomorphism. *International Journal of Electronics*, 76(4):655–679, 1994.

[18] I. Fillotti and J. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *Proceddings of the 12th ACM Symposium on Theory of Computing*, pages 235–243, 1980.

[19] S. Goldwasser, S. micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18:186–208, 1989.

[20] J. Hopcroft and J. Wonk. A linear time algorithm for isomorphism of planar graphs. *Proceeedings of the 6th Annual ACM Symposium on the Theory of Computing*, pages 172–184, 1974.

[21] W. Hsu. O(MN) algorithms for the recognition and isomorphism probelm on circular-arc graphs. *SIAM Journal of Computing*, 24(3):411–439, June 1995.

[22] J. JaJa and S. Kosaraju. Parallel algorithms for planar graph isomorphism and realted problems. *IEEE Transactions on Circuits and Systems*, 35:304–311, 1988.

[23] J. Kobler, U. Schoning, and J. Toran. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhauser, Boston, 1993.

[24] E. Luks. Isomorphism of graphs of bounded valence can be testes in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[25] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8:131–132, 1979.

[26] J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Sciences*, 19:229–250, 1979.

[27] B. McKay. Nauty User's Guide (version 1.5). Computer Science Department, Austrialian National University.

[28] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[29] R. Read and D. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[30] E. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, New Jersey, 1977.

[31] U. Schoning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988.