Université de Montréal

A survey of graph and subgraph isomorphism problems

par

Yaohui Lei

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maîtrise ès sciences (M.Sc.)

en informatique

Avril, 2003

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

A survey of graph and subgraph isomorphism problems

présenté par:

Yaohui Lei

a été évalué par un jury composé des personnes suivantes:

Alain Tapp (président-rapporteur)

Gilles Brassard (directeur de recherche)

Gena Hahn (Co-directeur de recherche)

Pierre McKenzie (membre de jury)

Mémoire accepté le : 19 décembre 2003

Abstract

Graphs are useful as a flexible and versatile data structure for the representation of objects and concepts. The graph and subgraph isomorphism problems have been thoroughly studied for decades. They have been drawing great interest in many theoretical and practical domains including transport planning, chemistry, geography, information retrieval, automata theory, linguistics, computer-aided-design, mathematics and computer science.

An introduction and preliminaries begin this survey. This includes basic computational complexity theory, group theory and graph theory. Then, we begin the graph isomorphism by showing the isomorphism-complete class. Some special graph isomorphisms in the P class are presented from two approaches: combinatorial approach and group-theoretic approach.

The survey is followed by an analysis of the subgraph isomorphism problem. First, we give an overview of complexity results for this problem. Then, we present some special subgraph isomorphisms such as subtree, planar subgraph, etc.

From the practical algorithms point of view, several graph and subgraph isomorphism algorithms are introduced. A performance comparison of these algorithms is outlined.

At the end of the survey, we give a conclusion and a discussion of these problems. Since we usually refer to special cases for these problems, we consider some other possibilities which might be interesting.

Key words: graph isomorphism, subgraph isomorphism, computational complexity, group theory, isomorphism-complete class, combinatorial approach, group-theoretic approach

Résumé

Les graphes sont utiles comme une structure de données flexible et versatile pour la représentation des objets et des concepts. Les problèmes de l'isomorphisme de graphe et l'isomorphisme de sous-graphe ont été bien étudiés depuis des décennies. Ils ont été pris du grand intérêt dans beaucoup de domaines théoriques et pratiques incluant la planification de transport, chimie, géographie, recherche d'information, automates théorie, linguistique, Conception Assistée par ordinateur, des mathématiques et informatique.

Une introduction et les préliminaires commencent cette synthèse. Ceci inclut la base de la théorie de complexité du calcul, la théorie de groupe et la théorie de graphe. Ensuite, nous commençons l'isomorphisme de graphe en montrant la classe isomorphisme-complet. Quelques isomorphisms spéciaux de graphe dans la classe de P sont présentés de deux approches : approche combinatoire et approche groupe-théorétique.

La synthèse est suivie d'une analyse du problème d'isomorphisme de sous-graphe. D'abord, nous donnons une vue générale des résultats de complexité pour ce problème. Puis, nous présentons certains isomorphisms spéciaux de sous-graphe tels que le sous-arbre, le sous-graphe planaire, etc.

Du point de vue d'algorithmes pratiques, plusieurs algorithmes sur l'isomorphisme de graphe et l'isomorphisme de sous-graphes sont présentés. Une comparaison de performance de ces algorithmes est décrite.

À la fin de cette synthèse, nous donnons une conclusion et une discussion de ces problèmes. Puisque nous référons souvent à des cast spéciaux pour ces problèmes, nous considérons quelques autres possibilités qui pourraient être intéressantes.

Mots-clés: isomorphisme de graphe, isomorphisme de sous-graphe, complexité du calcul, théorie de groupe, isomorphisme-complet, approche combinatoire, approche groupe-théorique

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction and Preliminaries

1.1 Introduction

Graphs are useful as a flexible and versatile data structure for the representation of objects and concepts. It is well known that graph representations are widely used for dealing with structural information in different domains such as transportation, networks, image interpretation and processing, computer-aided design, pattern recognition, and many other subfields of science and engineering. For example, the intersections and traffic routes of a city can be represented by graphs. The intersections are represented by *vertices* while the routes are drawn as *edges* in a graph.

Two graphs are *equal* if they have the same vertex set and the same edge set. But there are other ways in which two graphs could be regarded as being the same. For instance, one could regard two graphs as being "the same" if it is possible to rename the vertices of one and obtain the other. Such graphs are identical in every respect except for the names of the vertices. In this case, we call the graphs *isomorphic*. When graphs are small enough,

whether two graphs are isomorphic can be detected easily manually, while this becomes infeasible when the graphs are much bigger.

In this survey, we give an overview of the subject not only from a theoretical point of view but also from a practical aspect. On the one hand, we give an introduction and preliminaries to the mathematics in order to make understanding easier. On the other hand, complicated proofs and algorithms with deep theory background are simplified and outlined. Nevertheless, in order to keep the integrity and the continuity, some of these proofs and algorithms are quoted almost verbatim from their sources.

1.1.1 Graph isomorphism

The *graph isomorphism* (GI) problem was listed as an important open problem already in Karp [75] over three decades ago. The graph isomorphism problem is deciding whether two given graphs are isomorphic, i.e. whether there is a bijective mapping from the vertices of one graph to the vertices of the second graph such that the edges are respected. Much work [79] is dedicated to the search for an exact isomorphism between two graphs or subgraphs.

It is a problem of interest in many theoretical and practical domains [79] including transport planning, chemistry, geography, information retrieval, automata theory, linguistics, computer-aided-design, mathematics and computer science [23, 30, 89]. Graphs represent various real structures or situations; we want to know whether two structures or situations are essentially the same with respect to a selected point of view, in other words, isomorphic. Figure 1.1 gives two examples: one is for the directed graph and the other is for the undirected graph.

The GI problem is very simple to define and understand, but it seems very difficult to give an efficient solution, i.e. a polynomial-time algorithm. Because of its theoretical and

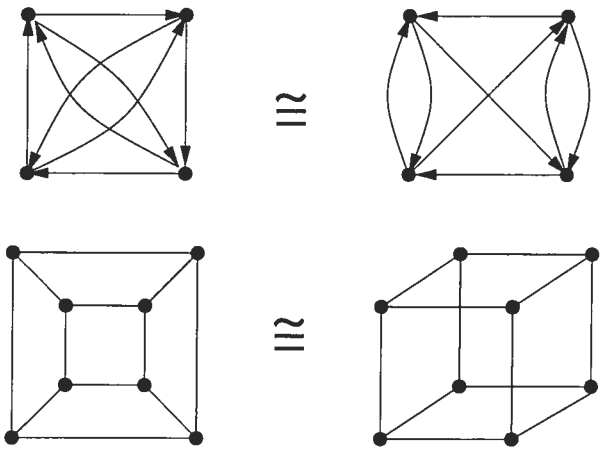

Figure 1.1: two examples of isomorphic graphs

practical importance the problem has been studied from many different points of view [51]. There are some algorithms to solve this problem, but they have an exponential-time complexity [79]. Hence, time complexity is the main issue of the graph isomorphism problem.

1.1.2 Subgraph isomorphism

While graph isomorphism treats the isomorphism relation between whole graphs, the *subgraph isomorphism* focuses on subgraphs of one graph. Subgraph isomorphism problem is to determine whether there is a subgraph of one given graph which is isomorphic to a given second graph. Subgraph isomorphism is very important in computer vision, bio-computing and image processing. Like graph isomorphism, it has been studied in depth for both theoretical and practical interests. For example, [127], one of possible applications of subgraph isomorphism is for finding whether a given chemical compound is a sub-compound of a further specified compound, given the structural formulas. Moreover, subgraph isomorphism is an important and very general form of exact pattern matching, such as string searching, sequence alignment, tree comparison and pattern matching on

graphs.

Subgraph isomorphism is a common generalization of many important graph problems [47], including Hamilton paths, cliques, matchings, girth, and shortest paths. Most of the research on subgraph isomorphism algorithms has been based either on heuristic search techniques as in [35, 127], or on constraint satisfaction techniques as in [40, 93]. The best known algorithms for subgraph isomorphism are based on a relational view approach [40] on exhaustive search with backtracking.

The rest of this survey is organized as follows. Chapter 1 introduces the terminology and preliminaries on computational complexity, group theory and graph theory. Chapter 2 focuses on the graph isomorphism problem. Complexity results, combinatorial and group-theoretic approaches to solve graph isomorphism problem are shown as well as polynomial-time algorithms on special cases of graphs such as trees, planar graphs, bounded valence graphs, etc. In Chapter 3, the subgraph isomorphism problem is discussed by presenting complexity results in subtrees, planar graphs and embedded graphs. Another point of view regarding subgraph isomorphism, the relational view, is presented too. As for practical algorithms in graph and subgraph isomorphism problems, Chapter 4 shows basic ideas of major algorithms, Nauty, Ullmann and Schmidt & Druffel, along with a comparison of performance. A conclusion and a discussion of these aspects are given in Chapter 5.

1.2 Basic computational complexity

In this section, we give a brief overview of computational complexity theory. For more detailed description, readers are encouraged to consult [3, 110, 111].

The theory of computation, a subfield of computer science and mathematics, is the study of mathematical models of computing, independent of any particular computer hardware. Complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are *time* (how many steps it takes to solve a problem) and *space* (how much memory it takes to solve a problem).

Given a problem, we need an algorithm to solve it. How do we know that an algorithm is a "good" one? A useful measure of performance is "the time or space required to solve a problem as a function of the size of data". Generally speaking, computational complexity theory studies:

- the efficiency of algorithms
- the inherent difficulty of problems of practical and/or theoretical importance

An important discovery in the area is that computational problems can vary tremendously in the effort required to solve them precisely.

Definition 1.1 *Consider functions* $f, g : N \to R^+$. *Say that* $f(n)$ *is of the order of* $g(n)$, *written* $f(n) \in O(g(n))$ *(called big-O notation), if there is a positive constant* c *such that for every* n, $f(n) \leq c \cdot g(n)$.

Algorithms which have a polynomial or sub-polynomial time complexity (that is, they take time $f(n) \in O(g(n))$, where $g(n)$ is a polynomial), are often practical.

Algorithms with complexities which cannot be bounded by polynomial functions are called **exponential-time** algorithms.

1.2.1 Turing machine

In computational complexity theory, we use frequently the idea of a **Turing machine**. A Turing machine is an abstract model of computer execution and storage introduced in 1936 by Alan Turing to give a mathematically precise definition of "algorithm" or "mechanical procedure".

Definition 1.2 *A Turing machine consists of:*

1. *A* tape *which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special blank symbol and one or more other symbols. The tape is assumed to be arbitrarily extendible to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to during a computation are assumed to be filled with the blank symbol.*

2. *A* head *that can read and write symbols on the tape and move left and right.*

3. *A* state *register that stores the state of the Turing machine. The number of different states is always finite and there is one special start state with which the state register is initialized. Some states may be designated as "accept" states.*

4. *A* transition table *that tells the machine what symbol to write, how to move the head ("L" for one step left, and "R" for one step right) and what its new state will be, given the symbol it has just read on the tape and the state it is currently in. If there is no entry in the table for the current combination of symbol and state then the machine will halt.*

5. *The Turing machine accepts its input if it halts in an "accept" state and refuses otherwise.*

Definition 1.3 *An ordinary (deterministic) Turing machine (DTM) is a tuple:*

$$T = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$$

where Q is a finite set of states, Γ is the finite set of tape symbols, $B \in \Gamma$ is the blank symbol, $\Sigma \subseteq \Gamma$ is the set of input symbols, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the move function, $q_0 \in Q$ is the start state and $F \subseteq Q$ is the set of final states.

Definition 1.4 *An non-deterministic Turing machine (NTM) is a tuple:*

$$T = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject} \rangle$$

where Q is a finite set of states, Σ is the input alphabet, not containing blank symbol, Γ is the finite set of tape symbols, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the move function, $q_0 \in Q$ is the start state, $q_{accept} \in Q$ is the accept state and $q_{reject} \in Q$ is the reject state.

A NTM differs from a DTM in that rather than a single instruction triplet, the transition rule may specify a number of alternate instructions. NTM can be thought of a generalization of DTM. At each step of the computation we can imagine that the computer "branches" into many copies, each of which executes one of the possible instructions. Whereas a DTM has a single "computation path" that it follows, a NTM has a "computation tree". If any branch of the tree halts with an "accept" condition, we say that the NTM accepts the input.

Definition 1.5 *A configuration of a Turing machine is a 3-tuple (u, q, v), where*

- *uv is the current tape contents;*
- *u is the part (possibly empty) of the string from the leftmost symbol till the scanned cell of the tape;*
- *if a is the symbol in the scanned cell, then v is the part (possibly empty) of the string from a to rightmost non-blank symbol.*
- *q is the current state.*

We next define Alternating Turing Machine (ATM). Just as an NTM is a generalization of a DTM, an ATM is a generalization of an NTM.

Definition 1.6 *An ATM* $M = \langle Q, \Sigma, \Gamma, \delta, F \rangle$ *(*Q *is a set of states,* Γ *is the tape alphabet,* δ *is the transition function,* Σ *is the input alphabet, and* F *is the set of final states) is an NTM with the following differences:*

1. *Each state* $q \in Q$ *is a pair* $< n, z >$*, where* $z \in \{"Universal", "Existential"\}$ *is a "label" for the state and* n *is the state name. This partitions* Q *into a set of existential (*$\exists$*) states and a set of universal (*$\forall$*) states. Fix an input* x*. We call a configuration (tape contents, position of R/W head, state of control) to be an existential configuration if its state is existential. Universal configurations are defined similarly.*
2. *Acceptance of M: If a Turing Machine can legally go from a configuration* C_1 *to another configuration* C_2 *in a single step according to the transition function,* C_1 *is called the parent of* C_2 *or* C_2 *is the child of* C_1*. Configurations without any children are called leaf configurations and others are called non-leaf configurations. We now recursively label each configuration to be either accepting or rejecting as follows.*

(a) A leaf configuration whose state is a final state is labeled "accepting". A leaf configuration whose state is not a final state is labeled "rejecting".

(b) A non-leaf existential configuration is labeled "accepting" if at least one of its children is labeled "accepting", and it is labeled "rejecting", otherwise. A non-leaf universal configuration is labeled "accepting" if all of its children are labeled "accepting", and it is labeled "rejecting", otherwise.

(c) The M is said to accept the input x, if and only if its starting configuration is labeled "accepting".

In thinking about the computation of an ATM, it is helpful to represent the computation as a tree, see figure 1.2.

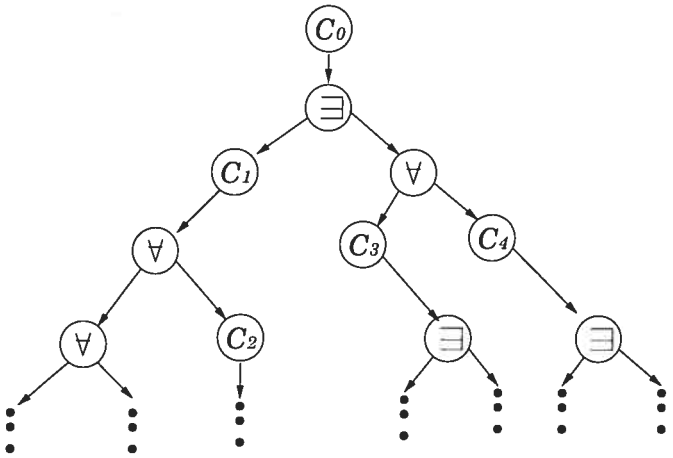

Figure 1.2: sample computation tree of an ATM

Each node of the tree is labeled with the machine's configuration and has arrows pointing to the configurations reachable by outgoing transitions from the node. The outcome of the computation is determined recursively as follows. A node which is in the machine's accept state q_f accepts. A node in the $\exists$ state accepts if and only if at least one of its children accepts. A node in the $\forall$ state accepts if and only if both of its children accept. Every

other node has only one child, and accepts if and only if its child accepts. The machine accepts if and only if the root of its computation tree accepts.

1.2.2 Decision problems

In the theory of computation a **problem** is a set of finite-length questions (strings) with associated finite-length answers (strings). A **decision problem** is a problem that requires a YES or NO answer. These problems are also referred to as **recognition problems**.

A decision problem is usually formalized as the problem of deciding whether a given string belongs to some specified set of strings, also called a formal language. The set contains exactly those questions whose answers were "YES". If there is an algorithm that is able to correctly decide for every possible input string whether it belongs to the language, then the problem is called *decidable* and otherwise it is called *undecidable*. Important points are:

- If a problem is decidable, there is a Turing machine M that when processing any instance, x, of P (i.e., any string x on its input tape) will eventually finish in state q_{accept} if x is a "YES" instance of the problem and will eventually finish in state q_{reject} if x is a "NO" instance of the problem.
- A problem for which no such Turing machine exists is undecidable.
- Decision problems are a whole lot easier to deal with when looking for special problems, like unsolvable problems, because proposed solutions only either accept or reject their input rather than producing some likely complex output on its tape that needs to be analyzed.

- If we can find a decision problem that is undecidable, the we know that there are unsolvable problems. We don't need to look for some very complex general problem that is unsolvable if we can find a very simple decision problem that we can prove is undecidable by showing that there is no possible Turing machine that could decide it.

Computer programs, from a tiny "Hello, world!" procedure to a huge operating system, may be viewed as computing functions. Since all computers employ binary notation, such functions are defined over sets of binary strings. In considering the question "What problems can be solved by computers?", it is sufficient to concentrate on decision problems. Hence, "What problems can be solved by computers?" is equivalent to "What decision problems can be solved?".

Any binary string can be viewed as a representation of some natural number. Thus for decision problems on binary strings we can concentrate on the set of functions of the form

$f : N \rightarrow \{0, 1\}$

INPUT: n a natural number

OUTPUT: 1 if n satisfies a given property; 0 if n does not satisfy it.

An example is the **Prime** problem: return 1 if n is a prime number; 0 if n is a composite number.

Decision problems are important because any general problem with an n-bit answer can be transformed into a decision problem with a YES/NO answer. Solving the general problem can't be more than n times harder than solving the decision problem. There are several ways to do this transform. For example, if the general problem is of the form:

Given an input X, return the answer string Y

then the associated decision problem is:

Given an input X and an integer k, return whether the kth bit of Y is 1

1.2.3 Polynomial reductions and transformations

The basic tools for relating the complexities of various problems are polynomial reductions and transformations. We say that a problem A reduces to another problem B in polynomial-time, denoted as $A \propto_p B$ if:

1. there is an algorithm for A which uses a subroutine for B, and
2. each call to the subroutine for B counts as a single step, and
3. the algorithm for A runs in polynomial-time.

If $A \propto_p B$ and $B \propto_p A$ we say that the problems are **polynomially equivalent** and write $A \equiv_p B$.

The practical implication comes from the following proposition and its contrapositive:

If A polynomially reduces to B and there is a polynomial-time algorithm for B, then there is a polynomial-time algorithm for A.

There are three cases related to the proposition.

1. (A reduces to B) and (B is "easy") $\Longrightarrow$ A is "easy"
2. (A reduces to B) and (A is "hard") $\Longrightarrow$ B is "hard"
3. (A reduces to B) and (B is "hard") $\Longrightarrow$ no conclusion for A - (a very common case)
4. (A reduces to B) and (A is "easy") $\Longrightarrow$ no conclusion for B - (also a very common case)

That is, if A polynomially reduces to B, then B is at least as hard as A.

1.2.4 The classes P, NP, NP-hard and NP-complete

Definition 1.7 *The class P (polynomial-time) consists of all those decision problems that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input; the class NP (non-deterministic polynomial-time) consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time by a non-deterministic Turing machine.*

Definition 1.8 *The NP-hard (Non-deterministic Polynomial-time hard) refers to the class of decision problems that contains all problems H such that for all decision problems L in NP there is a polynomial-time many-one reduction to H. Informally this class can be described as containing the decision problems that are at least as hard as any problems in NP, although it might, in fact, be harder.*

Definition 1.9 *The NP-complete is the complexity class of decision problems for which answers can be checked for correctness by an algorithm whose run time is polynomial in the*

size of the input (that is, it is NP) and no other NP problem is more than a polynomial factor harder. Informally, a problem is NP-complete if answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly.

In complexity theory, the NP-complete problems are the hardest problems in NP, in the sense that they are the ones most likely not to be in P.

Clearly, $P \subseteq NP$. Is P a proper subset of NP? This is the most important open question in theoretical computer science. Most people think that the answer is probably "yes", then there are some problems in NP which are not in P (See Figure 1.3).

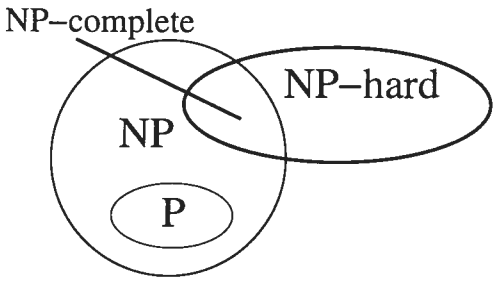

Figure 1.3: classes P and NP

If $P = NP$ then all of the NP problems collapse to P. Ladner [80] shows that this is the only case.

Theorem 1.1 *If $P \neq NP$ then there exists sets in NP that are neither in P nor $NP-complete$.*

Some people believe the question may be undecidable within the current axiomatization. A $1,000,000 prize [131] has been offered for a correct solution.

The question "Is $P = NP$?" can be rephrased as: if positive solutions to a YES/NO problem can be verified quickly, can the answers also be computed quickly? Here is an

example to get a feeling for the question. Given two large numbers X and Y, we might ask whether Y is a multiple of some integer between 1 and X, exclusive. For example, we might ask whether 69799 is a multiple of some integer between 1 and 250. The answer is YES, though it would take a fair amount of work to find it manually. On the other hand, if someone claims that the answer is YES because 223 is a divisor of 69799, then we can quickly check that with a single division. Verifying that a number is a divisor is much easier than finding the divisor in the first place. The information needed to verify a positive answer is often called a "certificate". So we conclude that given the right certificates, positive answers to our problem can be verified quickly (i.e. in polynomial time) and that's why this problem is in NP. It is not known whether the problem is in P. The special case where $X = Y$ was first shown to be in P in 2002 [1], after many years of research.

The NP-complete term for hard problems essentially means: "abandon all hope of finding an efficient algorithm for the exact solution of this problem". We should point out that proving or knowing that a problem is NP-complete is not all that negative. Knowing such limitations, people do not waste time on impossible projects and instead turn to less ambitious approaches, for example to find approximate solutions, to solve special cases or to alter problems a little so that they become tractable (even at a loss of some fit to real-life situation, which is particularly useful in practical application since sometimes we cannot provide or guarantee an exact mapping between "real life" and theoretical representation). The goal of this theory is therefore to assist algorithm designers in directing their efforts toward promising areas and avoid impossible tasks.

A NP-complete problem has the following most important property. Finding an efficient algorithm for any NP-complete problem implies that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be recast as any other

member of the class: they are all polynomially equivalent.

The practical significance of showing the recognition version of an optimization problem to be NP-complete is that one should not pursue the search for a good optimizing algorithm for such a problem and be content with finding a good approximating (i.e. heuristic) algorithm.

1.2.5 The class NC

The class NC (short for "Nick's Class", introduced by Nick Pippenger) is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors. In other words, a problem is in NC if there are constants c and k such that it can be solved in time $O((\log n)^c)$ using $O(n^k)$ parallel processors.

Just as the class P can be thought of as the class of tractable problems, NC can be thought of as the class of problems that can be solved efficiently on a parallel computer. It is unknown whether $NC = P$, but most researchers suspect this to be false, meaning that there are some tractable problems which are "inherently sequential" and cannot significantly be sped up by using parallelism.

The parallel computer in the definition can be assumed to be a **parallel, random-access machine** (PRAM). That is, a parallel computer with a central pool of memory, and any processor can access any bit of memory in constant time.

1.3 Group theory preliminaries

We begin with a brief review of elementary facts from group theory, which we give usually without proofs. For details, readers can consult [6, 81]. It is essential to understanding the

recent approach, group-theoretic techniques, to solve graph isomorphism.

1.3.1 Group definitions

Definition 1.10 *A **group** is a set G together with a binary operation $\circ$ on G such that*

1. *$a \circ (b \circ c) = (a \circ b) \circ c$, for all $a, b, c \in G$ (associative law),*
2. *There exists an element $e \in G$, called the **identity** element, such that $a \circ e = e \circ a = a$, for all $a \in G$,*
3. *To each $a \in G$, there exists an element b, called the **inverse** of a, such that $a \circ b = b \circ a = e$.*

In the third condition, b is usually denoted as a^{-1} because it is unique. For any c such that $a \circ c = c \circ a = e$, we have $c = c \circ e = c \circ (a \circ b) = (c \circ a) \circ b = b$.

A common group example is $\mathbb{Z}_p^* = \langle \{1, 2, \cdots, p-1\}, \cdot \rangle$, where p is a prime and the operation is multiplication modulo p.

The **order** of the group G is the number of its elements and is denoted $|G|$. A group of order p^n, with p a prime number and $n \geq 1$, is called a p-**group**.

Let G be a group and $a \in G$. Let n be the smallest positive integer, if it exists, such that $a^n = e$. Then n is called the order of a and we shall write $order(a) = n$. One also says that a is of **finite order** with order n.

An element $g \in G$ such that $order(g) = |G|$ is called **generator**. A group G that has such a generator is called **cyclic**.

A **trivial group** I is the group consisting of the identity e only.

A group G is **commutative**, if, for all $g, h \in G$, $g \circ h = h \circ g$. Otherwise, G is **non-commutative**. A commutative group is also called **Abelian** in honour of one of the first group theorists, Neils Henrik Abel.

A nonempty subset H of a group G is called a **subgroup** of G if

1. $a, b \in H$ implies that $ab \in H$,
2. $e \in H$ (where e is the identity of G),
3. $a \in H$ implies that $a^{-1} \in H$.

We write $H \leq G$, when H is a subgroup of G.

A group G is called **permutation group** if G is a set of permutations of a fixed set X and the group operation is the composition of permutations (we think of a **permutation** as a bijection from the set X onto itself).

Let G_1 and G_2 be groups with operations $\circ, *$ respectively. The **Cartesian product** $G_1 \times G_2$ is the set of pairs $G_1 \times G_2 = \{(g_1, g_2) \mid g_1 \in G_1, g_2 \in G_2\}$. Let us define a group operation **multiplication** on $G_1 \times G_2$. For two arbitrary elements (a_1, a_2) and (b_1, b_2) in $G_1 \times G_2$, define their product by $(a_1, a_2)(b_1, b_2) = (a_1 \circ b_1, a_2 * b_2)$. The set of all ordered pairs (x_1, x_2) such that $x_1 \in G_1$, and $x_2 \in G_2$ form a group under the operation multiplication. We call this group the **direct product** of G_1 and G_2.

Let X be a fixed set of cardinality n. Let $Sym(X)$ denote the set of all bijections from X onto itself, i.e. the set of all permutations of X, and let the operation be composition. Then $Sym(X)$ is a permutation group and is called the **symmetric group** on X. If Y

is another set of cardinality n, then a bijective map f between X and Y defines a unique correspondence between the elements in $Sym(X)$ and the elements in $Sym(Y)$. This correspondence says that we have $f(\pi) \circ f(\varphi) = f(\pi \circ \varphi)$ for all $\pi, \varphi \in Sym(X)$. We call f an **isomorphism** between $Sym(X)$ and $Sym(Y)$. We usually choose $X = \{1, 2, \cdots, n\}$, the set of the first n natural numbers. In this case, $Sym(X)$ is abbreviated by S_n.

A **transposition** is a permutation of a set which fixes all but two elements. Let X be a set of cardinality greater than 1. Consider the set of those elements in $Sym(X)$ which can be expressed as the product of an even number of transpositions. This set is closed under composition and thus forms a permutation group $Alt(X)$ which is called **alternating group** on X. The order of $Sym(X)$ is exactly twice the order of $Alt(X)$.

1.3.2 Cosets and Lagrange's Theorem

Given H, a subgroup of G, and $g \in G$, the set $Hg = \{h \circ g \mid h \in H\}$ is a **right coset** of H in G. Similarly, the set $gH = \{g \circ h \mid h \in H\}$ is a **left coset** of H in G. Note that H is both a left and a right coset of itself. It is easy to show that two left (right) cosets of H are either disjoint or equal, and that all cosets are of cardinality equal to the order to H. Thus, we may **partition** G into the left (right) cosets of H.

The number of distinct left (equivalently, right) cosets of H is called the *index* of H in G, and is written $[G : H]$.

Theorem 1.2 (Lagrange) *The order of G is equal to the product of the order of H and the index of H in G, i.e. $|G| = |H| \times [G : H]$.*

It follows that the order of any subgroup H must divide the order of G.

1.3.3 The Orbit-Stabilizer Theorem

Definition 1.11 *A group G is said to act on a set X when there is a map $\phi : G \times X \to X$ such that the following conditions hold for all elements $x \in X$:*

- $\phi(e, x) = x$ *where e is the identity element of G.*
- $\phi(g, \phi(h, x)) = \phi(gh, x)$ *for all $g, h \in G$.*

We write gx for $\phi(g, x)$. Suppose that the group G acts on the set X. If we start with the element $x \in X$ and apply group elements in all possible ways, we get

$$B(x) = \{gx : g \in G\}$$

which is called the **orbit** of x under the action of G. The action of G on X is **transitive** (we also say that G acts **transitively** on X) if there is only one orbit, in other words, for any $x, y \in X$, there exists $g \in G$ such that $gx = y$. Note that the orbits partition X, because they are the equivalence classes of the equivalence relation given by $y \sim x$ if and only if $y = gx$ for some $g \in G$.

The **stabilizer** of an element $x \in X$ is

$$G(x) = \{g \in G : gx = x\},$$

the set of elements that leave x fixed. A direct verification shows that $G(x)$ is a subgroup. This is a useful observation because any set that appears as a stabilizer in a group action is guaranteed to be a subgroup; we need not bother to check each time.

The following theorem, known as the **Orbit-Stabilizer** theorem, is fundamental in many applications.

Theorem 1.3 *Suppose that a group G acts on a set X. Let $B(x)$ be the orbit of $x \in X$, and let $G(x)$ be the stabilizer of x. Then the size of the orbit is the index of the stabilizer, that is,*

$$|B(x)| = [G : G(x)].$$

Thus if G is finite, then $|B(x)| = |G|/|G(x)|$; in particular, the orbit size divides the order of the group. □

1.3.4 Normal Subgroups, Homomorphism and Automorphism

A subgroup H of G is **normal** written $H \lhd G$, if, for all $g \in G$, $Hg = gH$.

Let G and G' be two groups, h a map from G to G'. Then the map h is a **group homomorphism** if, for all $g_1, g_2 \in G$, $h(g_1 \cdot g_2) = h(g_1) \cdot h(g_2)$.

The set K of all elements in G which are mapped to the identity e' of G' is a normal subgroup of G and is called the *kernel* of the homomorphism, denoted by $Ker(h)$. If the subgroup K is I, the trivial group, then h is an **isomorphism**. The *image* of a homomorphism h is the set of all the elements of G' to which are mapped the elements of G, denoted by $Im(h)$.

An isomorphism from G onto itself is called an **automorphism** of G.

1.4 Graph preliminaries

1.4.1 Basic graph terminology

Since graphs have been widely studied in different contexts, there are various different terminologies in this field. The comprehensive book written by Brandstädt, Le and Sprinrad [25] is a good reference. The notation and concepts in this survey are based on this book.

- A **graph** is an ordered pair of sets $G = (V, E)$ where V (or $V(G)$ to emphasize that it belongs to the graph G) is the vertex set and E (or $E(G)$ to emphasize that it belongs to the graph G), $E \subseteq \{\{u, v\} \mid u \neq v, u \in V, v \in V\}$, the edge set. Usually, the number of vertices, $|V|$ is denoted by n, while the number of edges, $|E|$, is denoted by m. If $e = \{u, v\} \in E(G)$, we say that vertices u and v are **adjacent** in G, and that e joins u and v. We'll also say that u and v are the **ends** of e, denoted by $u \in e, v \in e$. The edge e is said to be **incident** with u (and v), and vice-versa. We write uv (or vu) to denote the edge $\{u, v\}$, on the understanding that no order is implied. Note that $E(G)$ is a set. This means that two vertices either are adjacent or are not adjacent, there is no possibility of more than one edge joining a pair of vertices. The elements of E are 2-subsets of V. Thus, a vertex cannot be adjacent to itself.

- A **digraph** (short for **directed graph**) is an ordered pair of sets $G = (V, A)$, where V is a set of vertices and A is a set of ordered pairs (called **arcs**) of vertices of V.

- The **open neighbourhood** of a vertex v in a graph G is the set $N(v) = \{u \mid uv \in E\}$; the **closed neighbourhood** is $\overline{N}(v) = N(v) \cup \{v\}$.

- The **adjacency matrix** of a graph on the vertex set $\{1, \cdots, n\}$ is an $n \times n$ 0-1 matrix $A = (a_{ij})$ in which the entry $a_{ij} = 1$ if there is an edge from vertex i to vertex j and is 0 if there is no edge between vertex i and vertex j.

- The **incidence matrix** of a graph on the vertex set $\{1, \cdots, n\}$ and the edge set $\{1, \cdots, m\}$ is an $n \times m$ matrix $A = (a_{ij})$ in which then entry $a_{ij} = 1$ if edge j is incident with vertex i and 0 otherwise.

- A **walk** (or, $v_0 - v_n$ walk) in a graph is an alternating sequence of vertices and edges, $v_0, e_1, v_1, e_2, v_2, e_3, v_3, \cdots, e_n, v_n$ such that $e_i = v_{i-1}v_i$ for $1 \leq i \leq n$. The integer n is the **length** of the walk. It is the number of edges in the walk, one less than the number of vertices. A **closed walk** is a walk that starts and ends at the same vertex.

- A **trail** is a walk in which no edge is repeated. Similarly, a **closed trail** is a trail that starts and ends at the same vertex. A **path** is a walk in which no vertex is repeated. A graph which has a path between every pair of vertices is called **connected graph.**

- A **cycle** (or **circuit**) is a closed path which does not contain a vertex twice (except at the beginning and end).

- A **loop** is an edge that connects a vertex to itself.

- The **distance** $d_G(x, y)$ in graph G of two vertices x, y is the length of a shortest $x - y$ path in G; if no such path exists, we set $d(x, y) := \infty$.

- The greatest distance between any two vertices in graph G is the **diameter** of G, denoted by $diam(G)$.

- A **wheel** is a graph that consists of a cycle and one vertex in the "middle" which is connected to all the vertices on the cycle. An **odd wheel** is a wheel whose outer cycle is of odd length, and an **even wheel** has an even cycle for the "rim".

- The **girth** of a graph G is the length of the shortest circuit of G (or infinity if G has no circuit).

- A **tree** is a connected graph that has no circuits. Sometimes it is convenient to consider one vertex of a tree as special; such a vertex is then called **root** of this tree. A tree with a fixed root is a **rooted tree**. Choosing a root r in a tree T imposes a partial ordering on $V(T)$ by letting $x \leq y$ if $(x, y) \in T$. This is the **tree-order** on $V(T)$ associated with T and r. Note r is the least element in this partial order, every leaf $x \neq r$ of T is a maximal element.

- The path P_n (n the number of vertices) is a tree with two vertices of degree 1 and the other $(n-2)$ vertices of degree 2. This graph is called **path graph** P_n.

- The **subgraph** of G induced by a subset W of its vertex set V (i.e. $W \subseteq V$) is the graph formed by the vertices in W and all the edges of G whose two endpoints are in W. It is denoted as $G[W]$. Analogously, we define the subgraph $G[F]$ induced by the set of edges F.

- The **complement** $\bar{G}$ of G is a graph on V, but two distinct vertices are adjacent in $\bar{G}$ if and only if they are non-adjacent in G.

- A **stable set** of G is a subset of vertices with no edge between any two of them.

- The **degree** of a vertex V of G is the number of edges incident to it. It is also called **valence**, is denoted by $d(v)$ and is given by $d(v) = |N(v)|$.

- The **connected components** of a graph G are the connected subgraphs of G induced by sets of vertices such that no two vertices in different sets are connected.

- A **cut vertex** is a vertex whose removal (along with all edges incident with it) produces a graph with more connected components than the original graph.

- A **complete graph** is a graph in which any two distinct vertices are adjacent. A complete graph on n vertices is denoted by K_n. A **clique** of G is a complete subgraph of G.

- The **line graph** $L(G)$ of a graph $G = (V, E)$ is the graph whose vertex set is E and whose edge set is E', where $e_1e_2 \in E'$ if and only if e_1 and e_2 are incident to the same vertex in G.

- A graph G is k-**vertex-connected** (resp. k-**edge-connected**) if we need to delete at least k vertices (resp. k edges) in order to get a non-connected graph. The (vertex or edge)-connectivity of the graph is the largest k such that G is k-connected.

1.4.2 Graph homomorphism and isomorphism

Although we have mentioned several isomorphism terms above, we would like to give the definitions formally since our approaches, presented later, frequently refer to them.

Definition 1.12 *If G and H are graphs, a homomorphism from G to H is a map $f : V(G) \to V(H)$ with the property that $f(u)$ is adjacent to $f(v)$ whenever u is adjacent to v. A bijective homomorphism whose inverse is also a homomorphism is an isomorphism.*

We write $G_1 \cong G_2$ if G_1 and G_2 are isomorphic.

Definition 1.13 *An automorphism of a graph G is an isomorphism f that maps G to itself. Formally, an automorphism of a graph G is a one-to-one, onto map $f : V(G) \to V(G)$ such that $(u, v) \in E(G) \Leftrightarrow (f(u), f(v)) \in E(G)$.*

Chapter 2

Graph Isomorphism

The GI problem has been intensively studied. It occupies an important position in the complexity family because no one knows what is its computational complexity. It is well known that GI is in NP, but despite decades of study by mathematicians and computer scientists, it is not known whether GI is in P or not [51]. There is some evidence that it is not likely to be NP-complete [90, 126]. Many researchers conjecture that GI's complexity lies somewhere between P and NP-complete. If $P \neq NP$ then, by Ladner's theorem [80], there exist problems which are of intermediate status; many people think that GI lies in this level.

One early result on the complexity of GI is an $O(exp(n^{1/2+o(1)}))$ (moderately exponential) algorithm due to Babai [9]. Moderately exponential means that on a problem of size n, the measure of computation, $m(n)$, is more than any polynomial n^k, but less than any exponential c^n, where $k > 0, c > 1$. Formally, $m(n)$ is of moderately exponential growth if for all $k > 0$, $m(n) = \Omega(n^k)$ and for all $\epsilon > 0$, $m(n) = o((1+\epsilon)^n)$.

The best existing upper bound for the problem is $exp(\sqrt{cn \cdot logn})$ (c is a constant) given

by Luks and Zemlyachenko [14], but there is no evidence of this bound being optimal. By imposing certain restrictions on the properties of the graphs, however, it is possible to design algorithms that have polynomially bounded complexity. In the following sections, we will give the complexity classes of restricted graph isomorphism problems which have been compiled by many mathematicians during the last decades.

2.1 Isomorphism-complete class

The graph isomorphism problem is not isolated. It is, in fact, a class of problems. A rigorous discussion of the structural complexity of the graph isomorphism problem is given in Köbler, Schöning, and Torán [78]. In an attempt to classify the graph isomorphism problem a new class of problems has been developed: the isomorphism-complete class [59]. A problem is said to be **isomorphism-complete** if it is provably equivalent to the isomorphism problem. This class includes problems that can be shown to be polynomially equivalent to the graph isomorphism problem.

As mentioned, the complexity of GI is polynomial if we add some restriction on the graphs. Conversely, many other restricted isomorphism problems are known to be polynomially equivalent to GI. It has been proved that the following types of graphs are in the isomorphism-complete class: bipartite graphs, line graphs [63], rooted acyclic digraphs, chordal graphs, transitively orientable graphs, regular graphs [23], directed path graphs [15], k-trees (unbounded k), and comparability graphs [103]. In 1978, Colbourn [37] proved that the question of deciding whether a graph is self-complementary, is graph isomorphism complete. In 2002, Kaibel and Schwartz [73] proved that the problem of deciding whether two (convex) polytopes are combinatorially isomorphic is graph isomorphism-complete, even for simple or simplicial polytopes. In the same year, Nagoya, Uehara and Toda [103] showed that

chordal bipartite graphs are in the isomorphism-complete class, too.

Now we review several types of graphs in the isomorphism-complete class and give brief proofs.

2.1.1 Bipartite graph isomorphism

A **bipartite** graph is a graph G whose vertex set V can be partitioned into two non empty sets V_1 and V_2 in such a way that every edge of G joins a vertex in V_1 to a vertex in V_2. An alternative way of thinking about it is as about colouring the vertices in V_1 one colour and those in V_2 another colour, with no edge between vertices of the same colour.

Theorem 2.1 *Bipartite graph isomorphism* $\equiv_p$ *graph isomorphism.*

Proof. Testing the isomorphism of bipartite graphs is isomorphism-complete, since any graph can be made bipartite by replacing each edge by two edges connected with a new vertex (see Figure 2.1).

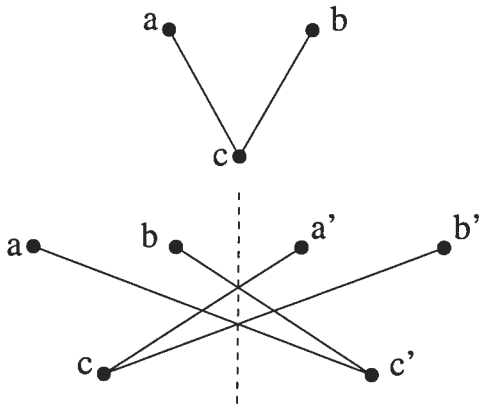

Figure 2.1: change a graph to a bipartite graph

Clearly, the original graphs are isomorphic if and only if the transformed graphs are. □

2.1.2 Chordal graph isomorphism

Definition 2.1 *[58] An undirected graph is called* ***chordal*** *if every cycle of length greater than 3 possesses a chord, that is, an edge joining two nonconsecutive vertices of the cycle.*

An example of chordal graphs is shown in Figure 2.2. Chordal graphs are also called *triangulated graphs*, *rigid-circuit graphs, monotone transitive graphs* and *perfect elimination* graphs in the literature [25].

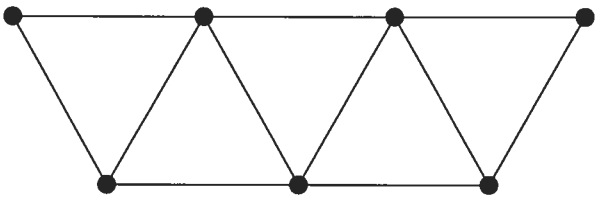

Figure 2.2: a chordal graph

Theorem 2.2 (Lueker and Booth [87], 1979)

Chordal graph isomorphism $\equiv_p$ *graph isomorphism.*

Proof. We construct a polynomial mapping M from a graph G to a graph $M(G)$ such that $M(G)$ is a chordal graph, and G can be recovered from $M(G)$ up to isomorphism. We will show that the question of whether G_1 is isomorphic to G_2 is reduced to the question of whether $M(G_1)$ is isomorphic to $M(G_2)$.

"For the reduction to chordal graph isomorphism, let $M(G) = G' = (V', E')$, where $V' = V \cup E$, and

$$E' = \{\{v, w\} \mid v \neq w, v, w \in V\} \cup \{\{v, e\} \mid v \in V, e \in E, v \in e\}.$$

The construction can be implemented to take $O(n+m)$ time, where $n = |V|$, $m = |E|$."

Now, we consider any cycle of length greater that 3 in G'. If the cycle contains only V-vertices, then it has a chord, since all V-vertices are adjacent. Otherwise, the cycle contains an E-vertex, and then the two vertices adjacent to this E-vertex must be V-vertices, thus, they are adjacent. Therefore, G' is chordal.

"Assume now that $n \geq 4$. It turns out that G' contains enough structure to allow us to reconstruct G, up to isomorphism. As a matter of fact, since all V-vertices are adjacent, all of them have degree at least equal to $n-1$, which is more than 2, while E-vertices always have degree 2, because an E-vertex is adjacent to exactly two V-vertices. Furthermore, two vertices of G are adjacent if the corresponding V-vertices are adjacent to a common E-vertex."

Thus, it is easy to see that the problem of testing isomorphism of G_1 and G_2 is then polynomially reduced to the problem of testing isomorphism of $M(G_1)$ and $M(G_2)$. □

2.1.3 Chordal bipartite graph isomorphism

Definition 2.2 *A graph is* ***chordal bipartite*** *if the graph is bipartite and every cycle of length at least* 6 *has a chord.*

We have shown that the bipartite graph isomorphism as well as chordal graph isomorphism are in the isomorphism-complete class. Naturally, we wonder in which class does chordal bipartite graph isomorphism lie. It is well-known that chordal bipartite graphs form a subclass between bipartite graphs and convex graphs. Recently, the complexity of this class of graphs was proved polynomially reducible to the general graph isomorphism too [103].

Theorem 2.3 (Nagoya, Uehara and Toda [103], 2002)

Chordal bipartite graph isomorphism $\equiv_p$ *graph isomorphism.*

Proof. (sketch) The proof is based on a construction technique.

Babel, Ponomarenko and Tinhofer show that the GI problem for directed path graphs is isomorphism-complete in [15]. They give a reduction from any given bipartite graph to a directed path graph: two given bipartite graphs are isomorphic if and only if reduced directed path graphs are isomorphic.

"Given bipartite graph $G = (X, Y, E)$ with $|X \cup Y| = n$ and $|E| = m$, the reduced directed path graph $\hat{G} = (\hat{V}, \hat{E})$ is constructed as follows (see an example Figure 2.3): $\hat{V} = X \cup Y \cup E$, and $\hat{E}$ contains:

1. $\{e, e'\}$ for e, e' in E,
2. $\{x, e\}$ for each $x \in X$ and $e \in E$ with $x \in e$,
3. $\{y, e\}$ for each $y \in Y$ and $e \in E$ with $y \in e$."

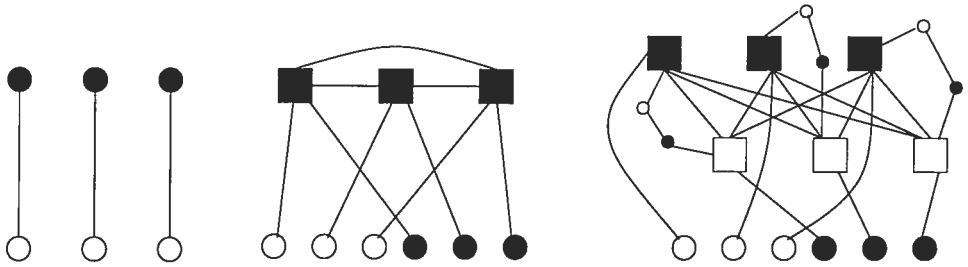

Figure 2.3: reduction of graph G, $\hat{G}$ and $\mathcal{G}$

"By this reduction, $\hat{G}$ has the following properties:

1. $\hat{G}[E]$ is a clique of size m,

2. $\hat{G}[X \cup Y]$ is an independent set of size n,

3. for each $e \in E$, e has exactly one neighbour in X, and another neighbour in Y. Thus, each vertex $e \in E$ has degree $m+1$.

Without loss of generality, we assume that $m > 1$ and $|X| > 1, |Y| > 1$."

Following this reduction, Nagoya, Uehara and Toda [103] construct a chordal bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ from the directed path graph $\hat{G} = (X \cup Y \cup E, \hat{E})$ in polynomial time. Let $\mathcal{V} = X \cup Y \cup E \cup E' \cup B \cup W$. Each vertex $e \in E$ corresponds to three vertices $e' \in E', e_b \in B$, and $e_w \in W$, respectively. That is, $|E| = |E'| = |B| = |W| = m$.

"First, we show how to connect the vertices in $E \cup E' \cup B \cup W$.

1. for each vertex $e \in E$, four edges $\{e, e'\}, \{e', e_b\}, \{e_b, e_w\}, \{e, e_w\}$ are added into $\mathcal{E}$,

2. for each pair of vertices e_1 and e_2, $\{e_1, e_2'\}, \{e_1', e_2\}$ are added into $\mathcal{E}$

Since $\hat{G}[E]$ is a clique, $\mathcal{G}[E \cup E']$ is a bipartite complete graph. In figure 2.3, black square vertices are in E, white square vertices are in E', small black vertices are in B, and small white vertices are in W. "

We recall that the vertices in $B \cup W$ are not connected to any vertices in $X \cup Y$.

"The next step in the construction is to show how to connect the vertices in X and Y to the vertices in $E \cup E' \cup B \cup W$ as in the example Figure 2.3.

1. for each vertex $x \in X$, $\{x, e\}$ is added into $\mathcal{E}$ if $\{x, e\} \in \hat{E}$,

2. for each vertex $y \in Y$, $\{y, e'\}$ is added into $\mathcal{E}$ if $\{y, e\} \in \hat{E}$.

Then, it is proved that reduced chordal bipartite graph isomorphism is polynomially equivalent to directed path graph isomorphism. Given a bipartite graph G, the reduced graph $\mathcal{G}$ has $n + 4m$ vertices and $m^2 + 5m$ edges."

Hence, the chordal bipartite graphs are in isomorphism-complete class. □

2.1.4 Self-complementary graph isomorphism

Definition 2.3 *A (di)graph G is **self-complementary** (sc) if it is isomorphic to its complement $\bar{G}$.*

There are relatively few self-complementary graphs; on twelve vertices, for instance, only 720 of the 165,091,172,592 graphs are self-complementary [113]. These are some examples (See Figure 2.4):

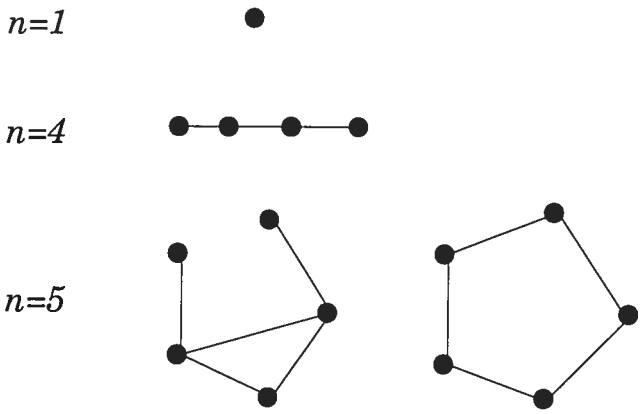

Figure 2.4: self-complementary graphs

Theorem 2.4 (Colbourn [37], 1978)

The recognition of self-complementary digraphs $\equiv_p$ graph isomorphism.

Proof. For convenience, let us consider Figure 2.5. The problem of determining the isomorphism of two graphs G and H can be polynomially reduced to the recognition of self-complementary digraphs. "By reduction, we substitute G for vertex 1 and $\bar{H}$ (graph H's complement) for vertex 2, and call the resulting digraph S. Thus, digraph S is self-complementary if and only if G and H are isomorphic."

Figure 2.5: self-complementary digraph

Now, we assume that S is self-complementary. "In one direction, since every vertex in G has out-degree at least n, whereas any vertex in $\bar{H}$ has out-degree at most $n-1$, any isomorphism carrying S into $\bar{S}$ must map G into H."

In the other direction, assume that G is isomorphic to H. "For any isomorphism f mapping G to H, we build the inverse mapping g which is an isomorphism from $\bar{H}$ into $\bar{G}$. An isomorphism from S to $\bar{S}$ is constructed by using the mapping f to map vertices from the portion of S representing G to the portion of $\bar{S}$ representing H, and using mapping g to perform the parallel mapping from $\bar{H}$ to $\bar{G}$. Thus, we can see that S is self-complementary." □

Colbourn also showed that:

Theorem 2.5 (Colbourn [37], 1978)

The recognition of self-complementary graphs $\equiv_p$ *graph isomorphism.*

Theorem 2.6 (Colbourn [37], 1978)

Self-complementary graph isomorphism $\equiv_p$ *graph isomorphism.*

Similar proofs can be found in [37].

2.1.5 Regular graph isomorphism

A graph in which every vertex has the same degree is called **regular**. If every vertex has degree k then we say the graph is regular of degree k or k-regular. Null graphs are regular of degree zero.

Theorem 2.7 (Booth [23], 1978)

Regular graph isomorphism $\equiv_p$ *graph isomorphism.*

Proof. Here, we give an outline of the proof given by Booth [23]. Since any isomorphism test for arbitrary graphs will also work for regular graphs, we need only show that graph isomorphism is polynomially reducible to regular graphs isomorphism. This proof constructs a regular graph *REGULAR(G)* from any given general graph G and proves that $G_1 \cong G_2 \Leftrightarrow REGULAR(G_1) \cong REGULAR(G_2)$.

"Let $G = (V, E)$ be any graph having $V = \{v_i \mid 1 \leq i \leq n\}$ and $E = \{e_i \mid 1 \leq j \leq m\}$ where every vertex belongs to at least one edge and $m - n > 2$. Define the following sets:

$$V_1 = \{f_i \mid 1 \leq j \leq m\},$$
$$V_2 = \{g_k \mid 1 \leq k \leq m - 2\},$$
$$V_3 = \{h_l \mid 1 \leq l \leq m - n + 2\}$$

and

$$E_1 = \{\{v_i, e_j\} \mid v_i \in e_j, 1 \leq i \leq n, 1 \leq j \leq m\},$$
$$E_2 = \{\{v_i, f_j\} \mid v_i \notin e_j, 1 \leq i \leq n, 1 \leq j \leq m\},$$
$$E_3 = \{\{e_j, g_k\} \mid 1 \leq k \leq m-2, 1 \leq j \leq m\},$$
$$E_4 = \{\{f_j, h_l\} \mid 1 \leq l \leq m-n+2, 1 \leq j \leq m\}.$$

Let $REGULAR(G)$ be the graph $(V \cup E \cup V_1 \cup V_2 \cup V_3, E_1 \cup E_2 \cup E_3 \cup E_4)$. " We can establish two facts about $REGULAR(G)$: it is a regular graph of degree m and given $REGULAR(G)$ we can recover G uniquely.

The first fact is easily verified. "Each $v_i \in V$ has degree m in $REGULAR(G)$ because it is adjacent to either e_j or f_j for all $1 \leq j \leq m$; each $e_j \in E$ has degree m because it is adjacent to exactly 2 of the $v_i \in V$ and to all $m-2$ of the $g_k \in V_2$; each $f_j \in V_1$ is adjacent to exactly $n-2$ of the $v_i \in V$ and also to all $m-n+2$ of the $h_l \in V_3$; each $g_k \in V_2$ is adjacent to all m of the $e_j \in E$; finally each $h_l \in V_3$ is adjacent to all m of the $f_j \in V_1$."

The second fact follows from the observation that in $REGULAR(G)$ every $g_k \in V_2$ has exactly the same set of neighbours and every $h_l \in V_3$ has exactly the same set of neighbours. "We can tell these two sets apart because $|V_2| > |V_3|$ since $n > 4$ if $m - n > 2$ in a graph. Having thus located V_2, we know that

$$E = \{vertices\ at\ distance\ 1\ from\ V_2\},$$
$$V = \{vertices\ at\ distance\ 2\ from\ V_2\}$$

and also that $\{u, v\} \in E$ if and only if there is an edge in $REGULAR(G)$ from both u and v to some $e_j \in E$. The encoding $(G_1, G_2) \rightarrow (REGULAR(G_1),\ REGULAR(G_2))$ thus has the property that $G_1 \cong G_2$ if and only if $REGULAR(G_1) \cong REGULAR(G_2)$. Moreover, it is clearly computable in polynomial time and hence is a polynomial reduction

of graph isomorphism to regular graph isomorphism if we realize that isolated vertices can be handled with a simple pretest and that adding an equal number of copies of K_4 to both G_1 and G_2 will not affect their isomorphism but will ensure that $m - n > 2$, without increasing the size of the input by more than a polynomial." □

2.2 Some graph isomorphism problems in P

As we mentioned before, although it seems to be hard to have a polynomial-time algorithm for general graph isomorphism, many graphs with restrictions are readily handled. For example, trees, planar graphs, graphs of bounded genus, graphs of bounded valence, graphs of bounded tree-width, graphs of bounded eigenvalue multiplicity and trivalent graphs have polynomial-time algorithms.

The first major result in this field was given by Luks [51, 88] in 1978. He showed that graphs with bounded valence can be solved in polynomial time $O(n^{ck \cdot \log k})$, where k is the bounded valence. This result was obtained by applying powerful group theory. We will give a special presentation of group theory techniques in a later section.

Other interesting graphs are interval graphs.

Definition 2.4 *A undirected graph is called interval graph if its vertices can be put into one-to-one correspondence with a set of intervals of the real line, such that two vertices are connected by an edge if and only if their corresponding intervals have nonempty intersection.*

Interval graphs can be tested for graph isomorphism in $O(mn)$, where m is the number of edges and n is the number of vertices, following results by Hsu [69] in 1995. Compared to other results, this result is very interesting in that it does not require some explicit

parameter to be fixed (constant) which is often required by many polynomial algorithms, and seems feasible to apply to a large and practical group of graphs [51].

In the following sections, several major results in the graph isomorphism problem will be shown using two approaches: combinatorial approach and group-theoretic approach. All of these results are in polynomial time at most; some of them are even in linear time or *alternating logtime (Alogtime)*. We discuss this problem in the next section.

2.3 Combinatorial approach

2.3.1 Tree isomorphism

As we have seen, a tree is a finite, connected, acyclic graph. Tree isomorphism is the basis of naïve solutions to the more general problems of subtree isomorphism, largest common subtree, and perhaps also smallest common super-tree.

Trees isomorphism has been studied since the 1970's. First, in 1974, Aho, Hopcroft and Ullman [2] gave a linear-time algorithm for tree isomorphism, based on comparing two trees in a bottom-up fashion. Certainly, linear time is the best possible sequential run time for tree isomorphism, but it is possible to consider refined algorithms, say parallel run, in smaller complexity classes [32], for instance, the class NC. In 1981, Ruzzo [116] found an NC-algorithm for solving the tree isomorphism problem for trees of logarithmic degree. Later, in 1991, Miller and Reif [100] mentioned an NC-algorithm for this problemproblem and the tree canonization problem for trees of arbitrary degree and depth. Further, Lindell [86] showed deterministic logarithmic-space algorithms for the tree isomorphism, tree comparison and tree canonization problems. Finally, in 1997, Buss [32] gave an *alternating*

logtime (Alogtime) algorithm for tree isomorphism. In this survey, we shall show the idea of Buss Alogtime algorithm.

Preliminaries and definitions

Definition 2.5 *[42] The class of languages accepted by ATMs within time $O(\log n)$ is called* ***Alogtime****.*

Definition 2.6 *An* ***immediate subtree*** *of T is a subtree whose root is a child of T's root vertex.*

Definition 2.7 *[32] Let S and T be trees. We define $S \equiv T$, called* ***tree equality****, by induction on the number of vertices in S and T by defining that $S \equiv T$ holds if and only if*

1. *$|S| = |T| = 1$ or*
2. *S and T both have the same number, m, of immediate subtrees, and there is some ordering $S_1, \cdots, S_m$ of the immediate subtrees of S and some ordering $T_1, \cdots, T_m$ of the immediate subtrees of T such that $S_i \equiv T_i, \forall i, 1 \leq i \leq m$.*

It is easy to check that $S \equiv T$ if and only if there is an isomorphism of S and T.

Definition 2.8 *Let S and T be trees. We define $S \preceq T$ and $S \prec T$, called* ***linear ordering of trees****, simultaneously by induction on the size of S and T. The linear ordering $S \preceq T$ holds if and only either $S \prec T$ or $S \equiv T$. The linear ordering $S \prec T$ holds if and only if either $|S| < |T|$ holds or the following conditions hold:*

1. $|S| = |T|$, *and*

2. *Let* $S_1, \cdots, S_m$ *be the immediate subtrees of* S *ordered so that* $S_m \preceq S_{m-1} \preceq \cdots \preceq S_1$, *and let* $T_1, \cdots, T_n$ *be the immediate subtrees of* T, *similarly ordered with* $T_{i+1} \preceq T_i$ *for all* i. *Then*

   (a) *For some* $i \leq min\{m, n\}, S_i \prec T_i$ *and for all* $1 \leq j < i, S_j \equiv T_j$, *or*

   (b) $m < n$ *and* $T_i \equiv S_i$ *for all* $1 \leq i \leq m$.

Miller and Reif [100] introduced a method to represent trees by strings over the two symbol alphabet containing open and close parentheses. The tree with a single vertex is denoted by the string "()". If T is a tree with more than one vertex, if $\alpha_1, \cdots, \alpha_m$ are strings representing the immediate subtrees, then "$(\alpha_1, \cdots, \alpha_m)$" is a string representation of tree T.

Hence, the isomorphism of trees becomes the problem of determining whether two input string representations are isomorphic.

The basic idea of the algorithm

It is known that Alogtime algorithms are capable of parsing parenthesis languages. For more information on these aspects of Alogtime, readers are advised to consult [31]. Particularly, by counting parentheses, an Alogtime algorithm can compute the depth of a vertex in a tree, can determine the i-th child of any vertex in a tree and know the ancestor/descendant predicates, etc. Also, Alogtime algorithms are capable of converting between prefix and infix notations [32].

Definition 2.9 *Let* S *be a subtree of a tree* T. *Let* $T = T_0, T_1, \cdots, T_k = S$ *be the (unique) sequence of subtrees of* T *such that each* T_{i+1} *is an immediate subtree of* T_i. *The* ***size-***

signature *of S in T is defined as the sequence $\langle |T_0|, |T_1|, \cdots, |T_k| \rangle$. If S' is a subtree of a tree T', then S and S' are* ***similar*** *provided:*

1. *They have the same size-signature, and*
2. *They are isomorphic, i.e., $S \cong S'$.*

It is easy to see that the size-signature is invariant under isomorphism. By parsing and counting techniques, there is an Alogtime procedure which, from a string representation of a tree T and a given subtree S of T, can generate the size-signature of S in T [32].

Let $\log n$ denote the logarithm (in base 2) of n rounded down to the integer. The **logsize**, $logsize(T)$, of a tree T is defined to equal $\log |T|$.

Definition 2.10 *Let T_1, T_2 be non-equal and non isomorphic trees. Let S be a subtree of T_1. We say that S distinguishes T_1 from T_2 provided that S is a proper subtree of T and:*

1. *The logsize of S is strictly less than the logsize of the parent tree of S, and*
2. *The number of subtrees of T_1 which are similar to S is not equal to the number of subtrees of T_2 which are similar to S.*

Now, we present the idea of tree isomorphism algorithm in the help of the representation of trees.

"We can view an Alogtime algorithm as a game between two players: the first player is asserting that the two trees are non-isomorphic, while the second player is asserting that the two trees are isomorphic. The input to the game consists of two string representations

of two trees T_1 and T_2, and we denote this instance of the game $G[T_1, T_2]$. The game begins with the first player identifying a subtree S_1 that distinguishes T_1 from T_2. Then the two players play a log time game to count the number of subtrees of T_1 and of T_2 which are similar to S_1. If these numbers are equal, the second player wins. Otherwise the first player wins."

Determining the truth of these assertions involves:

1. comparing the size-signatures of S and S_1, which is easily done in Alogtime, and
2. checking whether $S \cong S_1$, which involves recursive call $G[S, S_1]$.

In addition, Buss [32] showed that the entire game $G[T_1, T_2]$, including the recursive calls to the game, uses only $O(\log n)$ rounds, where n is the maximum size of T_1 and T_2. Thus, we have the following theorem.

Theorem 2.8 (Buss [32], 1997)

The tree isomorphism problem is in Alogtime.

It is obvious that the proof is essentially a formal implementation of the game described just above.

2.3.2 Planar graph isomorphism

When we draw a graph on a piece of paper, we naturally try to make it as clear as possible. One obvious way to limit the mess created by all the lines is to avoid intersections. For example, we may ask if we can draw the graph in such a way that no two edges meet in

a point other than a common end. Graphs drawn in this way are called **planar graphs**. A graph is called **outer-planar** if it can be embedded in the plane such that every vertex lies on the boundary of the same half-plane, without loss of generality on the boundary of the upper half-plane.

Planar graphs corresponding to the regular polyhedra and other geometric figures have been investigated since the time of the ancient Greeks. More recently, planar graphs appear in some applied disciplines. An example is VLSI design where one would like to design a large electric network on a planar electric board so that the connections between the components of the network do not intersect (or intersect as little as possible). Some results on planar graphs were inspired by such practical problems.

Complexity results

Planar graph isomorphism has been extensively studied during last decades. The graph isomorphism problem for triconnected (also called 3-connected) planar graphs is particularly simple since a triconnected planar graph has a unique embedding on a sphere [129]. Weinberg [128] studied this fact while developing an algorithm for testing isomorphism of triconnected planar graphs in $O(n^2)$ time. Although it is for triconnected planar graphs, this result has been extended to general planar graphs and improved to $O(n \cdot \log n)$ steps by Hopcroft and Tarjan [66, 67]. Furthermore, in 1974, a linear time $O(n)$ algorithm was found by Hopcroft and Wong [68]. In this survey, we give an overview of this approach. We intend only to establish the existence of a linear algorithm which subsequent work might make truly efficient.

The motivation

The previous work on isomorphism of planar graphs shows that, without loss of generality, we can restrict attention to determining isomorphism of embeddings of triconnected planar graphs. If G_1 and G_2 are nontrivial, triconnected planar graphs, they have a unique representation on a sphere only up to parity (that is, left or right depend on whether the graph is viewed from inside or outside the sphere). Thus one must actually test isomorphism of one planar representation of G_1 with both representations of G_2 in order to determine if G_1 and G_2 are isomorphic. Henceforth, we restrict our attention to the isomorphism of fixed embeddings of planar graphs. From now on, the word "graph" refers to a specific labeled planar representation of a planar graph.

General ideas of the algorithm

At the beginning, the algorithm assigns integer labels to vertices and pairs of integer labels to edges, one label with each end. The integer associated with a vertex is called the **vertex label**. Let edge e be incident at vertices u and v. The integer associated with the vertex u end of e is called the **u-label** of e and the integer associated with the vertex v end of e is called the **v-label** of e.

Next, the algorithm treats each graph as a simplified one by a sequence of reductions. A **reduction** of graph G is a replacement of each labeled subgraph of G of a given type by a labeled subgraph of another given type. A list of possible reductions, each having an associated priority, will be shown in detail later.

The isomorphism algorithm assigns the label 1 to each vertex and the label 2 to each edge end. Then the highest priority reduction which is applicable is applied to G_1 and G_2.

Certain discrepancies may be detected at this stage in which case the algorithm terminates and G_1 and G_2 are not isomorphic. For example, if the number of subgraphs of the type to be collapsed by the reduction differ in G_1 and G_2, then clearly the graphs are not isomorphic. The process of applying reductions continues until no further reduction is applicable. At every stage the highest priority applicable reduction is applied.

Given a type of graph, the actual process of applying a reduction will sequentially collapse all its subgraphs. Therefore, the subgraph modifications cannot interfere with each other if the result is to be order independent. Moreover, the modified labels encode sufficient information to insure that the resulting graphs are isomorphic if and only if the original graphs are isomorphic. To ensure that both graphs receive the same labels in each reduction, label assignments for each reduction are always done simultaneously for both graphs.

After each reduction, the graph is simplified because there is a strict decrease in the complexity of the graph as measured by the sum of the number of edges and vertices. The work done to achieve this decrease in the sum of the number of edges and vertices is proportional to the decrease. The fact that a triconnected planar graph has a number of edges less than three times the number of vertices insures termination of the algorithm in time which is linear in the number of vertices.

When no further reduction is applicable, the graphs are the five regular polyhedral graphs or a trivial graph consisting of a single vertex. These graphs can be tested for isomorphism (as labeled graphs) by exhaustive matching in a fixed finite time.

The reduction algorithm

As mentioned above, the algorithm treats several reductions. Here, we give a brief description for each reduction.

1. *"Removal of loops and 1-degree vertices.* The highest priority reductions involve loops, 1-degree vertices and bonds. Suppose e_1 and e_2 are incident at w and that e_1 immediately precedes e_2 in the clockwise ordering of edges at w in the planar embedding, then we write $e_2 CW_w e_1$. We write $e_1 CCW_w e_2$ to denote $e_2 CW_w e_1$.

   (a) *Removing loops.* A **loop tuple** is a triple (non-loop edge $e = (v, w)$, loop edge f, loop vertex v) such that edge e is counterclockwise adjacent to the loop f at vertex v. The **number triple** of a loop tuple (e, f, v) is the ordered triple (v-label(e), label of end of f clockwise adjacent to edge e, other label of f). The reduction consists of constructing the corresponding number triple for each loop tuple, assigning each number triple an integer, assigning the integer associated with loop tuple (e, f, v) to the v-label of e, and removing the loop f. Given a list of loop tuples for a graph G, this reduction produces a unique resultant graph G' independent of the order of the list of loop tuples. Furthermore, the algorithm can be implemented in time linear in the number of loops removed.

   (b) *Removing 1-degree vertices.* A **spoke** is a 1-degree vertex and its associated edge. Each 1-degree vertex is associated with a unique spoke. A **spoke center** has no edges incident other than spoke edges and the number of spoke edges is greater than one, then we have a **star**. If we have just one spoke, we have a **dumbbell**. With the similar manner to remove loops, this reduction also produces a unique resultant graph G' in time linear in the number of vertices."

2. *"Bond associated reductions.*

   (a) A **clump** is a maximal set of edges $e_1, \cdots, e_k$, $k > 1$, connecting two distinct vertices v and w such that (i) at least one of v and w is adjacent to a vertex other than v and w, (ii) $e_i CCW_v e_{i+1}$ and $e_i CW_w e_{i+1}$ for $1 \leq i < k$. The two vertices v and w are called **clump vertices**. During the reduction, each clump is replaced by a single edge labeled with an integer.

   (b) A **skein** is a graph consisting of two vertices u and w and k edges, $k > i$, each edge incident at both u and w. The vertices u and w are the **skein vertices** and the k edges are the **skein edges**. During the reduction, each vertex in a skein is associated with an integer. Replace each skein by a vertex labeled with the smaller of the two integers."

3. *"Four general reductions and two special cases.* Once loops, bonds and degree one vertices have been removed, Euler's theorem guarantees the existence of a degree 2, 3, 4 or 5 vertex [107]. With this in mind, we call a vertex of degree 2, 3, 4 or 5 a low degree vertex. Thus we need only insure that we can apply a reduction whenever a low degree vertex exists. The remaining reductions, in order of priority, are as follows. Note, for each case, the reduction can be implemented in linear time. For details of each reduction, readers can consult [68].

   (a) The first reduction is the replacement of all degree d vertices, all of whose neighbours are of degree other than d. This is done for $d = 2, 3, 4, 5$. At this point either the graph is a regular degree d graph or there exists a degree d vertex which is adjacent to a non-degree d vertex, $d = 2, 3, 4, 5$.

   (b) The next class of reductions collapses an edge connecting a degree d vertex with a non-degree d vertex. This also is done for $d = 2, 3, 4, 5$.

(c) The final two general classes of reductions handle graphs which are regular degree d.

(d) There are also two special reductions, involving degree four vertices."

General outline of the algorithm

Now we give the outline of the algorithm. Before each reduction application, the REDUCTION array is scanned for the first (highest priority) non-empty list of items. This can be done easily through querying an array which tells the current number of items in each list. At the same time, pointers to vertices, edge-ends, faces which are modified or whose local conditions have changed, are stored. After a subsequent pass over these pointers, items in the reduction lists are removed and added to reflect the new relationships. It is easy to see that this updating can be done in time linear in the decrease in complexity of the graph, the decrease affected by the prior reduction.

Discussion

We think this planar graph isomorphism algorithm's importance is mostly theoretical, demonstrating existence rather than providing a practical algorithm; its relative inelegance seems to suggest that "better", perhaps even practical, linear algorithms exist and that the problem is still not yet fully understood.

Recently, Gazit and Reif [57] developed a parallel planar graph isomorphism algorithm in $O(\log n)$ with $O(n^{1.5} \cdot \sqrt{\log n})$ processors with probability to fail of $1/n$ or less.

2.3.3 Convex bipartite graph isomorphism

In the isomorphism-complete section, we have proved that bipartite graph isomorphism is polynomially reducible to general graph isomorphism. Nevertheless, some subclasses of bipartite graphs, for example, convex bipartite graphs, can be tested in polynomial time for the isomorphism. The class of circular convex bipartite graphs is properly contained in the class of convex bipartite graphs, for which an $O(n^3)$ isomorphism testing algorithm using *identification matrices* method was announced by Chen [34] in 1989. In 1999, Chen [33] presented an optimal $O(n+m)$ isomorphism testing algorithm for convex bipartite graphs using the theory of *identification matrices.* Before showing the ideas of the latter algorithm, we introduce the basics of identification matrices.

Identification matrices

Definition 2.11 *A **permutation matrix** is any matrix which can be created by permuting the rows and/or columns of an identity matrix.*

In other words, a permutation matrix P is a square $(0,1)$-matrix with exactly a single 1 in each of its rows and columns so that PM is equivalent to permuting the rows of the matrix M, MP is equivalent to permuting the columns of M, and PMP^t is equivalent to permuting the rows and the corresponding columns of M (here, P^t is the transpose of P and M is an arbitrary matrix of the same size as P).

Now, we suppose that $\mathfrak{R}$ is a relation which defines a certain graph class $\mathfrak{C}$.

Definition 2.12 *Let M_1 and M_2 be two matrices representing, respectively, two graphs G_1 and G_2 of the class $\mathfrak{C}$, according to the relation $\mathfrak{R}$. Suppose G_1 and G_2 are isomorphic if*

and only if there exist two permutation matrices P_1 and P_2 such that $M_1 = P_1 M_2 P_2$. Then M_1 and M_2 are said to be ***identification matrices*** *for G_1 and G_2 of $\mathfrak{C}$, with respect to $\mathfrak{R}$.*

Lemma 2.1 *Suppose M_1 and M_2 are identification matrices for graphs G_1 and G_2, with respect to a certain relation $\mathfrak{R}$. Then two graphs are isomorphic if and only if there exists a permutation matrix P such that M_1 and M_2P have the same set of rows.*

Proof. ($\Longrightarrow$) Suppose G_1 and G_2 are isomorphic. Then there exist two permutation matrices, say P_1 and P_2, such that $M_1 = P_1 M_2 P_2$, by the definition of identification matrices. It follows that M_1 and M_2P_2 have the same set of rows.

($\Longleftarrow$) Suppose there exists a permutation matrix P such that M_1 and M_2P have the same set of rows. Then there exists another permutation matrix, say P_1, such that $M_1 = P_1 M_2 P$. It follows from the definition of identification matrix that G_1 and G_2 are isomorphic. □

Therefore, to test isomorphism of two graphs, given two identification matrices with respect to a relation, it suffices to test if, by permuting the columns, the two (resulting) matrices can have the same set of rows.

Theorem 2.9 *Adjacency matrices are identification matrices for bipartite graphs.*

Basic ideas of the algorithm

Definition 2.13 *A matrix is called an* ***augmented adjacency matrix*** *if it can be obtained from the adjacency matrix by adding 1's along the main diagonal.*

Definition 2.14 *A $(0,1)$-matrix is said to satisfy the* ***consecutive 1's property*** *for rows if the columns of the matrix can be permuted so that in the resulting matrix all the 1's*

in each of its rows are consecutive. A $(0,1)$*-matrix is said to satisfy the* ***circular*** 1***'s property*** *for rows if its columns can be permuted so that each row of the resulting matrix has circularly consecutive* 1*'s (that means, if we treat the first column and the last column as adjacent columns, these 1's are consecutive).*

Theorem 2.10 *Given two graphs represented by two identification matrices with respect to a certain relation, isomorphism can be tested in* $O(a+b+f)$ *time if at least one of the two matrices satisfies the consecutive* 1*'s property, assuming either matrix is of size* $a \times b$ *and contains* f *elements with value one.* □

Definition 2.15 *If the vertices of a bipartite graph* $G(U,V,E)$ *can be ordered so that for each element* v *in one vertex set* V*, the elements of* U *adjacent to* v *occur consecutively in* U*, then the graph* G *is a* ***convex bipartite graph****. Formally, a bipartite graph* $G(U,V,E)$ *is a convex bipartite graph if there exists an ordering* $(v_1, v_2, \cdots, v_{|V|})$ *of* V *such that, for all* $u \in U$ *and* $1 \leq i < j \leq |V|$*, if* $(u,v_i) \in E$ *and* $(u,v_j) \in E$ *then* $(u,v_k) \in E$ *for all* $i \leq k \leq j$*.*

Definition 2.16 *A connected bipartite graph* $G(U,V,E)$ *is a* ***circular convex bipartite graph****, if the vertices can be ordered such that for any vertex* u *in one vertex set, say* U*, the vertices adjacent to* u *occur circularly consecutively in* V*, the other vertex set.*

Definition 2.17 *If the vertices of a bipartite graph* $G(U,V,E)$ *can be ordered so that the* U *by* V *incidence matrix has the consecutive 1's property for both rows and columns, then the graph is called a* ***doubly convex bipartite graph****.*

Theorem 2.11 *A graph is a doubly convex bipartite graph if and only if its adjacency matrix satisfies the consecutive* 1*'s property.*

Theorem 2.12 *Isomorphism for doubly convex bipartite graphs can be tested in $O(n+m)$ time.*

Proof. "By Theorem 2.9, adjacency matrices are identification matrices for doubly convex bipartite graphs. Since the adjacency matrix for a doubly convex bipartite graph satisfies the consecutive 1's property and there are $2m$ 1-elements in one matrix, it follows from Theorem 2.10 that isomorphism for doubly convex bipartite graphs can be tested in $O(m+n)$ time." □

The following theorem can be obtained immediately from Theorem 2.10 and Theorem 2.12.

Lemma 2.2 *Isomorphism for connected convex bipartite graphs can be tested in $O(n+m)$ time.*

Now, for arbitrary convex bipartite graphs, we can test for isomorphism as follows.

"Partition a convex bipartite graph G into two parts, G_1 and G_2 with G_1 consisting of the connected components each of which is a doubly convex bipartite graph, and G_2 consisting of the rest. Let $G' = (G'_1, G'_2)$ be such a partition for another convex bipartite graph. Then G and G' are isomorphic if and only if G_1 and G'_1 are isomorphic and G_2 and G'_2 are isomorphic. Adjacency matrices are identification matrices for G_1 and G'_1 and the isomorphism can be tested in linear time by Theorem 2.12. For each of G_2 and G'_2, we partition the vertex set in such a way that the vertex incidence matrix has the consecutive 1's property. The isomorphism for G_2 and G'_2 can be tested in linear time."

As a result, we have the following theorem.

Theorem 2.13 (Chen [33], 1999)

Isomorphism for convex bipartite graphs can be tested in $O(n+m)$ time.

Obviously, the isomorphism-testing algorithm for convex bipartite graphs is optimal since the time complexity matches the trivial lower bound of $\Omega(n+m)$.

2.3.4 Bounded distance width graph isomorphism

We give a very important concept related to trees, introduced by Robertson and Seymour [115], now standard in graph theory.

Definition 2.18 *Given a graph $G=(V,E)$, we call the pair $(\{X_i \mid i \in I\}, T=(I,F))$ a **tree decomposition** of G, where I is an index set, $\{X_i \mid i \in I\}$ is a collection of subsets of V and T is a tree, such that*

1. $\bigcup_{i \in I} X_i = V(G)$,
2. *for each edge $\{v,w\} \in E$, there is an $i \in I$ such that $v,w \in X_i$,*
3. *for each $v \in V$ the set of vertices $\{i \mid v \in X_i\}$ forms a subtree of T.*

Definition 2.19 *The **width** of a tree decomposition $(\{X_i \mid i \in I\}, T=(I,F))$ equals $\max_{i \in I}(|X_i|-1)$. The **tree-width** of a graph G is the minimum width over all tree decompositions of G.*

For a given graph G and two vertices $u,v \in V(G)$, $d_G(u,v)$ denotes the distance between u and v, that is, the number of edges on a shortest path between u and v. For a set $S \subseteq V(G)$ and a vertex $w \in V(G)$, $d_G(S,w)$ denotes $\min_{v \in S} d_G(v,w)$.

Definition 2.20 *A **tree distance decomposition** of a graph $G=(V,E)$ is a triple $(\{X_i \mid i \in I\}, T=(I,F), r)$, where*

1. $r \in I$.

2. $\bigcup_{i \in I} X_i = V(G)$, *for all* $i \neq j$, $X_i \cap X_j = \varnothing$,

3. *for each* $v \in V$, *if* $v \in X_i$, *then* $d_G(X_r, v) = d_T(r, i)$,

4. *for each edge* $\{v, w\} \in E$, *there are* $i, j \in I$ *such that* $v \in X_i$, $w \in X_j$ *and either* $i = j$ *or* $\{i, j\} \in F$,

Vertex r *is called the* **root** *of the tree* T, *and* X_r *is called the* **root set** *of the tree distance decomposition. The width of a tree distance decomposition* $(\{X_i \mid i \in I\}, T, r)$ *is equal to* $\max_{i \in I} |X_i|$. *The* ***tree distance width*** *of a graph* G *is the minimum width over all possible tree distance decompositions of* G.

Definition 2.21 *A* ***rooted tree distance decomposition*** *of a graph* $G = (V, E)$ *is a tree distance decomposition* $(\{X_i \mid i \in I\}, T = (I, F), r)$ *of* G *in which* $|X_r| = 1$. *The* ***rooted tree distance width*** *of a graph* G *is the minimum width over all rooted tree distance decompositions.*

Graph isomorphism can be solved in polynomial time for graphs of bounded degree [88], tree-width, path-width, or bandwidth [130]. However, in each of these three cases, the exponent of the algorithm grows with the parameter [22]. Thus, a question is, whether algorithms exist for graph isomorphism with a running time $O(f(k)\, n^c)$, where c is small constant, k is the maximum degree (tree-with, path-with, etc.); in other words, whether graph isomorphism is *fixed parameter tractable* [44]. These questions are apparently hard. In [130], some interesting special cases of these problems are discussed; several natural graph parameters are introduced: the (rooted) path distance width, and the (rooted) tree distance width. Here, we give an overview of the (rooted) tree distance width graph isomorphism.

Basic ideas of the algorithm

Let $D = (\{X_i \mid i \in I\}, T = (I, F), r)$ be a tree distance decomposition of graph G. Tree D is **minimal** if $G[V(D, i)]$ is connected for each $i \in I$. The algorithm 1 (cited from [130]) is to find the minimal tree distance decomposition in $O(|E(G)|)$ time.

Algorithm 1 find the minimal tree distance decomposition
PROCEDURE GetTDD

INPUT: a graph $G = (V, E)$ and a root set S
OUTPUT: the minimal tree distance decomposition
$(\{X_i \mid i \in I\}, T = (I, F), r), X_r = S)$

for any $v \in V$ set $distance(v) = d_G(S, v)$;
$m := \max_{v \in V} distance(v)$;

$I := \varnothing$; $F := \varnothing$; $h := 0$;
for any $i, 0 \leq i \leq m + 1$ set $V_i = \{v \in V \mid distance(v) = i\}$;

FOR $i := m$ **DOWN TO** 0 **DO**
Compute the connected components of $G[\{v \in V \mid i \leq distance(v) \leq i + 1\}]$;
/* We call the connected components $S_1, \cdots, S_t$ */

FOR $j := 1$ **TO** t **DO**
$X_{h+j} := S_j - V_{i+1}$;

Add edges $\{v, u\}$, $v, u \in X_{h+j}$ to $E(G)$ such that
$G[X_{h+j}]$ becomes connected;

$I := I \cup \{h + j\}$;
$F := F \cup \{\{h + j, k\} \mid X_k \subset S_j \wedge k \leq h\}$;
END FOR

$h := h + t$;
END FOR

END PROCEDURE

It is proved [130] that given a graph G and a set $S \subseteq V(G)$, we can compute in $O(|E(G)|)$ time the unique minimal tree distance decomposition with root set S. Hence, $O(k \cdot n^2)$

time is needed to compute a rooted tree distance decomposition of minimum width of a graph G.

Let $D^G = (X_i^G \mid i \in I^G), T^G = (I^G, F^G), r_G)$ and $D^H = (X_i^H \mid i \in I^H), T^H = (I^H, F^H), r_H)$ be two rooted tree distance decompositions of the graphs G and H respectively. We call D^G and D^H isomorphic if there exists an isomorphism $f : V(G) \to V(H)$ from G to H and an isomorphism $g : I^G \to I^H$ from T^G to T^H such that $g(r^G) = r^H$ and for each $i \in I^G$, $x \in I_i^G$, $f(x) \in I_{g(i)}^H$.

In [130], it is showed that the algorithm takes $O((k!)^2 k^2 n^2)$ time to check if two rooted tree distance decomposition are isomorphic. Now, we give the final algorithm 2 (cited from [130]) to determine if two graphs G and H are isomorphic.

Algorithm 2 check if G and H are isomorphic

PROCEDURE RTDW_ISO(G, H)

INPUT: graphs G and H of rooted tree distance width at most k
OUTPUT: **TRUE**, if they are isomorphic; otherwise, **FALSE**

use GET-TDD to compute a minimum width rooted tree distance decomposition D^G of G with width at most k and root set consisting of an arbitrary vertex $v_G \in V(G)$;

FOR each $v_H \in H$ **DO**
    use GET_TDD to compute a rooted tree distance decomposition
    D^H of H with root set $\{v_H\}$;

    IF the width of D^H is at most k **THEN**
        IF ISO_CHECK(D^G, D^H) **THEN** return **TRUE**;

END FOR

return **FALSE**

END PROCEDURE

This algorithm has two phases. In the first phase, a rooted tree distance decomposition of minimum width is computed for G. For each vertex $v \in V$, GET_TDD is used to compute

the unique minimal rooted tree distance decomposition of G with root set $\{v\}$. Then, the decomposition D^G of smallest width, say k is selected. It is shown [130] that this phase needs $O(k \cdot n^2)$.

In the second phase, for each $w \in V(H)$, the algorithm computes the unique minimal rooted tree distance decomposition D^H of H with root set $\{w\}$. If the width of D^H equals k, then procedure ISO_CHECK(D^G,D^H) is used to test whether decomposition D^G and D^H are isomorphic.

The procedure ISO_CHECK (cited from [130]) is listed below.

Algorithm 3 ISO_CHECK procedure

PROCEDURE ISO_CHECK(D^G, D^H)

INPUT: decomposition $D^G = (\{X_i^G \mid i \in I^G\}, T^G = (I^G, F^G), r_G)$,
$D^H = (\{X_i^H \mid i \in I^H\}, T^H = (I^H, F^H), r_H)$.
OUTPUT: TRUE if D^G is isomorphic to D^H; FALSE if not.

IF T^G and T^H are not isomorphic **THEN** return FALSE;
let m be the depth of T^G

FOR $l := m$ **DOWNTO** 0 **DO**
  FOR each pair (p, q), $p \in V(T^G)$ and $q \in V(T^H)$
    such that $d_{T^G}(p, r_G) = d_{T^H}(p, r_H) = l$ **DO**
    compute $R_l^{p,q}$ using GET_IB(p, q, l);

IF $R_0^{r_G, r_H} = \varnothing$ **THEN** return FALSE;

return TRUE;

END PROCEDURE

This procedure first tests whether T^G and T^H are isomorphic. This test requires $O(n)$ time. Now, suppose T^G and T^H are isomorphic. Let m denote the maximum depth of a node in T^G (and hence in T^H since they are isomorphic). Now, for each level l, $0 \leq l \leq m$, and each pair of nodes p, q, with $p \in I^G$ and $q \in I^H$, the algorithm computes the set $R_l^{p,q}$

using sub-procedure GET_IB. The following algorithm shows the sub-procedure GET_IB (cited from [130]).

This sub-procedure computes $R_l^{p,q}$ as follows. First, it checks if $|X_p^G| = |X_q^H|$ and the number of children of p equals the number of children of q. If not, then $R_l^{p,q} = \varnothing$. Otherwise, for each bijection $f : X_p^G \to X_q^H$ from $G[X_p^G]$ to $H[X_q^H]$ that is an isomorphism, GET_IB tries to make a matching between the children of p and the children of q.

It is shown [130] that the overall complexity of ISO_CHECK is $O(k!^2k^2n^2)$. As the number of edges in T^G is $O(n)$, the running time of the second phase of algorithm RTWD_ISO is $O(k!^2k^2n^3)$. Moreover, as the number of different root sets for H is n, the total running time to check the isomorphism of graphs G and H is $O((k!)^2k^2n^3)$.

2.4 Group-theoretic approach

Recently, group theory was used effectively to solve what looks like a purely graph-theoretic problem, graph isomorphism [64]. This approach is based on group-theoretic concepts and the study of permutation groups.

Group theory can be thought of as an algebraic study of symmetry, and the lovely insight that connects the two topics is that in order to tell efficiently whether two graphs are the same it suffices to "know" the symmetries that the two graphs possess. Using this approach, efficient (as well as not-so-efficient) polynomial-time algorithms were obtained to determine whether graphs from several important classes are isomorphic. Among these classes are graphs where all vertices have bounded degree. Sometimes these algorithms are thought of as the apotheosis of this approach.

Algorithm 4 GET_IB sub-procedure

PROCEDURE GET_IB(p, q, l)

INPUT: Nodes p in T^G and q in T^H such that $d_{T^G}(r_G, p) = d_{T^H}(p, r_H) = l$.
OUTPUT: $R_l^{p,q}$.

$R_l^{p,q} := \varnothing$;
IF $|X_p^G| \neq |X_q^H|$ **THEN** return;

Count the number of children of p in T^G;
Count the number of children of q in T^H;
IF the number of children of p and q are different **THEN** return;

FOR each bijection $f : X_p^G \mapsto X_q^H$ that is an isomorphism **DO**
BEGIN
    Set $childrenP := \{\hat{p} : \hat{p}$ *is a child of* $p\}$;
    Set $childrenQ := \{\hat{q} : \hat{q}$ *is a child of* $q\}$;
    boolean found := FALSE;

    FOR each $\hat{p} \in childrenP$ **DO**
    BEGIN
        IF found **THEN** break;
        FOR each $\hat{q} \in childrenQ$ **DO**
        BEGIN
            IF found **THEN** break;
            FOR each $g \in R_{l+1}^{\hat{p},\hat{q}}$ **DO**
                IF $G[X_p^G \cup X_{\hat{p}}^G]$ and $H[X_q^H \cup X_{\hat{q}}^H]$ are isomorphic
                under the function $f \cup g$ **THEN**
                BEGIN
                    $childrenP := childrenP - \{\hat{p}\}$;
                    $childrenQ := childrenQ - \{\hat{q}\}$;
                    found := TRUE;
                    break;
                END
        END
        IF NOT found **THEN** return;
    END
$R_l^{p,q} := R_l^{p,q} \cup f$;
END

END PROCEDURE

The following sections give an overview of this approach. For details, readers may consult corresponding references.

2.4.1 Bounded eigenvalue multiplicity graph isomorphism

Consider an undirected graph X with n vertices, represented by its adjacency matrix A. Viewing A as a linear transformation in $\mathbb{R}^n$, the eigenvalues of A are the roots of the characteristic polynomial, $det(\lambda I - A)$. We say that the graph X is of eigenvalue multiplicity m if no root of the characteristic polynomial has multiplicity exceeding m. The analysis of graphs through their eigenvalues constitutes the theory of graph spectra [20, 41]. In [13], the isomorphism of graphs X and Y can be tested by an $O(n^{4m+c})$ deterministic and by an $O(n^{2m+c})$ Las Vegas algorithm, where n is the number of vertices of X and Y. The term "Las Vegas algorithm" was introduced in [10]. It means an algorithm which uses flips of a coin; its output may be NO ANSWER, but whenever an answer is reached it is correct, and for any particular input, the probability of receiving NO ANSWER is less than 1/2.

Linear algebra preliminaries

Consider an undirected graph X on n vertices represented by its adjacency matrix A. Since A is an $n \times n$, symmetric real valued matrix it has n real **eigenvalues**. Let $\{\lambda_1, \lambda_2, \cdots, \lambda_r\}$ be the set of distinct eigenvalues. Associated with the eigenvalue λ_i is the **eigenspace** S_i containing the **eigenvectors** associated with $\lambda_i : S_i = \{x \in \mathbb{R}^n \mid Ax = \lambda_i x\}$. By virtue of the symmetry of the matrix A:

1. If λ_i is an eigenvalue with multiplicity m_i then S_i has dimension m_i.

2. The direct sum $S_1 \oplus S_2 \oplus \cdots \oplus S_r = \mathbb{R}^n$.

3. If $i \neq j$ then S_i and S_j are mutually orthogonal.

Let $V = \{e_1, e_2, \cdots, e_n\}$ be the standard basis of $\mathbb{R}^n$, i.e. the unit vectors. These vectors are identified with the vertices of X as enumerated in the adjacency matrix A. Recall that the **automorphism group** of X is the set of permutations on V which preserve adjacency under isomorphism. The automorphisms of X induce orthogonal linear transformations on $\mathbb{R}^n$ by permuting the unit vectors. So the automorphism group of X may equivalently be defined as the set of permutation matrices π which commute with the adjacency matrix of $X : \pi \in Aut(X) \Leftrightarrow \pi A = A\pi$.

Tower of groups approach

In 1979, Babai introduced the "tower of groups" approach to give a polynomial-time coin tossing algorithm to decide multiplicity [10]. With this technique, Hoffmann solved the graph isomorphism for cone graphs of bounded degree [65]; later, Furst, Hopcroft and Luks applied it to trivalent graphs [53].

In general, a permutation group on n points may have as many as $n!$ elements. However, any permutation group G may be represented by a set of at most n^2 generating permutations whose closure under multiplication is equal to G [52].

We first consider the problem of determining a set of generators of the automorphism group of a graph X with eigenvalue multiplicity $\leq m$. The following theorems are proved in [13].

Theorem 2.14 *For a graph X with eigenvalue multiplicity $\leq m$, the generators of the automorphism group $Aut(X)$ can be found by an $O(n^{2m+c})$ deterministic and by an $O(n^{m+c})$ Las Vegas algorithm.*

We use the symbol $\restriction$ to mean "restricted to".

Theorem 2.15 *Let X be a graph with eigenvalue multiplicity $\leq m$. Then one can partition the vertex set of X as $C_1 + C_2 + \cdots + C_s$ in n^c time so that*

1. *each C_i is invariant under $Aut(X)$;*
2. *a permutation group $H_i \leq Sym(C_i)$ of order $|H_i| \leq n^m$ can be listed in n^{m+c} time such that $(Aut\ X) \restriction C_i \leq H_i$, $i = 1, 2, \cdots, s$.*

Let X be a graph with coloured vertices, the colour classes forming a partition $C_1 + C_2 + \cdots + C_s = V$ of the vertex set. Assume that groups $H_i \leq Sym(C_i)$ are explicitly listed for $i = 1, 2, \cdots, s$. Their direct product $H_1 \times H_2 \times \cdots \times H_s$ acts on V. The question about "graphs with restricted colour-groups" is to determine a set of generators for $G = Aut(X) \cap (H_1 \times H_2 \times \cdots \times H_s)$. This is a particular case of the "intersection of group-cylinders" problem solved in [10]. Let $N = \max\{|H_i| : i = 1, 2, \cdots, s\}$. The problem complexity is bounded by $O(N\, n^c)$.

Now, let $G^0 \geq G^1 \geq \cdots \geq G^t = \{e\}$ be a tower of groups. The elements of G^0 are encoded by words in an alphabet, and group operations are performed by an Oracle (Black-box). The question $\pi \in G^i$ is decided by an Oracle, for any $\pi \in G^0$. In addition, a set of generators of G^0 is provided. The problem is to find generators for each G^i. This problem was first formulated by [10].

An algorithm essentially due to Sims and analyzed by Furst, Hopcroft and Luks [52] solves this problem by $O(T^2)$, where

$$T = \sum_{i=1}^{t} ([G^{i-1} : G] - 1).$$

Hence, the total running time of this approach will be $O(T^2 \cdot n^2) = O(N^2 \cdot n^{c+2})$. In the problem of determining the automorphism group of a graph with no more than m-tuple eigenvalues, we have $N \leq n^m$. The running time of the entire algorithm is dominated by the "tower of groups" part. Finally, the problem can solved in $O(n^{4m+c})$ time.

The Las Vegas algorithm for the "tower of groups" for "graphs with restricted colour-groups" requires only $O(N \cdot n^c)$ time. This results in $O(n^{2m+c})$ for determining the automorphism group of a graph with no more than m-tuple eigenvalues.

2.4.2 Trivalent graph isomorphism

As mentioned above, group theory plays an important role in solving the graph isomorphism problem. Consider a graph X which is the disjoint union of X_1 and X_2, and look at its automorphism group A (elements are vertex relabellings which preserve adjacency; the group operation is composition of labellings). If the graphs are isomorphic, then A, and hence any generator set for A, has elements which map a vertex of X_1 to a vertex X_2. The converse is also true, so it suffices to compute a generator set for A. In other words, testing graph isomorphism is polynomially reducible to determining generators for the automorphism group of graphs and there are small sets of generating permutations [64]. *Trivalent* graphs, also called *cubic* graphs, are graphs all of those vertices have degree 3. Here, we consider the trivalent graph isomorphism.

Basic ideas of the algorithm

In order to test isomorphism of connected trivalent graphs X^1 and X^2 with n vertices and $O(n)$ edges, we answer $O(n)$ questions of the following form: Given $e_1 \in E(X^1)$ and

$e_2 \in E(X^2)$, is there an isomorphism from X^1 to X^2 that maps e_1 to e_2? This question is reduced to constructing a generator set [54] for $Aut_e(X)$, the group of automorphism of a connected trivalent graph X that fixes a specified edge, e. This reduction to the study of $Aut_e(X)$ was observed in [53]. In order to understand well this important reduction, we recall the proposition [88] formally and give the proof here.

Proposition 2.1 *Testing isomorphism of trivalent graphs is polynomial-time reducible to the problem of determining generators for $Aut_e(X)$, where X is a connected trivalent graph and e is a distinguished edge.*

Proof. "Assume we posses a polynomial-time algorithm which returns generators for any such $Aut_e(X)$. Once again, it suffices to be able to compare two connected trivalent graphs X^1, X^2. Fix an edge $e_1 \in E(X^1)$. For each edge $e_2 \in E(X^2)$ we can test whether there is an isomorphism from X^1 to X^2 which maps e_1 to e_2 as the following: Construct a connected trivalent graph X from the disjoint union $X_1 \cup X_2$ by

1. inserting new vertices v_1 in e_1 and v_2 in e_2,

2. joining v_1 to v_2 with a new edge e.

Then there is an isomorphism from X^1 to X^2 mapping e_1 to e_2 if and only if some element of $Aut_e(X)$ transposes v_1 to v_2. Furthermore, if such automorphism exists, any set of generators of $Aut_e(X)$ will contain one." □

The motivation to compute $Aut_e(X)$ is essentially from Tutte's observation [125] that $Aut_e(X)$ is a 2-group. The other useful feature is that there is a natural sequence of "approximations" to $Aut_e(X)$. For this, we let $X_r, r = 1, 2, \cdots, n-1$, be the subgraph of

X comprised of all vertices and edges on paths of length $\leq r$ through e (so X_1 is e and X_{n-1} is X). There are natural homomorphisms

$$\pi_r : Aut_e(X_{r+1}) \rightarrow Aut_e(X_r),$$

in which $\pi_r(\sigma)$ is the restriction of σ to X_r. In the rth stage, $r = 1, 2, \cdots$, we construct a generating set for $Aut_e(X_{r+1})$ given one for $Aut_e(X_r)$. This task can be broken into two problems:

1. Find a set of generators $\mathcal{R}$ for $Ker(\pi_r)$.

2. Find a set of generators $\mathcal{S}$ for $Im(\pi_r)$.

Thus, if $\pi_r(\mathcal{S}') = \mathcal{S}$ in $Aut_e(X_{r+1})$, then $\mathcal{R} \cup \mathcal{S}'$ generate $Aut_e(X_{r+1})$. The harder problem is the second. It is reduced to the following problem [88]:

Problem 1.
Input: A set of generators for a 2-subgroup, G, of $Sym(\mathbb{A})$, where $\mathbb{A}$ is a coloured set.
Find: A set of generators for the subgroup $\{\sigma \in G \mid \sigma \ is \ color \ preserving\}$.

To solve the colour automorphism algorithm for 2-groups, Luks [88] uses a *divide-and-conquer* strategy, the decomposition of the set into orbits. After investigating the problems [54, 88], it is shown that $Aut_e(X)$ can be computed in time $O(n^3)$, where X is an n-vertex, connected, trivalent graph.

Having the upper bound of $Aut_e(X)$, the reduction could be used to test isomorphism for n-vertex, connected, trivalent graphs X^1, X^2 in $O(n^4)$ steps. It can be done through an $Aut_e(X)$ computation corresponding to each $e_2 \in E(X^2)$. According to [54], however, an

examination of this process could include repetitive computation of the groups, blocks, etc. for $Aut_{e_1}(X^1)$.

Our ultimate goal is to compute $Iso_{e_1,e_2}(X^1, X^2)$, the set of all isomorphisms from X^1 to X^2 that take edge e_1 to edge e_2. If σ is one such isomorphism, then $Iso_{e_1,e_2}(X^1, X^2) = \sigma Aut_{e_1}(X^1)$. With this in mind, we compute $Aut_{e_1}(X^1)$, together with the blocks, groups, and search for a single representative isomorphism, if one exists. The authors show that $Iso_{e_1,e_2}(X^1, X^2) = \sigma Aut_{e_1}(X^1)$ is computable in time $O(n^2 \log n)$. Finally, isomorphism of n-vertex trivalent graphs can be tested in time $O(n^3 \log n)$. The algorithm determines the set of all isomorphisms. The Las Vegas algorithm for $Iso_{e_1,e_2}(X^1, X^2) = \sigma Aut_{e_1}(X^1)$ problem can be solved in time $O(n^2)$, thus for the isomorphism of n-vertex trivalent graphs the time is expected in time $O(n^3)$.

Discussion of the result

Can the result be generalized? The fast algorithms for 2-groups be generalized to p-groups. Although the groups that turn up in the consideration of graphs of higher valence are not p-groups, there is a sense in which they are almost p-groups [88]. One expects that this observation should lead to improvements of isomorphism testing for small valence, say 4 and 5 (where the groups are solvable).

2.4.3 Bounded valence graph isomorphism

Now, we study further the trivalent graph isomorphism problem. As discussed in the previous subsection, the problem of determining generators for $Aut(X)$ is reducible to problems of determining generators for the automorphism group of graphs and there are small sets

of generating permutations. Luks went on to show that trivalent-graph-isomorphism is in P by reducing it to a **colour automorphism** problem for 2-groups and presenting a polynomial-time solution for the latter [88]. The technique was extended to graphs of bounded valence. We recall the colour-automorphism problem here, since it is frequently investigated.

colour Automorphism Problem.
Input: A coloured set $\mathbb{A}$ and generators for a group G of permutations of $\mathbb{A}$.
Find: Generators for the subgroup consisting of the colour preserving maps.

Now, we look back on determining generators of $Aut(X)$. In the colour automorphism problem, computing $Aut(X)$ is a special case: Let G be the group of all permutations of the vertex set $V(X)$ while viewing G as an action on the set $\mathbb{A}$ of unordered pairs of vertices; colour $\mathbb{A}$ with two colours (for example, blue and red) to delimit edges and non-edges of X; then $Aut(X)$ is the colour-preserving subgroup.

We have discussed trivalent graph isomorphism. Now, we consider graphs of valence $\leq t$ where t is, henceforth, fixed. The procedure introduced in the previous subsection is still helpful and beneficial to our new situation. The reduction to determining the kernel and image of

$$\pi_r : Aut_e(X_{r+1}) \to Aut_e(X_r),$$

is the same as in the trivalent graph case.

"We consider the set $V(X_{r+1}) \setminus V(X_r)$. Let $\mathbb{A}$ denote all non-empty subsets of $V(X_r)$ of size $\leq t-1$. We define a "father-map"

$$f : V(X_{r+1}) \setminus V(X_r) \to \mathbb{A}$$

by $f(v) = \{w \in V(X_r) \mid \langle v, w\rangle \in E(X)\}$. An element $\sigma \in Aut_e(X_{r+1})$ now lies in $K_r = kernel(\pi_r)$ if and only if it stabilizes each set of "tuplets", $f^{-1}(a)$, for $a \in \mathbb{A}$. The sets $f^{-1}(a)$ form a partition of $V(X_{r+1}) \setminus V(X_r)$ and K_r is the direct product

$$K_r = \prod_{a \in \mathbb{A}} Sym(f^{-1}(a)).$$

Each of the factors in the direct product can be specified with at most two generators.

We observe next that $\sigma \in Aut_e(X_r)$ is in the image of π_r if and only if σ stabilizes, for each $0 \leq s \leq t-1$, the set of fathers of s-tuplets

$$\mathbb{A}_s = \{a \in \mathbb{A} \mid |f^{-1}(a)| = s\}$$

as well as the set $\mathbb{A}'$ of new edges. colour $\mathbb{A}$, accordingly, with $2t$ colours. The problem is once again one of finding the colour automorphism in $G = Aut_e(X_r)$ acting on $\mathbb{A}$."

With the similar idea presented in the trivalent graph case to reduce the problem, Luks introduced another concept.

Definition 2.22 *For $k \geq 2$, let Γ_k denote the class of groups G such that all the composition factors of G are subgroups of S_k.*

Then, he proves that for each r, $Aut_e(X_r) \in \Gamma_{t-1}$. Hence, testing isomorphism of graphs of bounded valence is polynomial-time reducible to the following problem with k fixed:

Problem 2.

Input: A set of generators for a subgroup, G, of $Sym(\mathbb{A})$, where $G \in \Gamma_k$ and $\mathbb{A}$ is a coloured set.

Find: A set of generators for the subgroup $\{\sigma \in G \mid \sigma \text{ is color preserving}\}$.

Solving this problem with the same divide-and-conquer strategy of trivalent graph case, Luks shows that it can be computed in polynomial time.

All in all, testing graph isomorphism of bounded valence can be solved in polynomial time. Actually, Luks gives an algorithm in complexity of $O(n^{c \cdot d \cdot \log d})$ with bounded valence d.

Chapter 3

Subgraph Isomorphism

Subgraph isomorphism is an important and very general form of exact pattern matching. In the general subgraph isomorphism problem, given a "text" G and a "pattern" H, one must either detect an occurrence of H as a subgraph of G, or list all those occurrences. For certain choices of G and H there can be exponentially many occurrences, so listing all occurrences cannot be solved in sub-exponential time. Because of reductions from Hamiltonian path and clique finding, the decision problem is NP-complete [55]. Hence, sub-exponential algorithms are unlikely. However, for any fixed pattern H with ℓ vertices, both the enumeration and decision problems can easily be solved in polynomial $O(n^\ell)$ time, and for some patterns, a better bound might be possible. Thus one is led to the problem of determining the algorithmic complexity of subgraph isomorphism for a fixed pattern.

3.1 Complexity results

Since this section is far away from the first chapter, we recall the definition of subgraph isomorphism here: given two graphs G_1 and G_2, find out if G_2 contains a subgraph that is

isomorphic to G_1, or find all such isomorphic subgraphs. Formally, the graph $G_1(V_1, E_1)$ is isomorphic to a subgraph of a graph $G_2(V_2, E_2)$, denoted by $G_1 \cong S_2 \leq G_2$, if there is an injection $\varphi : V_1 \to V_2$ such that, for every pair of vertices $v_i, v_j \in V_1$, if $(v_i, v_j) \in E_1$ then $(\varphi(v_i), \varphi(v_j)) \in E_2$.

For the general subgraph isomorphism problem, no better bound than the naïve $O(n^\ell)$ bound is known [72], where ℓ is the number of vertices in pattern H. Although the subgraph isomorphism is NP-complete, some special cases are interesting, and have polynomial-time algorithms. Shamir and Tsur [120] gave a polynomial-time $O((k^{1.5}/\log k) \cdot n)$, where k and n are the number of vertices of H and G, for subtree isomorphism. Under the assumption that the degree of some distinguished vertices is preserved under the subgraph isomorphism mapping, it was shown that the subgraph isomorphism problem is solvable in quadratic time as well [28]. While it still remains NP-complete, Eppstein [47] solved the subgraph isomorphism problem in planar graphs in linear time, for any pattern of constant size. This is the first known algorithm for this problem that is polynomial in $|G|$. Jiang and Bunke [72] also showed that embedded subgraph isomorphism can be solved in polynomial time.

It has long been known that if the pattern H is either K_3 or K_4, then there can be at most $O(n)$ instances of H as a subgraph of a planar graph G, and that these instances can be listed in linear time [17, 70, 112]. In [46], it is shown that listing all cycles in fixed length in **outer-planar** graphs can be done in linear time.

Furthermore, with the outer-planar cycle, any **wheel** with given fixed size can be found in linear time. Itai and Rodeh [70] discuss the problem of finding the girth of a general graph, or, equivalently, finding short cycles. Richards [114] gives $O(n \cdot \log n)$ algorithms for finding C_5 and C_6 subgraphs, and leaves open the question for larger cycle lengths. Bodlaender [21] discusses the related problem of finding a path or cycle longer than some given length in a general graph, which he solves in linear time for a given fixed length bound.

3.2 Subtree isomorphism

As described in the chapter on graph isomorphism, it has been proved that tree isomorphism can be tested in linear time, even Alogtime. However, for the subtree isomorphism problem, this is not the case. Polynomial-time algorithms for subtree isomorphism with tree-width at most 2 were first given by Matula [91] in 1968. Later, faster algorithms, with complexity time $O(k^{1.5}n)$, were given by Matula [92] and Chung [36]. In contrast, the subgraph isomorphism problem is NP-complete when G is a tree and H is a forest [55].

Basic notation

A **rooted tree** is a triplet $G(V, E, r)$, where (V, E) is an unrooted tree, and r is some vertex in V which is called the *root*. It is sometimes denoted as G^r. We also denote by G^r_v the rooted subtree of G^r whose vertices are all descendants of v, and its root is r. Let G^r and $H^{r'}$ be rooted trees, we write $H^{r'} \subseteq_R G^r$ if there is a rooted subtree J^r of G^r which is isomorphic to $H^{r'}$.

An $O(k^{1.5}n)$ algorithm

Based on Chung's algorithm [36], we briefly describe the idea of the $O(k^{1.5}n)$ algorithm.

Let $G(V, E)$ and $H = (V_H, E_H)$ be the input trees, and select a vertex r of G to be the root. We recall that the open neighbourhood of a vertex v in a graph G is $N(v) = \{u \mid uv \in E\}$; the closed neighbourhood is $\overline{N}(v) = N(v) \cup \{v\}$.

Lemma 3.1 *For any vertex v in G^r, vertex u in H and a vertex $w \in \overline{N}(u)$, we have that $H^w_u \subseteq_R G^r_v$ if and only if for every child u' of u in H^w_u, there is a distinct child v' of v such that $H^u_{u'} \subseteq_R G^r_{v'}$.* □

"We store this information in sets $S(v,u)$ defined as follows: for every $v \in V$, and for every $u \in V_H$,

$$S(v,u) = \{w \in \overline{N}(u) \mid H_u^w \subseteq_R G_v^r\}.$$

Notice that:

1. $u \in S(v,u)$ if and only if $H^u = H_u^u \subseteq_R G_v^r$
2. $u \in S(v,u)$ implies $S(v,u) = \overline{N}(u)$
3. $d(v) < d(u) - 1$ implies $S(v,u) = \varnothing$."

An example is illustrated in Figure 3.1. In this example, we have H^u, $H_u^{u_2} \not\subseteq_R G_v^r$ and $H_u^{u_1}, H_u^{u_3} \subseteq_R G_v^r$, so $S(v,u) = \{u_1, u_3\}$. The graph $B(v,u)$ is the bipartite graph constructed to compute $S(v,u)$. There is an edge $u_i v_j$ in this graph if and only if $u \in S(v_j, u_i)$. $H^u \not\subseteq_R G_v^r$ as $B(v,u)$ does not contain a matching of size 3. $H_u^{u_1} \subseteq_R G_v^r$ as $B_{u_1}(v,u) = B(v,u) - u_1$ contains a matching of size 2.

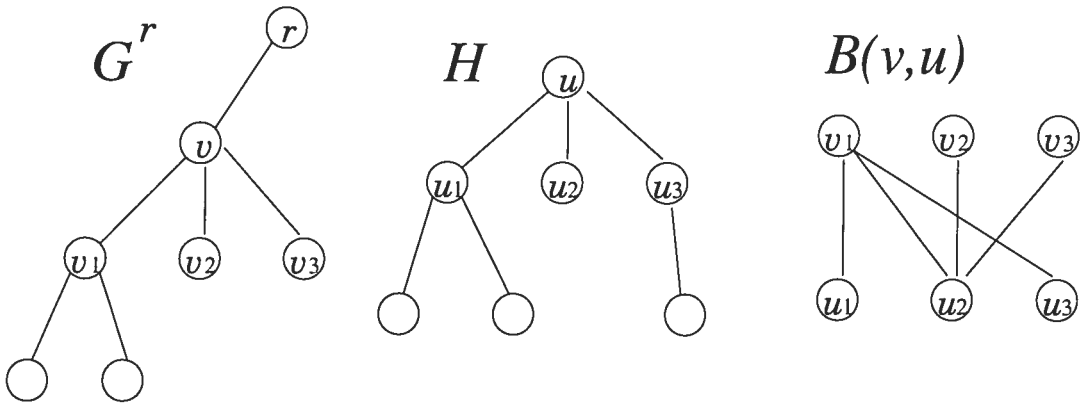

Figure 3.1: an example of subtree isomorphism

The general algorithm is described in Algorithm 5 (cited from [120]).

Algorithm 5 subtree-Isomorphism(G, H)

Select a vertex r of G to be the root of G.
FOR all $u \in H, v \in G$ **DO** $S(v, u) \leftarrow \varnothing$.

FOR all leaves v of G^r **DO**
    FOR all leaves u of H **DO** $S(v, u) \leftarrow N(u)$.

FOR all internal vertices v of G^r in a postorder **DO**
    Let $v_1, v_2, \cdots, v_t$ be the children of v.
    FOR all vertices $u = u_0$ of H with degree at most $t + 1$ **DO**
        Let $u_1, u_2, \cdots, u_s$ be the neighbors of u.
        Construct a bipartite graph $B(v, u) = (X, Y, E_{vu})$,
        where $X = \{u_1, \cdots, u_s\}, Y = \{v_1, \cdots, v_t\}$,
        and $E_{vu} = \{u_i v_j \mid u \in S(v_j, u_i)\}$.

        Denote $X_0 = X$ and $X_i = X - \{u_i\}$.
        FOR all $0 \leq i \leq s$ **DO**
            Compute the size m_i of a maximum matching
            between X_i and Y.

        $S(v, u) \leftarrow \{u_i \mid m_i = |X_i|, 0 \leq i \leq s\}$.
        IF $u \in S(v, u)$ **THEN** return YES

    END FOR
END FOR

return NO

Theorem 3.1 (Shamir and Tsur [120] , 1997)

Algorithm Subtree-Isomorphism solves the subtree isomorphism problem in $O(k^{1.5}n)$ *time and* $O(kn)$ *space.*

An $O((k^{1.5}/\log k) \cdot n)$ algorithm

In the above algorithm, we need to find a maximum matching. In 1995, Feder and Motwani [48] found an algorithm for bipartite graphs with equal-size parts. Here, this idea can be extended to general bipartite graphs [120]. Thus, an improved algorithm could be

implemented by modifying the Subtree-Isomorphism algorithm. We omit the modification here. For details, please consult [120]. Finally, we have:

Theorem 3.2 (Shamir and Tsur [120] , 1997)

The subtree isomorphism can be tested in $O((k^{1.5}/\log k) \cdot n)$ *time.*

3.3 Planar subgraph isomorphism

Now, we consider the special case in which G and H are planar graphs, a restriction naturally occurring in many applications. As mentioned in the graph isomorphism section, a planar graph is one that can be drawn on the plane in such a way that there are no "edge crossings", i.e. edges intersect only at their common vertices.

Eppstein uses a graph decomposition method similar to the one used by Baker [16] to approximate various NP-complete problems on planar graphs. Baker's method involves removing vertices from the graph, leaving a disjoint collection of subgraphs of small tree-width; in contrast, Eppstein focused on a collection of non-disjoint subgraphs of small tree-width covering the neighbourhood of every vertex.

Definitions and preliminaries

Definition 3.1 *The* ***width*** *of a tree-decomposition* (T, W) *is the number* $\max\{|W_t| - 1 : t \in V(T)\}$, *and the* ***tree-width*** $tw(G)$ *of* G *is the least width of any tree-decomposition of* G.

Before presenting the basic ideas of the algorithm, we need to introduce briefly an algorithmic technique, *dynamic programming.*

Definition 3.2 ***Dynamic programming*** *is an algorithmic technique in which an optimization problem is solved by caching subproblem solutions (memorization) rather than recomputing them.*

Dynamic programming is an efficient programming technique for solving certain combinatorial problems. It is an approach developed to solve sequential, or multi-stage, decision problems; hence, the name "dynamic" programming. The word *programming* in the name has nothing to do with writing computer programs. Mathematicians use the word to describe a set of rules which anyone can follow to solve a problem. They do not have to be written in a computer language. Dynamic programming is recursion's somewhat neglected cousin. It tends to break the original problem to sub-problems and chooses the best solution in the sub-problems, beginning from the smaller in size. For an introduction of dynamic programming, please consult [76, 104].

Basic ideas of the algorithm

First, we show a key structural property of planar graphs: if they have low diameter they also have low tree-width. Such a result was already implicit in the work of Baker [16]. With a bound on tree-width, we can use dynamic programming techniques to compute many graph properties in linear time [19, 123]. In [47], Eppstein proves that a planar graph G with diameter D has tree-width $O(D)$, and a tree-decomposition with width $O(D)$ can be found in time $O(D \cdot n)$.

Lemma 3.2 *[47] Assume we are given graph G with n vertices along with a tree decomposition of G with width w. Let S be a subset of the vertices of G, and let H be a fixed graph with at most w vertices. Then, in time $O(c^{w \cdot \log w} n)$ for some constant c, we can count all*

isomorphisms of H in G that include some vertex in S. In time $O(c^{w \cdot \log w} n + kw)$ we can list all such isomorphisms.

Proof: (sketch) The basic technique used is dynamic programming.

In the first step, by the dynamic programming technique, we apply the tree decomposition recursively in a tree T coming from the tree representation of G. Each vertex in the tree corresponds to a clique in the tree decomposition of G, and the subtree rooted at that vertex corresponds to a subgraph separated from the rest of G by the vertices in that clique.

Next, we introduce a term *partial isomorph.* "A partial isomorph at a vertex N of the tree T is an isomorphism between an induced subgraph H' of the pattern H and a subgraph of the portion of G corresponding to the subtree rooted at N. "

Then, let G' be the graph induced in G by the vertices (of G) in the vertex N (of T), together with two new additional vertices, each connected to all vertices in N. Further, each of the two additional vertices is given a self-loop. Then from any partial isomorph at N we can derive a graph homomorphism from all of H to G', which is one-to-one on vertices of N, maps the rest of H' to the first additional vertex, and maps $H - H'$ to the second additional vertex in G'. Let a *partial isomorph boundary* be such a map.

"There are $O(c_1^{w \cdot \log w})$ possible partial isomorph boundaries for a given vertex of T, for some constant c_1. For each partial isomorph boundary, in each vertex, we compute the number of partial isomorphisms which give rise to that boundary. We also compute a similar count of those partial isomorphisms involving a vertex of S. These numbers can be computed in a straightforward way from the same information at the vertex's children, by combining the $O(c_1^{w \cdot \log w})$ counts from each children in pairs of children at a time, resulting in $O(c_2^{w \cdot \log w})$ work per combined pair and $O(c^{\ell \cdot \log \ell} n)$ overall work."

At the root vertex of the tree T, we simply sum the number of isomorphisms involving S among those partial isomorph boundaries for which none of H is mapped to the second additional vertex. To recover the isomorphisms themselves, we simply return back through the tree using the already computed counts to determine which portions of the total sum came from which partial isomorphisms at each level. □

Well, we succeeded in solving the subgraph isomorphism problem quickly in graphs of bounded tree-width. We also see the subgraph of any planar graph G induced by the vertices near some particular vertex has bounded tree-width.

Now, we reconsider our original question: how to decide the subgraph isomorphism without this restriction? Can we utilize the above result? Naturally, we hope to decide the subgraph isomorphism between H and G by covering G with a collection of all such subgraphs. This involves another technique: *neighbourhood covers*, introduced by Awerbuch and Peleg [7] who used them for distributed computation: one can apply local computations in each cover rather than in the whole graph, since each neighbourhood is covered, and the computations terminate quickly since each subgraph has small diameter.

Lemma 3.3 *Let G be a planar graph. Then, we can find a collection of subgraphs G_i with the following properties:*

- *For every vertex v of G, the subgraph G' induced by the vertices of G within distance w of v is a subgraph of one of the graphs G_i.*
- *Every vertex of G is included in at most three subgraphs G_i.*
- *Every subgraph G_i has tree-width $O(w)$.*

□

We give an algorithm for the subgraph isomorphism problem based on Lemma 3.2 and 3.3.

Theorem 3.3 (Eppstein [47], 1995) *We can count the isomorphisms or induced isomorphisms of a given connected pattern H with w vertices, in a planar graph G with n vertices, in time $O(c^{w \cdot \log w} n)$. If there are k such isomorphisms, they can be listed in time $O(c^{w \cdot \log w} n + wk)$.*

Proof. (sketch) We apply Lemma 3.3, with $S = (V, G)$, to find in $O(n)$ time a set of disjoint subgraphs G_i with tree-width $O(w)$, covering the radius w neighbourhoods of all vertices in G. We choose one such subgraph G_i, let S be the vertices in G_i with covered neighbourhoods, and find all subgraph isomorphisms involving vertices in S using the algorithm of Lemma 3.2. We then remove S from all other covering subgraphs G_j so that the resulting graphs form a cover of $G - S$, and we continue to use that cover to find all remaining subgraph isomorphisms in $G - S$. □

Discussion on the technique

Actually, this technique can also be extended to other families of graphs. Eppstein shows [47] linear or quadratic algorithms for any family having a certain relation between diameter and tree-width.

3.4 Embedded subgraph isomorphism

Definitions and preliminaries

An embedded graph is a graph with a combinatorial embedding of the edges around each vertex. Formally, we have:

Definition 3.3 *An* ***embedded graph*** $G = (V, E, L)$ *is a graph* (V, E) *together with a set* $L = \{L(v)\}$ *of ordered, circular lists of edges incident to each vertex* $v \in V$.

Two embedded graphs are isomorphic if the underlying graphs are isomorphic and the isomorphism preserves and reflects not only the structure of the graphs but also their combinatorial embeddings.

Definition 3.4 *An* ***embedded subgraph isomorphism*** *of an embedded graph* $G_1 = (V_1, E_1, L_1)$ *into an embedded graph* $G_2 = (V_2, E_2, L_2)$ *is a subgraph isomorphism* $f : V_1 \to V_2$ *of* (V_1, E_1) *into* (V_2, E_2) *such that* $L_2(f(v))$ *is a cyclic rotation of* $f(L_1(v))$, *for all vertices* $v \in V_1$.

Description of Algorithms

In a finite, undirected, connected graph it is always possible to construct a cyclic directed path passing through each edge once and only once in each direction. Such a path is called *Eulerian* path. It can be constructed by traversing each edge of the corresponding bi-directed graph exactly once in each direction, which guarantees that the degree of each vertex is even. Such a traversal is called a *leftmost depth-first traversal* (LMDFS), since the edges are explored in left-to-right order (if drawn downwardly) for any vertex of the graph and, more generally, the whole graph is explored in a left-to-right fashion.

An algorithm was formulated by Trémaux and recalled by Weinberg [128] for finding a way out of a maze, that is, for the leftmost depth-first traversal of an undirected graph. "Starting with an edge traversed in one of its directions,

- When a non-visited vertex is reached, take the next (in the counter-clockwise ordering of the edges around the vertex) edge.

- When a visited vertex is reached along a non-visited edge, take the same edge but in the opposite direction.
- When a visited vertex is reached along a visited edge, take the next (in the counter-clockwise ordering of the edges around the vertex) non-visited edge, if any."

The following algorithm 6 [71, 72, 128] performs LMDFS on two embedded graphs, starting with edges e_1 of G_1 and e_2 of G_2.

Algorithm 6 LMDFS on two embedded graphs

PROCEDURE Match(G_1, G_2, e_1, e_2, M)

```
let v1 be the target of edge e1
let v2 be the target of edge e2
IF vertex v1 has been visited THEN
    IF reversal of edges e1 has been visited THEN
        let e1' be the reversal of edge e1
        let e2' be the reversal of edge e2
        let e1'' be e1'
        REPEAT
            let e1' be the cyclic successor of edge e1'
            let e2' be the cyclic successor of edge e2'
        UNTIL e1' = e1'' or edge e1' has not been visited
        IF edge e1' has been visited THEN return
        ELSE
            let e1' be the reversal of edge e1
            let e2' be the reversal of edge e2
    ELSE
        let e1' be the cyclic successor of reversal of edge e1
        let e2' be the cyclic successor of reversal of edge e2
        add(v1, v2) to vertex mapping M

    mark edge e1 and vertex v1 as visited
    Match(G1, G2, e1, e2, M)
END PROCEDURE
```

As the synchronized leftmost depth-first traversal proceeds, procedure *Match* extends a vertex mapping $M : V_1 \rightarrow V_2$ into the maximal vertex mapping representing an embedded

subgraph isomorphism of a subgraph of G_1 into G_2.

Starting with an empty mapping, algorithm 7 [71, 72, 128] finds, whenever possible, a vertex mapping $M : V_1 \rightarrow V_2$ representing an embedded subgraph isomorphism of an embedded graph G_1 into an embedded graph G_2.

Algorithm 7 embedded subgraph isomorphism

FUNCTION Embedded_Subgraph_Isomorphism(G_1, G_2, M)

```
let e1 be an edge of G1
FOR all edges e2 of G2 DO
    let M be an empty vertex mapping
    Match(G1, G2, e1, e2, M)
    let s be the size of M
     IF s = n1 THEN return TRUE
    return FALSE
END FUNCTION
```

Since the *Match* algorithm visits every edge of the embedded graphs at most once in each direction, the worst-case time complexity is $O(m_1 + m_2)$. The worst-case time complexity of the *Embedded_Subgraph_Isomorphism* algorithm is $O((m_1 + m_2) \cdot m_2)$.

The algorithms can be readily extended in order to enumerate all embedded subgraph isomorphisms.

3.5 Relational view approach

As introduced in the first chapter, most of the research on subgraph isomorphism algorithms has been based either on heuristic search techniques or on constraint satisfaction techniques. In this section, another approach to the problem of finding all subgraph isomorphisms is presented. A relational formulation of the problem by Cortadella and Valiente [40], combined with a representation of relations and graphs by Boolean functions, allows us

to handle the combinatorial explosion in the case of small pattern graphs and large target graphs by using *Binary Decision Diagrams* (BDDs), which are capable to represent large relations and graphs in small data structures.

Definitions and preliminaries

Definition 3.5 *Given two sets A and B, a* ***binary relation*** $\mathfrak{R}$ *between A and B is a subset of $A \times B$. Given a set of sets $A_1, \cdots, A_n$, an n-ary relation $\mathfrak{R}$ over $A_1, \cdots, A_n$ is a subset of $A_1 \times \cdots \times A_n$. For a binary relation we say that $x\mathfrak{R}y$ if and only if $(x, y) \in \mathfrak{R}$.*

A binary relation $\mathfrak{R}$ can be represented by a $|A| \times |B|$ Boolean matrix $M_{\mathfrak{R}}$, in which $M_{\mathfrak{R}}[x, y] = x\mathfrak{R}y$. A binary relation $\mathfrak{R}$ between a set A and itself is a subset of A^2 and can also be represented by a directed graph $G = (V, E)$, where $V = A$ and $E = \mathfrak{R}$.

Considering Boolean functions over the set $\mathbb{B} = \{0, 1\}$, an n-variable *Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{B}$. Typically we will represent Boolean functions with Boolean formulae in which the operators $+$ and $\cdot$ will denote the disjunction and conjunction respectively, the operator $\cdot$ having higher precedence than $+$ operator. For simplicity, the operator $\cdot$ will be often omitted. For example, the 3-variable Boolean function

$$f(0,0,0) = f(0,0,1) = f(0,1,1) = f(1,1,1) = 1$$

$$f(0,1,0) = f(1,0,0) = f(1,0,1) = f(1,1,0) = 0$$

can be represented by the Boolean formula

$$f(x_1, x_2, x_3) = \bar{x_1}\bar{x_2} + x_2x_3$$

Given a set A, an *encoding function* of A is an injective function $\sigma : A \rightarrow \mathbb{B}^n$. A necessary condition for σ to be injective is that $|A| \leq 2^n$. Given an encoding function σ of a set A, the *characteristic function* of A is an n-variable function χ_A defined as follows:

$$\chi_A(x) = 1 \Leftrightarrow \exists a \in A : \sigma(a) = x$$

where x is a vector of n Boolean variables, $x = (x_1, \cdots, x_n)$.

Given a binary relation $\mathfrak{R}$ between two sets A and B and two encoding functions $\sigma_A : A \rightarrow \mathbb{B}^n$ and $\sigma_B : B \rightarrow \mathbb{B}^m$, the characteristic function of $\mathfrak{R}$ is an $(n+m)$-variable Boolean function $\chi_R : \mathbb{B}^{n+m} \rightarrow \mathbb{B}$ defined as follows:

$$\chi_R(x, y) = 1 \Leftrightarrow \exists (a, b) \in \mathfrak{R} : \sigma(a) = x \wedge \sigma(b) = y$$

where $x = (x_1, \cdots, x_n)$ and $y = (y_1, \cdots, y_m)$. Characteristic functions can be trivially extended to n-ary relations. Henceforth, and for the sake of simplicity, we will use the symbols A and $\mathfrak{R}$ to represent the characteristic functions χ_A and $\chi_{\mathfrak{R}}$ respectively under some implicit encoding functions.

Binary Decision Diagrams

Binary Decision Diagrams (BDDs) have emerged as an efficient canonical form to manipulate large Boolean functions. The introduction of BDDs is relatively old [82], but only after

the recent work of Bryant [26] are they transformed into a useful tool. For a good review of BDDs we refer to [27].

Now, we give a brief review of this method. A binary decision diagram represents a Boolean function as a rooted, directed acyclic graph. As an example, we give a representation of a Boolean function $f(x_1, x_2, x_3)$ as a graph (See Figure 3.2).

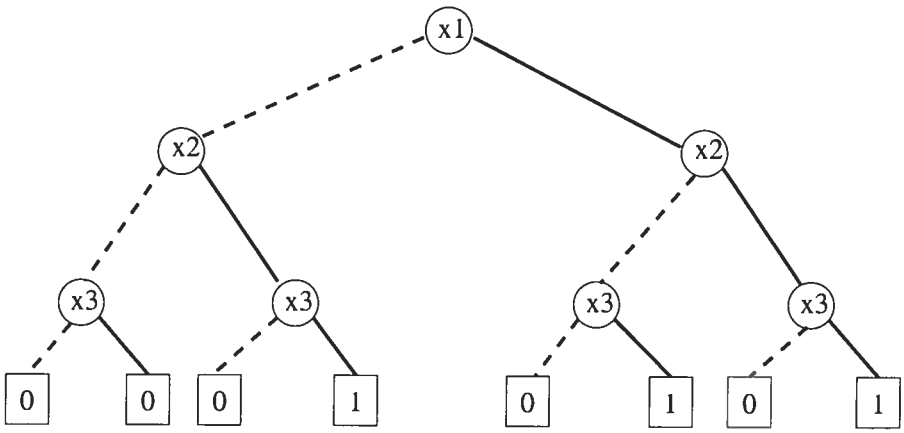

Figure 3.2: decision tree representation

Each nonterminal vertex v is labeled by a variable $var(v)$ and has arcs directed toward two children: $low(v)$ (shown as a dashed line) corresponding to the case where the variable is assigned 0, and $high(v)$ (shown as a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

"For an *Ordered* BDD (OBDD), we impose a total ordering $\prec$ over the set of variables and require that for any vertex u, and either nonterminal child v, their respective variables must be ordered $var(u) \prec var(v)$. In the decision tree of Figure 3.2, for example, the variables are ordered $x_1 \prec x_2 \prec x_3$.

Furthermore, we provide three rules to reduce the decision tree without altering the function represented:

- **Remove Duplicate Terminals:** Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.
- **Remove Duplicate Nonterminals:** If nonterminal vertices u and v have $var(u) = var(v)$, $low(u) = low(v)$, $high(u) = high(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
- **Remove Redundant Tests:** If nonterminal vertex v has $low(v) = high(v)$, then eliminate and redirect all incoming arcs to $low(v)$."

For example, Figure 3.3 illustrates the reduction of the decision tree shown in Figure 3.2 to an OBDD. Applying the first transformation rule (left graph) reduces the eight terminal vertices to two. Applying the second transformation rule (middle graph) eliminates two of the vertices having variable x_3 and arcs to terminal vertices with labels 0 (low) and 1 ($high$). Applying the third transformation rule (right graph) eliminates two vertices: one with variable x_3 and one with variable x_2. In general, the transformation rules must be applied repeatedly, since each transformation can expose new possibilities for further ones.

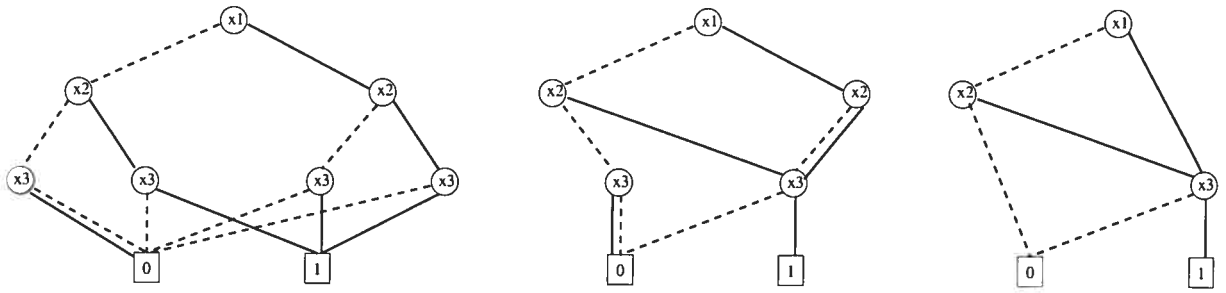

Figure 3.3: reduction of the decision tree to BDD

If a OBDD is *reduced* (no further reductions can be applied) then we have a Reduced Ordered BDD (ROBDD). Given a total ordering of variables, an ROBDD is a unique canonical form [27].

Representation of Directed Graphs

The approach to subgraph isomorphism is based on unlabeled, directed, simple graphs, also called relational graphs [119].

It is frequent to represent a graph by a Boolean adjacency matrix A, where an element $a_{ij} = 1$ if $(a_i, a_j) \in E$, and $a_{ij} = 0$ if $(a_i, a_j) \notin E$. We shall usually denote the boolean matrix A and the relation E associated to a graph $G(V, E)$ by the same name E.

Proposition 3.1 (Cortadella and Valiente [40], 2000) *The set of vertices of a graph $G = (V, E)$ with $|V| = n$ can be encoded using $\lceil \log_2 n \rceil$ variables.*

Proof. Let $k = \lceil \log_2 n \rceil$ and let $\sigma_V : V \to \mathbb{B}^k$ be a function mapping each vertex of V to a distinct k-bit Boolean vector. Then σ_V is an encoding of V, and it has $\lceil \log_2 n \rceil$ variables. □

Proposition 3.2 (Cortadella and Valiente [40], 2000) *Given an encoding $\sigma_V : V \to \mathbb{B}^k$ for the set of vertices of a graph $G = (V, E)$, where $|V| = n$ and $k = \lceil \log_2 n \rceil$, the set E of arcs of the graph can be represented by a characteristic function on $2\lceil \log_2 n \rceil$ variables.*

Proof. Let $\sigma_V : V \to \mathbb{B}^k$ be an encoding of the set of vertices V of a graph $G = (V, E)$, where $|V| = n$ and $k = \lceil \log_2 n \rceil$, and let x and y denote k-bit Boolean vectors, which will

be used to represent the source and target vertices of each arc in E. Then

$$\chi_E(x,y) = \sum_{u \in V} \sum_{v \in V, uEV} \left[\prod_{1 \le i \le k} x_i \bar{\oplus} \sigma_{V_i}(u) \right] \cdot \left[\prod_{1 \le i \le k} y_i \bar{\oplus} \sigma_{V_i}(v) \right]$$

represents the set of arcs E, and it has $2\lceil \log_2 n \rceil$ variables. □

Once an encoding function is defined on a graph, this graph can also be represented by a BDD.

Subgraph isomorphism

We now give a relational definition of subgraph isomorphism.

Definition 3.6 *A relation $\phi \subseteq V' \times V$ is an isomorphism of a graph $G' = (V', E')$ to a subgraph of a graph $G = (V, E)$ if*

$$\phi^T \phi \subseteq I, \qquad \phi \phi^T = I, \qquad E \subseteq \phi E' \phi^T$$

and it is also written $\phi : G' \to G$.

While the first two conditions assert that ϕ is an injective function from V' to V, the third condition guarantees that ϕ preserves the structure of G', that is, that it maps all arcs of G' to arcs of G.

"The set of all subgraph isomorphisms of a graph $G' = (V', E')$ into a graph $G = (V, E)$ can be represented by an n-ary relation $\mathbb{I} \subseteq V^n$, where $|V'| = n$. The subgraph isomorphism relation $\mathbb{I}$ is defined as follows:

$(v_1, \cdots, v_n) \in \mathbb{I} \Longleftrightarrow \exists$ a subgraph isomorphism $\phi : G' \to G$ such that $\phi(v'_1) = v_1, \cdots, \phi(v'_n) = v_n$.

Given two graphs $G' = (V', E')$ and $G = (V, E)$ with $|V'| = n$, let $E_{ij} \subseteq V^n$ denote the n-ary relation on V containing exactly those (pairwise disjoint) vertices of V which are joined by some arc in G, that is,

$$E_{ij} = \{(x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n) \in V^n \mid (x_i, x_j) \in E,\ x_r \neq x_s \ for\ all\ r \neq s\}."$$

The following theorem [40], shows the subgraph isomorphism relation $\mathbb{I}$.

Theorem 3.4 (Cortadella and Valiente [40], 2000)

$$\mathbb{I} = \bigcap_{(v_i', v_j') \in E'} E_{ij}.$$

Discussion

This approach is used to finding all subgraph isomorphisms of a (pattern) graph into another (target) graph. From experimental results [40], compared to Ullmann's algorithm, it has a better performance as long as the number of vertices of target graph is fairly small.

For very large target graphs, however, the approach is limited in the size of the pattern graph because BDDs representing all subgraph isomorphisms become too large. An open problem is therefore to find an optimal encoding of the target graph and optimal variable orderings in order to obtain smaller BDDs.

Chapter 4

Practical Algorithms

4.1 Review of practical algorithms

Practically, graph and subgraph isomorphism is aimed directly at developing an algorithmic procedure for isomorphism detection. Most of these algorithms are based on a state-space search with backtracking. A major improvement [50, 97] of the backtracking method was showed by Ullmann, who introduced a refinement method which reduces the search space of the backtracking procedure remarkably [127].

Another backtracking algorithm is the one presented in [118] by Schmidt and Druffel. It uses the information contained in the distance matrix representation of a graph to establish an initial partition of the graph vertices. This distance matrix information is then used in a backtracking procedure to reduce the search tree of possible mappings.

Regarding the graph isomorphism problem, it is also necessary to mention McKay's Nauty algorithm [98]. It is based on a set of transformations that reduce the graphs to be matched to a canonical form on which the testing of the isomorphism is significantly faster. Even

if Nauty is considered one of the fastest graph isomorphism algorithms available today, it has been shown that there are categories of graphs for which it requires exponential time [101].

A more recent algorithm, refereed to as VF, is based on a depth-first search strategy, with a set of rules to efficiently prune the search tree. Such rules, for the specific task of isomorphism testing, are shown in [39].

In this survey, we will show several major practical graph and subgraph isomorphism algorithms. After that, a performance comparison is given.

4.2 McKay's Nauty algorithm

This is a summary of the description of the **Nauty** algorithm as described in [98]. Nauty stands for "**N**o **AUT**omorphisms, **Y**es?". It is currently the preferred method for solving the graph isomorphism problem. Much of its strength comes from ideas of group theory.

Conceptually, Nauty looks at all the automorphisms in a graph and computes the smallest automorphism. The smallest automorphism is based upon the binary number formed by concatenating the rows of the adjacency matrix together and using the smallest such number.

Description of the algorithm

Basically Nauty uses iterative refinement to break up the partitions with an additional partitioning step. The additional step involves taking a vertex from a non-singleton

Algorithm 8 Nauty algorithm

INPUT: a graph G
OUTPUT: a canonical graph C

$\pi \longleftarrow$ the partition containing a single cell V
$S \longleftarrow$ stack containing π

WHILE (S is not empty)
  $x \longleftarrow$ pop the top of the stack S
  IF (x is a leaf partition) **THEN**
    update(C, x)
  ELSE
    refine(x)
    append the children of x to S
  END IF
END WHILE

return C

partition and putting it in its own partition and putting the remainder of the vertices in that partition into a separate partition.

The part of the algorithm that makes it the current best is its use of an indicator function A which takes a partition as input and, using eight different graph invariants, computes a hash function. The hash function hopefully computes a different value for different automorphisms.

It uses a depth-first search, invokes the indicator function at each vertex, and concatenates the result onto a sequence and uses that sequence to identify the vertex. This sequence can then be used to prune the search space. The algorithm stores the current smallest automorphism and can compare its sequence to the sequence it is currently searching. If the sequence it is searching has a sequence greater than the sequence of the smallest automorphism then we do not have to search any farther down that branch. We essentially prune large parts of the search tree in this way.

When a smaller automorphism is found, the canonical label for the graph is updated. This process continues until the smallest automorphism is found which causes the rest of the paths to not be searched since their indicator functions will result in larger values. The label is thus made up from the smallest canonical label. The algorithm is actually much more complicated and it involves ideas from group theory.

4.3 Ullmann's backtracking algorithm

In all methods mentioned, Ullmann's method [127] is considered one of the fastest algorithms for the subgraph isomorphism problem. This method is based on backtracking and a refinement procedure. The algorithm is devised for both graph isomorphism and subgraph isomorphism; it is still one of the most commonly used today for graph isomorphism.

The basic idea of the algorithm: enumeration

First, let's introduce an enumeration algorithm [127] designed to find all of the isomorphisms between a given graph $G_\alpha = (V_\alpha, E_\alpha)$ and subgraphs of a further given graph $G_\beta = (V_\beta, E_\beta)$. The numbers of vertices and edges of G_α and G_β are p_α, q_α and p_β, q_β, respectively. The adjacency matrices of G_α and G_β are $A = [a_{ij}]$ and $B = [b_{ij}]$, respectively.

Let M' be a matrix of $p_\alpha \times p_\beta$ whose elements are 1's and 0's such that each row contains exactly one 1 and no column contains more than one 1. The matrix M' can be used to permute the rows and columns of B to produce a further matrix C. Specifically, $C = [c_{ij}] = M' \times B \times (M')^t$, where t denotes transposition. If it is true that

$$\forall i, j, 1 \leq i \leq p_\alpha, 1 \leq j \leq p_\alpha, \quad (a_{ij} = 1) \implies (c_{ij} = 1), \tag{4.1}$$

then M' specifies an isomorphism between G_α and a subgraph G_β. In this case, if $m'_{ij} = 1$, then the j-th vertex in G_β corresponds to ith vertex in G_α in this isomorphism.

At the beginning of the algorithm, we construct a $p_\alpha \times p_\beta$ matrix $M^0 = [m^0_{ij}]$ in accordance with $m^0_{ij} = 1$ if the degree of the jth vertex of G_β is greater than or equal to the degree of the ith vertex of G_α; otherwise, $m^0_{ij} = 0$.

The enumeration algorithm generates all possible matrices M' such that for each and every element m'_{ij} of M', $(m'_{ji} = 1) \Rightarrow (m^0_{ji} = 1)$.

For each matrix M' the algorithm tests for isomorphism by applying condition 4.1. Matrices M' are generated by systematically changing to 0 all but one of the 1's in each of the rows of M^0, subject to the condition that no column of a matrix M' may contain more than one 1.

In the search tree, the terminal vertices are at depth $d = p_\alpha$, and they correspond to distinct matrices M'. Each nonterminal vertex at depth $d < p_\alpha$ corresponds to a distinct matrix M which differs from M^0 in that in d of the rows, all but one of the 1's has been changed to 0.

This enumeration algorithm can be improved by introducing a refinement procedure to eliminate successor nodes in the search. In [127], details of the refinement procedure are explained.

Discussion of this algorithm

This algorithm finds all isomorphisms in a time roughly proportional to p^3_α [127], and this satisfies Corneil and Gotlieb's criterion that an algorithm is efficient if the time is proportional to a power of p_α.

4.4 Schmidt and Druffel's backtracking algorithm

In this approach, Schmidt and Druffel [118] use a distance matrix method. Before giving the description of the method, we need to mention the concept of **composition** of a pair of vectors. It is defined to be the term-wise juxtaposition of their elements. In practice, this process can be performed in linear time using a list technique developed by Hopcroft and Wong [68]. We also need to talk about the degree sequence of a graph. It is merely a listing of the degrees. In terms of the adjacency matrix, the degree sequence can be generated by summing the rows and columns corresponding to each vertex. Obviously, if two graphs $G^1 \cong G^2$, then they must exhibit the same degree sequences, although the reversal is not always true.

The distance matrix

The distance matrix is a characterization of a graph which offers information on the relationship between all vertices in the given graph.

Definition 4.1 *The distance matrix D is an $n \times n$ matrix in which the element d_{ij} represents the length of the shortest path between the vertices v_i and v_j. For every pair of vertices v_i and v_j there is a unique minimum distance. If $i = j$, then $d_{ij} = 0$. If no path exists between the two vertices, the length is defined to be infinite.*

Theorem 4.1 (Hakimi and Yau [62], 1965) *If G is an n-vertex realization of D, then G is unique.*

Given a graph, generation of the distance matrix is a matter of finding the shortest distance between every pair of vertices. A number of algorithms have been developed to construct

the distance matrix, many of which are summarized by Dreyfus [45]. Floyd's algorithm [49], while of order $O(n^3)$, is simple and convenient to implement.

Initial partitioning

Since the distance matrix is a unique representation of a graph, and it contains information concerning the relationship between vertices in the graph, it offers a means of finding an initial partition which may be finer than that obtained by using the degrees of the vertices.

Definition 4.2 *A* ***row characteristic matrix*** XR *is an* $n \times (n-1)$ *matrix such that the element* xr_{im} *is the number of vertices which are a distance* m *away from* v_i.

Similarly define a ***column characteristic matrix*** XC *such that each element* xc_{im} *is the number of vertices from which* v_i *is a distance* m.

A ***characteristic matrix*** X *is formed by the term-wise juxtaposition of the appropriate elements in the corresponding rows of* XR *and* XC.

Information from the characteristic matrix may be used to form an initial partition. v_i^1 will map to v_r^2 in an isomorphism only if $x_{im}^1 = x_{rm}^2$, $\forall m$. Vertices which exhibit identical rows of the characteristic matrix will be assigned the same *class*. The partition so obtained from the characteristic matrix is often superior to that obtained from the degree sequence [118].

Theorem 4.2 *If two vertices* v_i *and* v_j *are partitioned into separate classes by the degree sequence, they will also be partitioned into separate classes by the characteristic matrix.*

Since the initial partition can be used to limit the size of the search tree, it may reduce the amount of computation that the backtracking algorithm must do. In some cases, it provides a way to determine immediately that no isomorphism exists for a given pair of vertices.

The backtracking algorithm

Definition 4.3 *The mapping of v_i^1 to v_r^2 is* ***consistent*** *if*

1. *every element $d_{ij}^1 = d_{rs}^2$ and $d_{ji}^1 = d_{sr}^2$, $\forall j, s$ such that v_j^1 has been mapped to v_s^2;*
2. *every element d_{ik}^1 (v_k^1 was not previously mapped) has a corresponding d_{rp}^2 (v_p^2 was not previously mapped) such that $c_k^1 = c_p^2$.*

Thus, a consistent mapping implies that row i and column i of D^1 have corresponding elements in row r and columns r of D^2, at least for all previously mapped vertices. The remaining elements of those rows and columns do not preclude further consistent vertex mappings if any mappings remain.

The algorithm will choose pairs of vertices v_i^1 and v_r^2 which are in the same class and will investigate the consistency of mapping v_i^1 to v_r^2. If the mapping is consistent, another pair of vertices can be chosen for mapping. If a partition is reached such that there are no consistent mappings, then a mapping between two vertices for which the mapping is not an isomorphism has been detected and it is necessary to backtrack to try another mapping.

4.5 Performance comparison

In previous sections, several major algorithms for testing graph isomorphism have been introduced. These algorithms have been devoted to improve performances both in terms of computational time and memory requirements. However, it is not clear how the behaviour of those algorithms vary as the type and the size of the graphs to be matched vary if in real applications.

Foggia, Sansone and Vento [50] have done great efforts to evaluate their performance. The comparison has been carried out on a large database of artificially generated graph. In order to do so, they built a database containing 10,000 couples of isomorphic graphs with different topologies (regular graphs, randomly connected graphs, bounded valence graph). The size of the considered graphs ranges from a few vertices to about 1,000 vertices.

There is no algorithm that is definitively better than all the others [50]. In particular, for randomly connected graphs, the Nauty algorithm is the best if the graphs are quite dense and/or of quite large size. In this case, Ullmann is faster than Schmidt and Druffel if the size of the graphs is smaller than 700 [50].

For bounded-valence graphs, the Ullmann algorithm is not always able to find a solution; if it happens, however, its time is smaller than the one of Schmidt and Druffel.

Finally, it is also worth noting that Schmidt and Druffel is always able to find a solution to the isomorphism problem in the tests, independently of the size and the kind of the graphs to be matched. As for Nauty and Ullmann algorithms, they are not.

In addition to practical algorithms above, there are other algorithms testing graph isomorphism, for example, VF and VF2 [39] which base on a depth-first search strategy, with a set of rules to efficiently prune the search tree. In [126], a new invariant, called *Probability Propagation*

Matrix, is introduced. By means of this graph invariant, a heuristic algorithm is presented. This algorithm is easy to implement and highly parallelizable.

Chapter 5

Conclusion and Discussion

5.1 Review

5.1.1 On graph isomorphism

Graph isomorphism problem belongs to those combinatorial search problem for which no polynomial-time algorithm is available yet. The two approaches usually used to test the graph isomorphism, the combinatorial approach and the group-theoretic approach, were presented. As for the former approach, trees, planar graphs, bounded average genus graphs and bounded distance width graphs are considered since we can have a polynomial-time algorithm to test their isomorphisms. As for the latter, with the help of powerful group theory, the isomorphisms of graphs with bounded valence and eigenvalue multiplicity have been shown to be computed in polynomial time.

These two approaches are very different in the way to think about the problem. The combinatorial approach focuses on the structure characteristics of special given graphs,

while group-theoretic techniques treat graphs in similar ways. With these methods, many graphs with restrictions can be tested in polynomial-time, although an efficient algorithm for general graphs is still unavailable.

From the view point of computational complexity, many scientists made efforts to reduce polynomially the general graph isomorphism to some special graphs such as bipartite graphs, regular graphs, chordal graphs, etc.

5.1.2 On subgraph isomorphism

For some restricted classes of graphs, the isomorphism can be tested in polynomial time. Many aspects of these classes have been studied. Coincidentally, the complexity of subgraph and graph isomorphism on planar graphs are both polynomial time. As a matter of fact, planar graphs are easy to be studied since they are pretty simple. Many other graph problems such as colouring, counting, become easier if we put them on planar graphs.

Of all the methods presented in this survey, I think that the tree-decomposition approach, together with the dynamic programming technique, are really special and interesting. Tree-decomposition is not the only way to decompose the problem into some sub-problems. *Modular decomposition, homogeneous decomposition, split decomposition* [25], are other techniques found in the literature.

5.1.3 On practical algorithms

Although there is still much effort to make on the theoretical complexity of the graph and subgraph isomorphism problems, many practical algorithms have been developed in the last decades. Theoretically, we consider only the problem of finding a graph or subgraph

isomorphism between two graphs at a time. However, in practical applications, we often build a graph database, so-called model graphs, and test single unknown input graph.

The algorithms reviewed in this survey are very general. By the performance comparison, we may conclude that there is no algorithm which is better than the others in all aspects. Each one of them has a good behaviour for some types of graphs. Nevertheless, the ideas used in these algorithms, for example, backtracking technique, are very beneficial for further improvement.

5.2 Look ahead

Not surprisingly, the graph and subgraph isomorphism problems are still a challenge for theoretical and practical scientists, although they have been extensively studied through many approaches and techniques. While some people intend to continue the research for untouched pieces of this area in the theoretical direction, others focus on the practical applications.

5.2.1 Turn to other graphs

As readers might have seen, not only graph isomorphism but also subgraph isomorphism are very difficult for the general case while fairly easy in some special cases, sometimes having complexity in polynomial, or even linear time. On the one hand, considering special cases helps us know more and more about these two problems as a whole. On the other hand, it also intrigues us to delve into other graph classes in that promising results might be found. Many other graphs with restrictions are worthy of looking for an efficient algorithm.

Look at an example, although chordal graphs and chordal bipartite graphs are proved to be in the isomorphism-complete class, it does not prevent us to search for a polynomial time algorithm provided that additional restrictions, for instance, k-regular, are imposed. Chordal graphs are crucially important since they represent a generalization of trees with many different rich properties, while the latter are pretty easy to be tested both for the graph and the subgraph isomorphism problems.

Since it was shown that directed graph isomorphism is polynomially reducible to undirected graph isomorphism [99], it might be helpful to use directed graphs since directed graphs have their own properties, for example, the relationships among vertices are asymmetrical. In [43], an algorithm for digraph isomorphism was given. Schmidt and Druffel also presented a fast algorithm to test directed graph isomorphism using distance matrices [118]. More recently, in 2002, this directed graph isomorphism algorithm was used to solve the topological morphing problem [74]. Generally speaking, directed graph isomorphism was not studied as widely as undirected graph isomorphism.

The isomorphism problem for Cayley graphs has been extensively investigated over the past decades. Cayley graphs are a graphic representation of abstract groups. It was shown that Cayley graphs have many nice combinatorial properties [8], such as long paths. The methods to solve the Cayley graph isomorphism range from deep group theory, including the finite simple group classification, through to combinatorial techniques. A massive research from Li about Cayley graphs can be found in [83, 84, 85].

5.2.2 Probabilistic vs. deterministic

So far, we consider only exact graph and subgraph isomorphism. In the real world, however, there is usually a certain amount of noise and distortion present in an input graph [96].

Therefore, perfect correspondences between the models and the input frequently do not exist.

A well-known algorithm for inexact graph matching, called the error-correcting subgraph isomorphism algorithm, was shown in [117, 124]. The algorithm is able to find the best matching by exploring only the most promising paths. Other methods are also proposed, such as heuristic search [29, 121], probabilistic relaxation [77] and linear programming [4]. More recently, in 2002, an optimization technique, Estimation of Distribution Algorithms (EDAs), has been used as a new approach [18] for inexact graph matching. Its foundations are based on an evolutionary computation paradigm.

Another interesting research direction is the *random graph isomorphism* which was studied by Babai, Erdös and Selkow [11] in 1980. They use a straightforward linear-time *canonical labeling* algorithm [12] that applies to almost all graphs (i.e. all but $o(2^{\binom{n}{2}})$ of the graphs on a fixed vertex set of cardinality n). They showed that in almost all graphs on n vertices, the largest $n^{0.15}$ degrees are distinct. Hence, for almost all graphs X, any graph Y can be easily tested for isomorphism of X by an extremely naïve linear time algorithm.

5.2.3 Quantum vs. classical

It was predicted that the basic units of chips in a computer would be the size of individual atoms in future. According to the theory of physics, the current (classical) theory of computation would become invalid at such level. In contrast to the classical computation theory, a new theory called *quantum computation and quantum information* has been emerging since the 1980's. The most fabulous feature of quantum computing is *quantum parallelism* and *quantum interference*. Remarkably, scientists predict that quantum computers (using quantum computing theory) can solve some hard problems breathtakingly faster

than classical computers (using classical computing theory).

In the early 1990's several scientists (Deutsch and Jozsa in 1992, Berthiaume and Brassard in 1992, Bernstein and Vazirani in 1993) sought computational tasks which could be solved by a quantum computer more efficiently than any classical computer [122]. Simon's algorithm (1993) examines an oracle problem which takes polynomial time on a quantum computer but exponential time on a classical computer. Later, Shor (1994) solved both factorization and discrete logarithms. Further, Grover (1997) discovered that searching an element in an unordered database of N elements can be improved from N to only $\sqrt{N}$ queries. A very good introduction about this theory can be found in [24, 105].

As far as the graph and subgraph isomorphism problems are concerned, until today, all of the known algorithms are classical and no quantum algorithm has been invented yet. Furthermore, even for other graph problems, the quantum approach is hard to be found in literature.

Viewing the hardness of the problems in the classical situation, we might ask quantum computation for help. An interesting idea is to apply Grover's algorithm for searching good solutions during matching procedures.

Another idea is to construct a quantum circuit with an Oracle. An Oracle is a black box, which could perform a certain kind of computation. We may assign a state to every vertex in a given graph. The state can be defined according to the characteristics of the graph. The Oracle would accept a set of certain vertices as the input and generate another set of vertices as the output. During each query to the Oracle, only the vertices which are probably more interesting to our expected result than others are strengthened. After certain queries, we might know whether two graphs are isomorphic by comparing the output of each Oracle.

Take as an example for a k-regular oriented graph. It has been proved that each k-regular oriented graph can be decomposed into k 1-regular oriented graphs or k permutations on its labeled vertices, say $[1, 2, \cdots, n]$. We construct an Oracle that accepts a vertex with its permutation index, that is $[1, 2, \cdots, k]$, as the input and transposes it to its next vertex in its permutation. All the vertices can be put in a superposition which is special in quantum computation; only one query of Oracle is necessary to transpose them to their neighbours. For each graph, there is a quantum circuit representing it. We can repeat the query of two circuits again and again until it can be determined with certain probability whether the two graphs are isomorphic. Although there are still other related problems, for instance, the permutation set can be different even for the same graph, it is quite worth considering how to construct another more hopeful quantum circuit. Other special graphs, such as trees, chordal graphs, bipartite graphs, are also suitable to construct an Oracle for querying.

In contrast to conventional graph representation, a novel representation related to quantum physics was shown in [132]: quantum graph. A quantum point is a vector in a complex Hilbert space whose basis vectors correspond to the vertices of a classical graph. A quantum arrow is thought of as an operator which destroys vertex v and creates u. A quantum arrow may be represented as a complex matrix. All other classical concepts are dramatically changed in quantum manner. Under quantum representation, the graph isomorphism problem is much more different than that under classical one. It is possible that a novel approach would emerge in future.

In short, until now, there has not been enough research between quantum computation and graph isomorphism problem. Two aspects can be considered: one can apply an existing quantum algorithm to improve the crucial part of classical algorithm; one can also construct a quantum circuit directly using special properties of the graph. Furthermore, an essentially different representation in quantum manner for a classical graph may also be interesting.

5.3 A propos de this survey

In short, this survey involves diverse theories and approaches. The results mentioned range from the very emergence of the graph and subgraph isomorphism problems to today. I included different typical sub-problems along with different approaches in order that readers have a chance to enjoy the beauty of intersections among them. For each sub-problem, the best complexity result till now was emphasized and presented specifically, without losing its main idea, integrity and continuity, I hope, while keeping an eye on relative approaches and results.

As it might have already been seen, the problems involve some related mathematical disciplines such as group theory, topological graph theory, computational complexity, linear algebra and so on. To enjoy the brilliant ideas, a solid mathematical background is necessary. All in all, I do hope that readers have found it understandable, comprehensive, informative and up-to-date.

Bibliography

[1] M. Agarwal, N. Saxena and N. Kayal, PRIMES is in P, Preprint, August 6, 2002.

[2] A.V. Aho, J.E. Hopcroft, and J. D. Ullman. The design and analysis of computer algorithms, Addison-Wesley, 1974.

[3] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms and Applications, Prentice-Hall, 1993.

[4] H.A. Almohamad and S.O. Duffuaa, A linear programming approach for the weighted graph matching problem, *IEEE Transaction on Pattern Analysis and Machine Intelligence PAMI,* 5, pp.522-525, 1993.

[5] K. Appel and W. Haken, Every planar map is four colourable. *Part I. Discharging,* Illinois Journal of Math. 21, 429-490, 1977.

[6] M. Aschbacher, Finite Group Theory, Second Edition, Cambridge University Press, 2000.

[7] B. Awerbuch and D. Peleg, Sparse partitions, *In Proceedings of the 31st IEEE Symposium Foundations of Computer Science*, pp.503-513, 1990.

[8] L. Babai, Automorphism groups, isomorphism, reconstruction, *In Handbook of Combinatorics,* Vol. 2, pp.1447-1540, Elsevier, Amsterdam, 1995.

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

[10] L. Babai, Monte Carlo algorithms in graph isomorphism Testing, *Technique Report, Université de Montréal, DMS 79-10,* pp.42, 1979.

[11] L. Babai, P. Erdös, and S. Selkow, Random graph isomorphism, *SIAM Journal on Computing*, 9, pp.628-635, 1980.

[12] L. Babai and L. Kučera, Canonical labelling of graphs in linear average time, *the 20th Annual IEEE Symposium on Foundations of Computer Science,* pp.39-46, 1979.

[13] L. Babai, D.Yu. Grigoryev and D.M. Mount, Isomorphism of graphs with bounded eigenvalue multiplicity, *Proceedings of 14th ACM Symposium on Theory of Computing*, pp.310-324, 1982.

[14] L. Babai and E. Luks, Canonical labeling of graphs, In *Proceedings of 15th ACM symposium on Theory of Computing,* pp.171-183, 1983.

[15] L. Babel, I.N. Ponomarenko, and G. Tinhofer, The isomorphism problem for directed path graphs and for rooted directed path graphs, *Journal of Algorithms*, Vol. 21, No. 3, pp.542-564, November 1996.

[16] B.S. Baker, Approximation algorithms for NP-complete problems on planar graphs, Journal of the ACM, Vol.41, No.1, pp.153-180, 1994.

[17] R. Bar-Yehuda and S. Even, On approximating a vertex cover for planar graphs, *In Proceedings of 14th ACM Symposium on Theory of Computing*, pp.303-309, 1982.

[18] E. Bengoetxea, P. Larranaga, I. Bloch, A. Perchant, and C. Boeres, Inexact graph matching by means of estimation distribution algorithms, *Pattern Recognition,* 35, pp.2867-2880, 2002.

[19] M.W. Bern, E.L. Lawler, and A.L. Wong, Linear-time computation of optimal subgraphs of decomposable graphs, *Journal of Algorithms*, 8, pp.216-235, 1987.

[20] N. Biggs, Algebraic Graph Theory, Cambridge University Press, 1993.

[21] H.L. Bodlaender, On linear time minor tests with depth-first search, *Journal of Algorithms*, 14, pp.1-23, 1993.

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

[23] K.S. Booth, Isomorphism testing for graphs, semi-graphs, and finite automata are polynomial equivalent problems, *SIAM Journal on Computing,* Vol. 7, No. 3, pp.273-279, 1978.

[24] D. Bouwmeester, A. Ekert and A. Zillinger, The physics of quantum information, quantum cryptography, quantum teleportation, quantum computation, Springer-Verlag, 2001.

[25] A. Brandstadt, V. Le and J. Spinrad, Graph classes: A survey, *SIAM Monographs on Discrete Mathematics and Applications,* 1999.

[26] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Transaction on Computers*, 35, pp.677-691, 1986.

[27] R.E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, *ACM Computing Surveys*, Vol. 24, No. 3, pp.293-318, 1992.

[28] H. Bunke, Graph matching: Theoretical foundations, algorithms, and applications, *In Proceedings of Vision Interface 2000*, pp.82-88, Montreal, 2000.

[29] H. Bunke and G. Allerman, Inexact graph matching for structural pattern recognition, *Pattern Recognition Letters 1,* 4, pp.245-253, 1983.

[30] R. Busacker and T. Saaty, Finite Graphs and Networks - An Introduction With Applications, *McGraw-Hill,* New York, pp.196-199, 1965.

[31] S.R. Buss, The Boolean formula value problem is in ALOGTIME, *In Proceedings of the 19-th Annual ACM Symposium on Theory of Computing*, pp.123-131, May 1987.

[32] S.R. Buss, Alogtime algorithms for tree Isomorphism, comparison, and canonization, *Kurt Gödel Colloquium,* pp.18-33, 1997.

[33] L. Chen, Graph Isomorphism and Identification Matrices: Sequential Algorithms, *Journal of Computer and System Sciences*, 59, pp.450-475, 1999.

[34] L. Chen, Testing Isomorphism for Transformable Convex Bipartite Graphs in Polynomial Time, Technical Report, OSU-CISRC-12 89-TR54, Department of Computer and Information Science, College of Engineering, The Ohio State University, December 1989.

[35] J.K. Cheng and T.S. Huang, A subgraph isomorphism algorithm using resolution, *Pattern Recognition*, 13, pp.371-379, 1981.

[36] M.J. Chung, $O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees, Journal of Algorithms, Vol. 8, No. 1, pp.106-112, 1987.

[37] M.J. Colbourn and J. Colbourn, Graph isomorphism and self-complementary graphs, *SIGACT News,* 10, pp.25-30, 1978.

[38] R. Cole, Parallel merge sort. In *27th Annual Symposium on Foundation of Computer Science, IEEE,* pp.511-516, October 1986.

[39] L.P. Cordella, P. Foggia, C. Sansone and M. Vento, Evaluating performance of the VF graph matching algorithm, *Proceedings of the 10th International Conference on Image Analysis and Processing,* IEEE Computer Society Press, pp.1172-1177, 1999.

[40] J. Cortadella and G. Valiente, A Relational view of subgraph isomorphism, *RelMiCS*, pp.45-54, 2000.

[41] D.M. Cvetkovic, M. Doob, and H. Sachs, Spectra of Graphs, Academic Press, 1979.

[42] C. Damm, M. Holzer and P. Rossmanith, Expressing uniformity via oracles, *Theory of Computing Systems,* 30, pp.355-366, 1997.

[43] N. Deo, J.M. Davis, and R.E. Lord, A new algorithm for digraph isomorphism, BIT, 17, pp.1630, 1977.

[44] R.G. Downey and M.R. Fellows, Fixed-parameter tractability and completeness I: Basic Results, *SIAM Journal of Computing*, 24, pp.873-921, 1995.

[45] S.E. Dreyfus, An appraisal of some shortest path algorithms, *Operations Research*, 17, pp.395-412, 1969.

[46] D. Eppstein, Connectivity, graph minors, and subgraph multiplicity, *Journal of Graph Theory*, 17, pp.409-416, 1993.

[47] D. Eppstein, Subgraph isomorphism in planar graphs and related problems, *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pp.632-640, January 22-24, 1995, San Francisco, California, United States.

[48] T. Feder and R. Motwani, Clique partitions, graph compression and speeding up algorithms, *Journal of Computer and System Sciences*, 51, pp.261-272, 1995.

[49] R.W. Floyd, Algorithm 97: Shortest path, *Communications of the ACM*, 5, pp.345, 1962.

[50] P. Foggia, C. Sansone and M. Vento, A performance comparison of five algorithms for graph isomorphism, *Proceedings of the 3rd IAPR-TC-15 International Workshop on Graph-based Representation*, Italy, 2001.

[51] S. Fortin, The graph isomorphism problem, Technical Report 96-20, University of Alberta, Edmonton, Alberta, Canada, July 1996.

[52] M. Furst, J. Hopcroft and E.M. Luks, Polynomial time algorithms for permutation groups, *Proceedings of 21st IEEE Symposium on Foundations of Computer Science,* pp.36-41, 1980.

[53] M. Furst, J. Hopcroft and E.M. Luks, A sub-exponential algorithm for trivalent graph isomorphism, *Technical Report*, 80-426, Computer Science Department, Cornell University, 1980.

[54] Z. Galil, C.M. Hoffmann, E.M. Luks, C.P. Schnorr, and A. Weber, An $O(n^3 logn)$ deterministic and an $O(n^3)$ Las Vegas isomorphism test for trivalent graphs, *Journal of the ACM*, Vol. 34, No.3, pp.513-531, July 1987.

[55] M.R. Garey and D.S. Johnson, Computers and intractability: A guide to NP-completeness, Freeman, 1979.

[56] H. Gazit, G.L. Miller, and S.H. Teng, Optimal tree contraction in the EREW model. In S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, editors, Concurrent

Computations, *Algorithms, Architecture and Technology*, pp.139-156. Plenum Press, 1988.

[57] H. Gazit and J.H. Reif, A randomized parallel algorithm for planar graph isomorphism. *Journal* of *Algorithms,* Vol.28, No.2, pp.290-314, 1998.

[58] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, 1980.

[59] G. Grigoras, On the isomorphism-complete problems and polynomial time isomorphism. *Acta Cybernetica,* 5, pp.135-142, 1981.

[60] M. Grohe, Isomorphism testing for embeddable graphs through definability, Proceedings *of the thirty-second annual ACM symposium on Theory of computing*, pp.63-72, Portland, Oregon, United States, May 21-23, 2000.

[61] J.L. Gross and M.L. Furst, Hierarchy for imbedding-distribution invariants of a graph, *Journal of Graph Theory,* 11, pp.205-220, 1987.

[62] S.L. Hakimi and S.S. Yau, Distance matrix of a graph and its reliability, *Quart of Applied Mathematics,* XXII, 4, pp.305-317, 1965.

[63] F. Harary, Graph Theory, *Addison-Wesley*, Reading, Mass, 1969.

[64] C.M. Hoffmann, Group-theoretic algorithms and graph isomorphism, *Lecture Notes in Computer Science*, ed. by G. Goos and J. Hartmanis, Vol. 136, Springer-Verlag, 1982.

[65] C.M. Hoffmann, Testing isomorphism of cone graphs, *In Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing,* pp.244-251, Los Angeles, California, 28-30 April 1980.

[66] J.E. Hopcroft and R.E. Tarjan, A $VlogV$ algorithm for isomorphism of triconnected planar graphs, *Journal of Computer and System Sciences,* Vol. 7, pp.323-331, 1973.

[67] J.E. Hopcroft and R.E. Tarjan, Isomorphism of planar graphs (working paper), in *Complexity of Computer Computations,* (R.E. Miller and J.W. Thatcher (eds.)), Plenum Press, pp.143-150, New York, 1972.

[68] J.E. Hopcroft and J.K. Wong, Linear time algorithm for isomorphism of planar graphs, *Proceedings of 6th Annual ACM Symposium on Theory of Computing,* pp.172-184. 1974.

[69] W. Hsu, $O(MN)$ algorithms for the recognition and isomorphism problem on circular-arc graphs, *SIAM Journal of Computing,* Vol. 24, No. 3, pp.411-439, June 1995.

[70] A. Itai and M. Rodeh, Finding a minimum circuit in a graph, *SIAM, Journal on Computing*, 7, pp.413-423, 1978.

[71] X.Y. Jiang and H. Bunke, Marked subgraph isomorphism of ordered graphs, *In Advances in Pattern Recognition*, Vol. 1451, *Lecture Notes in Computer Science*, pp.122-131, Springer-Verlag, 1998.

[72] X.Y. Jiang and H. Bunke, Optimal quadratic-time isomorphism of ordered graphs, *Pattern Recognition*, Vol. 32, No.7, pp.1273-1283, 1999.

[73] V. Kaibel and A. Schwartz, On the complexity of Polytope Isomorphism problem (To appear), *Graphs and Combinatorics,* July, 2002.

[74] P. Kanongchaiyos, T. Nishita, Y. Shinagawa and T. Kunii, Topological Morphing Using Reeb Graphs, *Cyber Worlds,* 11, pp.465-471, 2002.

[75] R.M. Karp, Reducibility among combinatorial problems, *Complexity of Computer Computations* (R.E. Miller and J. W. Thatcher, Eds.), Plenum, New York, 1972.

[76] A. Kaufmann, Graphs, Dynamic Programming and Finite Games, Academic Press, New York, 1967.

[77] J. Kittler, W.J. Christmas, and M. Petrou, Probabilistic relaxation for matching of symbolic structures, *Advances in Structural and Syntactic Pattern Recognition,* pp.471-480, World Scientific, 1992.

[78] J. Köbler, U. Schöning, and J. Torán, The Graph Isomorphism Problem: Its Structured Complexity, Birkhäuser, Boston, 1993.

[79] B. Krena, The Graph Isomorphism Problem, *Proceedings of 7th Conference Student FEI 2001*, pp.343–347, 2001.

[80] R. Ladner, On the Structure of polynomial-time reducibility, *Journal of the Association for Computing Machinery,* Vol.22, 155-171, 1975.

[81] W. Ledermann, Introduction to the Theory of Finite Groups, Oliver and Boyd, 1967.

[82] C.Y. Lee, Representation of switching circuits by binary-decision programs, *Bell System Technical Journal*, Vol. 38, No. 4, pp.985-999, 1959.

[83] C.H. Li, Isomorphisms of connected Cayley digraphs, *Graphs and Combinatorics*, 14, pp.37-44, 1998.

[84] C.H. Li, On Isomorphisms of finite Cayley graphs – a survey, *Discrete Mathematics*, 256, pp.301-334, 2002.

[85] C.H. Li, C.E. Praeger, and M.Y. Xu, Isomorphisms of finite Cayley digraphs of bounded valency, *Journal of Combinatorial Theory, Series B*, 73, pp.164-183, 1998.

[86] S. Lindell, A logspace algorithm for tree canonization, *In Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp.400-404, 1992.

[87] G.S. Lueker and K.S. Booth, A linear time algorithm for deciding interval graph isomorphism, *Journal of the ACM,* 26, pp.183-195, 1979.

[88] E. Luks, Isomorphism of graphs of bounded valence can be tested in polynomial time, *Journal of Computer and System Sciences,* 25, pp.42-65, 1982.

[89] M.F. Lynch, A storage and retrieval of information on chemical structures by computer, *Endeavour,* Vol. 27, No. 2, pp.68-73, 1968.

[90] R. Mathon, A note on the graph isomorphism counting problem, *Information Processing Letters,* Vol. 8, No. 3, pp.131-132, 1979.

[91] D.W. Matula, An algorithm for subtree identification, *SIAM Review*, 10, pp.273-274, 1968.

[92] D.W. Matula, Subtree isomorphism in $O(n^{5/2})$, *Annals of Discrete Mathematics,* Vol. 2, No. 1, pp.91-106, 1978.

[93] J.J. McGregor, Relational consistency algorithms and their application in finding subgraph and graph isomorphisms, *Information Sciences*, 19, pp.229-250, 1979.

[94] B.T. Messmer, Efficient graph matching algorithms for preprocessed model graphs, Ph.D Thesis, Institute of Computer Science and Applied Mathematics, University of Bern, 1996.

[95] B.T. Messmer and H. Bunke, A new method for efficient error-correcting subgraph isomorphism, *Syntactic and Structural Pattern Recognition,* World Scientific Publish Company, Singapore, 1995.

[96] B.T. Messmer and H. Bunke, Fast Error-Correcting Graph Isomorphism Based on Model Precompilation, *Technical Report IAM-96-012*, University of Bern, Switzerland, September 1996.

[97] B.T. Messmer and H. Bunke, Subgraph isomorphism in polynomial time, *Technischer Bericht IAM 95-003,* Institut für Informatik, Universität Bern, Schweiz, 1995.

[98] B. McKay, Practical graph isomorphism, *Congressus Numerantium,* 30, pp.45-87, 1981.

[99] G. Miller, Graph isomorphism, general remarks, *Journal of Computer and System Sciences*, Vol. 18, No.2, pp.128-142, April 1979.

[100] G. Miller and J.H. Reif, Parallel tree contraction part 2: Further applications, *SIAM Journal on Computing*, 20, pp.1128-1147, 1991.

[101] T. Miyazaki, The complexity of McKay's canonical labeling algorithm, *Groups and Computation, II* (L. Finkelstein and W.M. Kantor, eds.), American Mathematics Society, Providence, RI, pp.239-256, 1997.

[102] S.H. Myaeng and A. Lopez-Lopez, Conceptual graph matching: a flexible algorithm and experiments, *Journal of Experimental and Theoretical Artificial Intelligence,* 4, pp.107-126, April, 1992.

[103] T. Nagoya, R. Uehara, and S. Toda, Completeness of Graph Isomorphism Problem for Bipartite Graph Classes, *COMP2001-93, IEICE,* Technical Report, 3/12 2002.

[104] G.L. Nemhauser, Introduction to Dynamic Programming, Wiley, New York, 1966.

[105] M.A. Nielsen and I.L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2000.

[106] N.J. Nilsson, Problem-Solving methods in artificial intelligence, *McGraw-Hill*, New York, 1971.

[107] Ø. Ore, The Four-colour Problem, Academic Press, New York, 1967.

[108] V. Pan and J.H. Reif, Extension of parallel nested dissection algorithm to the path algebra problems, *Computer Science Department* TR-85-9, State University of New York at Albany, 1985.

[109] V. Pan and J. H. Reif, Fast and efficient solution of path algebra problems, *Journal of Computer and Systems Sciences,* Vol. 38, No. 3, pp.494-510, June 1989.

[110] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[111] C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization; Algorithms and Complexity, Prentice-Hall, New Jersey, 1982.

[112] C.H. Papadimitriou and M. Yannakakis, The clique problem for planar graphs, Information *Processing Letters*, 13, pp.131-133, 1981.

[113] R.C. Read, The enumeration of self complementary graphs and digraphs, *Journal of the London Mathematical Society,* 38, pp.99-104, 1963.

[114] D. Richards, Finding short cycles in planar graphs using separators, *Journal of* Algorithms, 7, pp.382-394, 1986.

[115] N. Robertson and P. D. Seymour, Graph minors II: algorithmic aspects of tree-width, *Journal of Algorithms*, 7, pp.309-322, 1986.

[116] W.L. Ruzzo, On uniform circuit complexity, *Journal of Computer and System* Sciences, 22, pp.365-383, 1981.

[117] A. Sanfeliu and K.S. Fu, A Distance Measure Between Attributed Relational Graphs for Pattern Recognition, *IEEE Transactions on Systems, Man, and Cybernetics,*Vol. 13, No. 3, pp.353-362, 1983.

[118] D.C. Schmidt and L.E. Druffel, A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices, *Journal of the Association for Computing Machinery,* Vol. 23, No. 3, pp.433-445, July 1976.

[119] G. Schmidt and T. Ströhlein, Relations and Graphs, *Discrete Mathematics for Computer Scientists*, Springer Verlag, 1993.

[120] R. Shamir, D. Tsur, Faster subtree isomorphism, *Journal of Algorithms*, Vol. 33, No. 2, pp.267-280, 1999.

[121] L.G. Shapiro and R.M. Haralick, Structural description and inexacting matching, *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI,* 3, pp.504-519, 1981.

[122] A. Steane, Quantum computing, *Reports on Progress in Physics*, 61, pp.117-173, 1998.

[123] K. Takamizawa, T. Nishizeki, and N. Saito, Linear-time computability of combinatorial problems on series-parallel graphs, *Journal of the ACM*, 29, pp.623-641, 1982.

[124] W.H. Tsai and K.S. Fu, Error-Correcting Isomorphisms of Attributed Relational Graphs for Pattern Analysis, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-9, No. 12, pp. 757-768, 1979.

[125] W.T. Tutte, A family of cubical graphs, *Proceedings of Cambridge Philosophy Society,* pp.459-474, 1947.

[126] W.-G. Tzeng, G.-S. King, A new graph invariant for graph isomorphism: probability propagation matrix, *Journal of Information Science and Engineering,* 15, pp.337-352, 1999.

[127] J.R. Ullmann, An algorithm for subgraph isomorphism, *Journal of the Association for Computing Machinery,* Vol. 23, No. 1, pp.31-42, 1976.

[128] L. Weinberg, A simple and efficient algorithm for determining isomorphism of planar triply connected graphs, *IEEE Transaction on Circuit Theory,* 13, pp.142-148, 1966.

[129] H. Whitney, A set of topological invariants for graphs, *American Journal of Mathematics,* 55, pp.321-325, 1933.

[130] K. Yamazaki, H.L. Bodlaender, B. De Fluiter, and D.M. Thilikos, Isomorphism for graphs of bounded distance width, *Algorithmica,* Vol. 24, No. 2, pp.105-127, 1999.

[131] http://www.wikipedia.org/wiki/Complexity_classes_P_and_NP

[132] http://members.aol.com/jmtsgibbs/qgraph.htm