TAMPERE UNIVERSITY OF TECHNOLOGY

JOONAS LAULUMAA

COMPUTATIONAL COMPLEXITY AND GRAPH ISOMORPHISM

Master's Thesis

Tarkastaja: Professori Keijo Ruohonen
Tarkastaja: Professori Esko Turunen
Tarkastajat ja aihe hyväksytty
tieto- ja sähkötekniikan tiedekunnan
tiedekuntaneuvoston kokouksessa
09.04.2014.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
JOONAS LAULUMAA: Computational Complexity and Graph Isomorphism
Diplomityö, 51 sivua
Helmikuu 2015
Pääaine: Diskreetti matematiikka
Tarkastajat: Professori Keijo Ruohonen, Professori Esko Turunen
Avainsanat: graafien isomorfismi, rekursioteoria, laskettavuus, Turingin kone, kompleksisuus, NP, piirikompleksisuus, NC, DET, polynominen hierarkia, interaktiiviset todistukset, Arthur-Merlin pelit, SPP

Graafien isomorfismi on laskennallinen ongelma missä tehtävänä on määrittää ovatko kaksi graafia keskenään isomorfiset eli rakenteellisesti samat. Graafien isomorfian kompleksisuus on avoin ongelma, ja se on yksi harvoista NP-ongelmista jonka ei tiedetä olevan NP-täydellinen mutta jolle ei myöskään tunneta polynomista algoritmia. Tämä on yksi tietojenkäsittelytieteen tutkituimmista avoimista ongelmista.

Laskettavuuden teorian perusteet ovat rekursiivisissa funktioissa ja rekursioteoriassa, jotka ovat vanhemmat laskennan mallit kuin Turingin koneet. Tässä diplomityössä käydään läpi aluksi rekursioteorian perusteet ja keskeiset tulokset lähtien aksioomista. Toisen luvun tavoitteena on käsitellä riittävästi rekursioteoriaa, jotta voidaan määritellä eri reduktioiden välinen implikaatiohierarkia ja tärkeimmät T- ja m-reduktiot.

Turingin koneelle voidaan määritellä erilaisia variantteja sekä aika- ja tilarajoituksia. Näistä tärkeimmät eli epädeterministinen ja oraakkeli Turingin kone käsitellään kolmannessa luvussa. Rajoittamalla käytettävissä olevia laskentaresursseja voidaan rekursiivisten funktioiden sisälle luoda eri kompleksisuusluokista koostuva hierarkia, jonka tunnetuimmat luokat ovat P ja NP. Kompleksisuusluokkia tunnetaan satoja ja tässä työssä esitellään niistä graafien isomorfismin kannalta keskeisimmät.

Neljännessä luvussa käsitellään piirikompleksisuutta sekä siihen liittyviä tuloksia. Tavoitteena on osoittaa, että graafien isomorfismi on DET-kova. Todistukseen tarvittavat kompleksisuusluokat sekä niihin liittyvää teoriaa käydään läpi.

Graafien isomorfian tiedetään kuuluvan luokkiin coAM ja SPP. Nämä luokat esitellään viidennessä luvussa, jossa käsitellään myös probabilistisia luokkia, polynominen hierarkia, interaktiiviset todistukset ja niiden erikoistapaus Arthur-Merlin hierarkia. Polynominen hierarkia romahtaa 2. tasolle jos GI on NP-täydellinen.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
JOONAS LAULUMAA : Computational Complexity and Graph Isomorphism
Master of Science Thesis, 51 pages
February 2015
Major: Discrete mathematics
Examiners: Professor Keijo Ruohonen, Professor Esko Turunen
Keywords: graph isomorphism, recursion theory, computability, Turing machine, complexity, NP, circuit complexity, NC, DET, polynomial hierarchy, interactive proofs, Arthur-Merlin games, SPP

The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic, that is, structurally the same. The complexity of graph isomorphism is an open problem and it is one of the few problems in NP which is neither known to be solvable in polynomial time nor NP-complete. It is one of the most researched open problems in theoretical computer science.

The foundations of computability theory are in recursion theory and in recursive functions which are an older model of computation than Turing machines. In this master's thesis we discuss the basics of the recursion theory and the main theorems starting from the axioms. The aim of the second chapter is to define the most important T- and m-reductions and the implication hierarchy between reductions.

Different variations of Turing machines include the nondeterministic and oracle Turing machines. They are discussed in the third chapter. A hierarchy of different complexity classes can be created by reducing the available computational resources of recursive functions. The members of this hierarchy include for instance P and NP. There are hundreds of known complexity classes and in this work the most important ones regarding graph isomorphism are introduced.

Boolean circuits are a different method for approaching computability. Some main results and complexity classes of circuit complexity are discussed in the fourth chapter. The aim is to show that graph isomorphism is hard for the class DET.

Graph isomorphism is known to belong to the classes coAM and SPP. These classes are introduced in the fifth chapter by using theory of probabilistic classes, polynomial hierarchy, interactive proof systems and Arthur-Merlin games. Polynomial hierarchy collapses to its second level if GI is NP-complete.

CONTENTS

LIST OF SYMBOLS

$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$	—	logical connectives in the evaluation order: negation NOT, conjunction AND, disjunction OR, implication, equivalence
$\oplus$	—	exclusive or XOR: $p \oplus q = (p \vee q) \wedge \neg(p \wedge q)$
$\Rightarrow, \Leftrightarrow$	—	implication, equivalence
$=, \neq$	—	equality, not equal
$\leq, <$	—	less or equal, strictly less
$\exists, \forall$	—	exists, for all
$\emptyset$	—	empty set
$\in, \notin$	—	belongs to, does not belong to
$\subseteq, \subset$	—	subset, proper subset
$\cup, \cap$	—	union, intersection
$\bar{A}$	—	complement of A
$\mathcal{P}(S)$	—	set of all subsets of S
(a, b)	—	ordered pair $\{\{a\}, \{a, b\}\}$
$\times$	—	cartesian product $X \times Y = \{(x, y) : x \in X \wedge y \in Y\}$
$\mathbf{N}$	—	set of natural numbers $\{0, 1, \ldots\}$
$\simeq$	—	equal as partial functions
$\downarrow, \uparrow$	—	converges, diverges
$\leq_T, \leq_m$	—	T-reduction, m-reduction
$\models$	—	models
ε	—	empty string
Σ^*	—	strings $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$
$\sqcup$	—	blank symbol

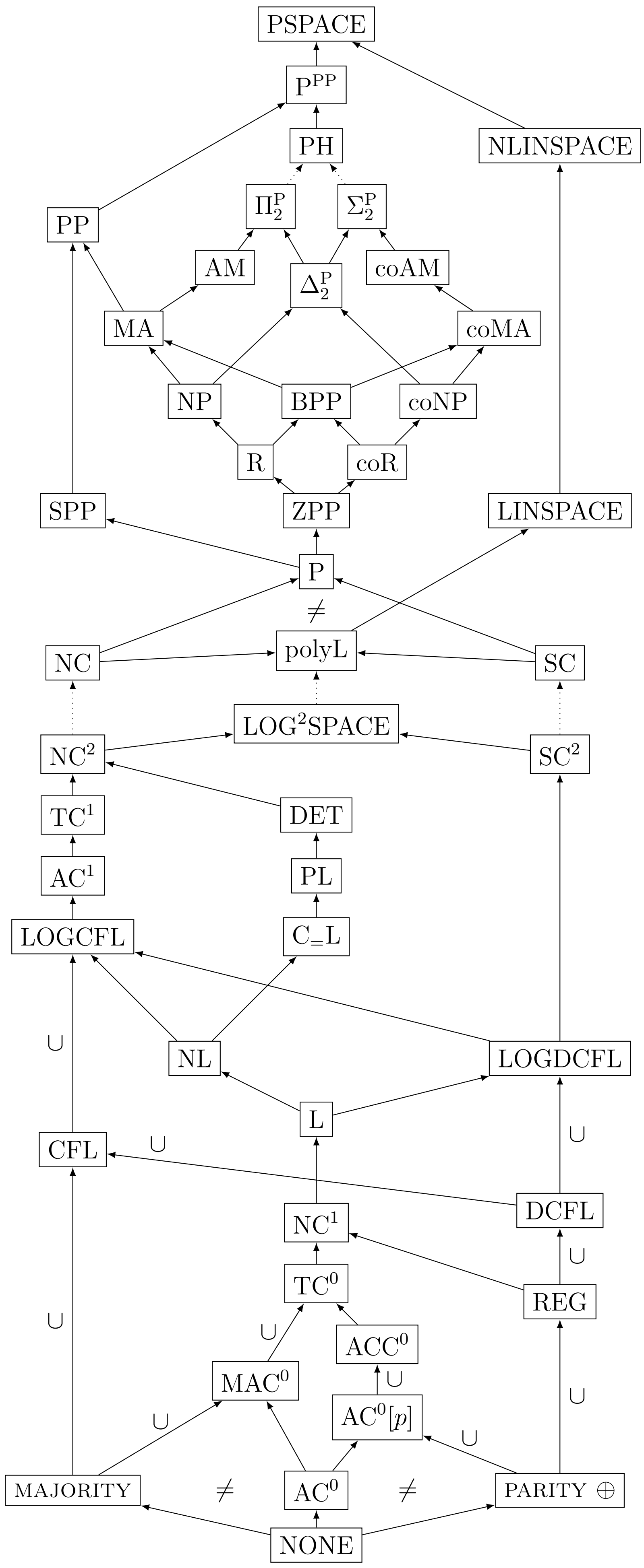

Figure 1: The known relationships and strict inclusions of complexity classes

1. INTRODUCTION

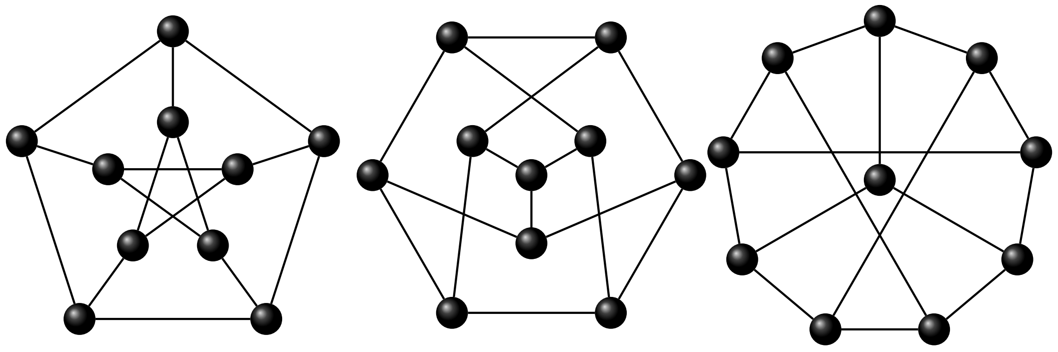

Figure 1.1: Three isomorphic Petersen's graphs

Graph isomorphism is a structure preserving bijection between two graphs. In other words, a graph G is isomorphic to a graph H if they have the same mathematical properties and differ only in naming of the nodes and edges. Three isomorphic graphs are pictured in figure 1.1. The graph isomorphism is an equivalence relation on graphs and as such it partitions the class of all graphs into equivalence classes. In this text we only consider undirected non-weighted graphs.

Graph isomorphism is one of the very small number of problems belonging to NP which is not known to be solvable in polynomial time or NP-complete. There is evidence that graph isomorphism is not NP-complete as the polynomial hierarchy would collapse to its second level if it were and it is commonly believed that the polynomial hierarchy does not collapse. Graph isomorphism is contained in coAM and SPP and the best known hardness result is that graph isomorphism is hard for DET. [32]

The best known upper bound for graph isomorphism is currently $2^{\sqrt{cn\log n}}$ for graphs with n vertices. [10] For many restricted classes of graphs like planar graphs, graphs of bounded degree and graphs with bounded eigenvalue multiplicity polynomial time algorithms are known. For trees and graphs with colored vertices and bounded color classes even NC algorithms are known. On the other hand, many different subclasses of graphs have been shown to be GI-complete including bipartite graphs, line graphs, rooted acyclic digraphs, chordal graphs, transitively orientable graphs and regular graphs. [16] [18]

Graph isomorphism has practical applications in chemistry where graphs represent molecular links and a fast graph isomorphism algorithm would allow a fast classification of different molecules with unique names. There are different implementations for graph isomorphism which take different approaches for solving the problem, for example Nauty [24], conauto [23] and vf2 [15].

1.1 Basic definitions

We start by defining some basic concepts.

Definition 1.1.

- A *problem* is a set X of ordered pairs (I, A) of strings in $\{0,1\}^*$ where I is called the *instance*. A is called an *answer* for that instance and every string in $\{0,1\}^*$ occurs as the first component of at least one pair.
- A *decision problem* is a function in which the only possible answers are "yes" and "no".
- A *counting problem* is a function in which all answers x are natural numbers $x \in \mathbf{N}$.
- A *function problem* is a problem where a single output is expected for every input, but the output is something more than that of a decision problem, that is, not just "yes" or "no".
- A *partial function* from X to Y is a function $f : X' \to Y,\ X' \subseteq X$. It generalizes the concept of a function by not forcing f to map every element of X to an element of Y. If $X' = X$, then f is called a *total function* and is equivalent to a function. The set X is the *domain* or *range* of the function.
- The *big O* or *asymptotic notation* describes the limiting behaviour of a function: $f(n) = O(g(n))$ if there are fixed positive constants c and n_0 for f, such that $f(n) \leq cg(n)\ \forall n \geq n_0$.

Next we define the concept of a *language* and some of its properties.

Definition 1.2. A *language* is any subset of $\{0,1\}^*$. If L is a language, then the decision problem R_L corresponding to L is $\{(x, yes) : x \in L\} \cup \{(x, no) : x \notin L\}$. Given a decision problem R, the language $L(R)$ corresponding to it is

$$L(R) = \{x \in \{0,1\}^* : (x, yes) \in R\}.$$

If L is a language, then its *complementary language* is $coL = \{0,1\}^* - L$.

The languages $L(R)$ and $coL(R)$ need not always belong to the same complexity class. A common question is whether a given complexity class C is "closed under complement", i.e., whether $L \in C \Rightarrow coL \in C$ for all languages L. A complexity class A is *hard* for a class of problems C if every problem in C can be reduced to A. A problem is *complete* for a class C if it is C-hard and belongs to the class C. This means that a solution to a complete problem yields a solution to all of the problems in that class.

The graph isomorphism problem is formally defined in 1.3. It is the problem of deciding whether two given graphs are isomorphic. In other words, the problem is to test whether there is a bijective function mapping the vertices of the first graph to the nodes of the second graph and preserving the adjacency relation. Three isomorphic graphs are pictured in figure 1.1. [6]

Definition 1.3. *Graph Isomorphism* (GI)
An isomorphism between two graphs G and G' is a bijective map f that preserves the edge structure.

Instance: Two undirected graphs $G = (V, E)$ and $G' = (V', E')$ where V and V' are finite sets of vertices, and E and E' are finite sets of edges (unordered pairs of vertices from V and V' respectively).

Answer: "Yes" if there is a bijective function $f : V \to V'$ such that for all pairs $\{u, v\} \subseteq V$, $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$. Otherwise, "no".

Graph isomorphism problem as a string relation: Let a_Y and a_N be strings chosen to represent "yes" and "no" respectively. The string relation would then be
$\{(x, a_Y) :$ string x consists of two representations of the same graph $G\} \cup$
$\{(x, a_N) :$ string x does not consist of two representations of the same graph $G\}$. [22] p. 70.

An *isomorphism* is a structure preserving bijection from one set to another and an *automorphism* is an isomorphism from a set to itself. A close relative to the graph isomorphism problem is the *graph automorphism* (GA). It is the problem of testing whether a graph has a nontrivial automorphism. Graph automorphism can be reduced to the graph isomorphism using polynomial time many-one reductions (GA $\leq_m^{\mathrm{P}}$ GI) but the converse reduction is unknown.

2. BASIC RECURSION THEORY

Recursion theory is the study of real numbers or, equivalently, functions over natural numbers. It tries to isolate the functions over **N** that are computable. The aim of this chapter is to introduce all the theory necessary to define different reducibilities between sets. The proofs are mostly omitted but some fundamental results of recursion theory are proven. The closely related foundational results of logic, Gödel's incompleteness theorems, are not treated here although the techniques which are used to prove these theorems, arithmetization and diagonalization, are introduced. This chapter is based mainly on the book "Classical Recursion Theory - The Theory of Functions and Sets of Natural Numbers" by Piergiorgio Odifreddi [26].

We start by introducing the successor function and an iterative procedure which generates the natural numbers. Arithmetical operations are axiomatized using the successor function. With primitive recursion and μ-recursion we define the class of recursive functions. A set is recursive if the membership in it is effectively computable. This means that both membership and nonmembership can be determined.

The main advantage of using the class of μ-recursive functions to define computation is their mathematical elegance. Proofs about this class can be presented in a rigorous and concise way, without long prose descriptions or complicated programs that are hard to verify. These functions need and make no reference to any computational machine model and still characterize computability. [8] ch. 26.3.

The recursively enumerable sets are defined using the partial recursive functions. A recursively enumerable set (r.e. set) is effectively generated; membership can be determined by waiting long enough in the generation of the set until the given element appears, but nonmembership requires waiting forever. In this chapter the properties of r.e. sets are studied in detail. Using the Cantor's diagonal argument we can construct a r.e. nonrecursive set. From the existence of such set follows the undecidability of r.e. sets. The halting problem is a reformulation of the undecidability result.

The rest of this chapter deals with different notions of reducibility. The two most important ones, T- and m-reductions, are introduced. The foundations of reducing one problem to another are in the theory of recursive functions. Other reductions are also introduced: truth-table like reductions fall between the T- and m-reductions in the implication hierarchy of different reductions.

2.1 Recursive functions

First we define the successor function $\mathcal{S}$. With the successor function we can define the natural numbers starting from the first element - the number 0 - and generate them with an iteration procedure. Thus, the first three natural numbers are

$$0,\ \mathcal{S}(0),\ \mathcal{S}(\mathcal{S}(0)),\ \ldots$$

The axioms 2.1 define the basis of the recursion theory. They are introduced for the sake of completeness and to give a starting point for our journey to the theory of computation.

Axioms 2.1. (Grassman [1861], Dedekind [1888] cited in [26] pp. 19-20)

A1 $\mathcal{S}(x) = \mathcal{S}(y) \rightarrow x = y$

A2 $0 \neq \mathcal{S}(y)$

A3 $x \neq 0 \rightarrow (\exists y)(x = \mathcal{S}(y))$

A4 $x + 0 = x$

A5 $x + \mathcal{S}(y) = \mathcal{S}(x + y)$

A6 $x \cdot 0 = 0$

A7 $x \cdot \mathcal{S}(y) = x \cdot y + x$

If φ is a formula with one free variable then

A8 $\varphi(0) \wedge (\forall x)[\varphi(x) \rightarrow \varphi(\mathcal{S}(x))] \rightarrow (\forall y)\varphi(y)$.

The axioms from A1 to A7 above give us the natural properties of addition and multiplication. The first axiom, A1, states that if the successor functions are the same, then the predecessing numbers x and y are also the same. The axiom A2 states that the number zero is the predecessor of all other natural numbers. Axiom A3 states that if the number is not equal to zero, there has to be another number predecessing it. Axioms A4 to A7 define the addition and multiplication of natural numbers.

The axiom A8 is called the axiom of induction. It can be equivalently expressed as the *Least Number Principle*

$$(\exists y)\psi(y) \rightarrow (\exists z)[\psi(z) \wedge (\forall x < z)\neg\psi(x)].$$

Based on the Least Number Principle, if we know that a number with a certain property exists, then we also know that there is the least number satisfying that property.

The primitive recursive functions are defined in 2.2. They are equivalent in computing power with *'for' programs* defined in 3.4.

Definition 2.2. (Dedekind [1888] cited in [26] p. 20) A function f is defined from g and h by *primitive recursion* if

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, y+1) &= h(\vec{x}, y, f(\vec{x}, y)). \end{aligned}$$

Definition 2.3. (Kleene [1936] cited in [26] p. 21) A function f is defined from a relation R by *μ-recursion* (minimum recursion) if

1. R is a regular predicate, i.e. $(\forall \vec{x})(\exists y)R(\vec{x}, y)$
2. $f(\vec{x}) = \mu y R(\vec{x}, y)$ is the least number y such that $R(\vec{x}, y)$ holds.

Similarly, f is defined from g by μ-recursion if

1. $(\forall \vec{x})(\exists y)(g(\vec{x}, y) = 0)$
2. $f(\vec{x}) = \mu y(g(\vec{x}, y) = 0)$.

The Least Number Principle introduced earlier can be written in μ-notation as $(\exists y)\psi(y) \to (\exists z)(z = \mu y \psi(y))$.

Using the primitive recursive functions defined in 2.2 we can build the class of primitive recursive funtions.

Definition 2.4. (Dedekind [1888], Skolem [1923], Gödel [1931] cited in [26] p. 22) The class of *primitive recursive functions* is the smallest class of functions

1. containing the initial functions

$$\begin{aligned} \mathcal{O}(x) &= 0 \\ \mathcal{S}(x) &= x+1 \\ \mathcal{I}_i^n(x_1, \ldots, x_n) &= x_i \qquad (1 \leq i \leq n) \end{aligned}$$

2. closed under composition, i.e. the schema that given $g_1, \ldots, g_m, h$ produces

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$$

3. closed under primitive recursion

A predicate is primitive recursive if its characteristic function is primitive recursive.

In the previous definition 2.4 the initial functions are the zero function $\mathcal{O}$, the successor function $\mathcal{S}$ and the identity (projection) $\mathcal{I}_i^n$. By using the composition rule and primitive recursion we can define the class of primite recursive functions.

Similarly, by using μ-recursion defined in 2.3 we can define the class of recursive functions 2.5.

Definition 2.5. (Kleene [1936] cited in [26] p. 22) The class of *recursive functions* is the smallest class of functions

1. containing the initial functions defined in 2.4
2. closed under composition, primitive recursion and μ-recursion.

A predicate is recursive if its characteristic function is recursive.

In the following theorem 2.6 the class of recursive functions is stated using sum, product, identity $\mathcal{I}_i^n$ and equality δ as the initial functions.

$$\delta(x, y) = \begin{cases} 0 & \text{if } x \neq y \\ 1 & \text{otherwise} \end{cases}$$

Theorem 2.6. (Gödel [1931], Kleene [1936] cited in [26] p. 28) The class of recursive functions is the smallest class

1. containing sum, product, identities $\mathcal{I}_i^n$ and the characteristic function δ of equality
2. closed under composition
3. closed under μ-recursion.

2.2 Partial recursive functions

Definition 2.7. A *partial function* is a function that may be undefined for some and possibly all arguments. The extended relation symbol $\simeq$ means that both sides are equal as partial functions. This means that their respective values are either both undefined, or both are defined and their value is the same. Also, $\varphi(\vec{x}) \downarrow$ means that φ is defined or *converges* for the arguments $\vec{x}$, while $\varphi(\vec{x}) \uparrow$ means that φ is not defined, it *diverges*.

The definition 2.5 (the class of recursive functions) is now adapted to partial functions.

Definition 2.8. (Kleene [1938] cited in [26] p. 127) The class of *partial recursive functions* is the smallest class of functions

1. containing the initial functions $\mathcal{O}$, $\mathcal{S}$ and $\mathcal{I}_i^n$

2. closed under composition i.e. the schema that given $\gamma_1, \ldots, \gamma_m, \psi$ produces

$$\varphi(\vec{x}) \simeq \psi(\gamma_1(\vec{x}), \ldots, \gamma_m(\vec{x})),$$

   where the left-hand side is undefined when at least one of the values of $\gamma_1, \ldots, \gamma_m, \psi$ for the given arguments is undefined

3. closed under primitive recursion, i.e. the schema that given ψ, γ produces

$$\begin{array}{rcl} \varphi(\vec{x}, 0) & \simeq & \psi(\vec{x}) \\ \varphi(\vec{x}, y+1) & \simeq & \gamma(\vec{x}, y, \varphi(\vec{x}, y)) \end{array}$$

4. closed under unrestricted μ-recursion, i. e. the schema that given ψ produces

$$\varphi(\vec{x}) \simeq \mu y[(\forall z \leq y)(\psi(\vec{x}, z)\downarrow) \wedge \psi(\vec{x}, y) \simeq 0],$$

   where $\varphi(\vec{x})$ is undefined if there is no such y.

The next theorem 2.9 reduces the partial recursive functions to a normal form. There is also a similar result for recursive functions. The reduction needs the concept of *arithmetization.* It means translation into the language of arithmetic. The reasoning of the logical-mathematical language is replaced by reasoning on natural numbers.

Theorem 2.9. *Normal Form Theorem for partial recursive functions* (Kleene [1938] cited in [26] p. 129).
There is a primitive recursive function $\mathcal{U}$ and (for each $n \geq 1$) primitive recursive predicates $\mathcal{I}_n$, such that for every partial recursive function φ of n variables there is a number e called *index* of φ for which the following hold:

1. $\varphi(x_1, \ldots, x_n)\downarrow \Leftrightarrow \exists\, y\; \mathcal{I}_n(e, x_1, \ldots, x_n, y)$

2. $\varphi(x_1, \ldots, x_n) \simeq \mathcal{U}(\mu y\; \mathcal{I}_n(e, x_1, \ldots, x_n, y))$

The Normal Form Theorem 2.9 states that every partial recursive function has an index. The index is found by associating numbers to functions and computations and putting them in a canonical form.

The corollary 2.10 states that there can be no confusion when discussing *total recursive functions*, meaning recursive functions as in definition 2.5, or partial recursive functions which are total.

Corollary 2.10. The recursive functions are exactly the partial recursive functions which happen to be total. [26] p. 129.

By introducing the partial functions every number e can be considered as the index of one particular partial recursive function. We use the notation presented in 2.11

Definition 2.11. [26] p. 130

1. φ_e^n (or $\{e\}^n$) is the e-th partial recursive function of n variables:

$$\varphi_e^n(\vec{x}) \simeq \{e\}^n(\vec{x}) \simeq \mathcal{U}(\mu y\ \mathcal{I}_n(e,\vec{x},y))$$

2. $\varphi_{e,s}^n$ (or $\{e\}_s^n$) is the finite approximation of φ_e^n of level s:

$$\varphi_{e,s}^n(\vec{x}) \simeq \{e\}_s^n(\vec{x}) \simeq \begin{cases} \varphi_{e,s}^n(\vec{x}) & \text{if} \quad (\exists\ y < s)\ \mathcal{I}_n(e,\vec{x},y)) \\ \text{undefined otherwise} \end{cases}$$

In the previous definition 2.11, regarding the computation φ_e^n, the finite approximation of calculation of level s is noted by $\varphi_{e,s}^n$. The step s is the cutting point of the calculation.

The next theorem 2.12 is the Enumeration Theorem. It states that the numbers have a double meaning in recursion theory. In addition to the natural meaning as a number they also have a meaning as a code of a function.

Theorem 2.12. *Enumeration Theorem* (Post [1922], Turing [1936], Kleene [1938] cited in [26] p. 130).
The sequence $\{\varphi_e^n\}_{e\in\mathbf{N}}$ is a partial recursive enumeration of the n-ary partial recursive functions, in the sense that:

1. for each e, φ_e^n is a partial recursive function of n variables

2. if ψ is a partial recursive function of n variables, then there is e such that

$$\psi \simeq \varphi_e^n$$

3. there is a partial recursive function φ of $n+1$ variables such that

$$\varphi(e,\vec{x}) \simeq \varphi_e^n(\vec{x}).$$

Given one index of a partial recursive function, we can effectively generate infinitely many other indices of the same function by attaching redundant equations to the description. This is also called the Padding Lemma.

The next theorem 2.13 proves the existence of computers.

Theorem 2.13. *Universal Partial Function* (Post [1922], Turing [1936], Kleene [1938] cited in [26] p. 132).
There is a partial recursive function $\varphi(e, x)$, called *universal partial function*, which generates all the partial recursive functions of any number of variables, in the sense that for every partial recursive function ψ of n variables there is e such that

$$\psi(x_1, \ldots, x_n) \simeq \varphi(e, (x_1, \ldots, x_n)).$$

Any machinery that computes the universal partial function is a computer in the contemporary sense of the word. It works as an interpreter: it decodes the program e given to it as data and simulates it.

The definition 2.14 associates partial functions with their domains.

Definition 2.14. (Post [1922], Kleene [1936] cited in [26] p. 134) An n-ary relation is *recursively enumerable* (*r.e.*) if it is the domain of an n-ary partial recursive function. $\mathcal{W}_e^n$ and $\mathcal{W}_{e,s}^n$ indicate the domains of φ_e^n and $\varphi_{e,s}^n$ respectively.

From the previous definition 2.14 we get the following characterization of recursive enumerable relations.

Theorem 2.15. *Normal Form Theorem for r.e. relations* (Kleene [1936], Rosser [1936], Mostowski [1947] cited in [26] p. 134)
An n-ary relation P is r.e. if and only if there is an $n+1$-ary recursive relation R such that

$$P(\vec{x}) \Leftrightarrow \exists y\ R(\vec{x}, y),$$

i.e. if and only if there is a number e, *index* of P, such that

$$P(\vec{x}) \Leftrightarrow \mathcal{W}_e^n(\vec{x}) \Leftrightarrow \exists y\ \mathcal{I}_n(e, \vec{x}, y).$$

The following proposition examines the close relationship between partial recursive functions and r.e. relations.

Proposition 2.16. Uniformation Property (Kleene [1936] cited in [26] p. 137)

1. If P is r.e. relation, there is a partial recursive function φ such that

$$\exists y P(\vec{x}, y) \Rightarrow \varphi(\vec{x}) \downarrow \wedge\ P(\vec{x}, \varphi(\vec{x}))$$

2. If P is r.e. and regular (see definition 2.3) relation, there is a recursive function f such that
$$\forall \vec{x} P(\vec{x}, f(\vec{x}))$$

The next two theorems 2.17 and 2.18 establish the difference between recursive enumerable and recursive sets.

Theorem 2.17. *Characterization of the r.e. sets.* (Kleene [1936] cited in [26] p. 138)
The following are equivalent:

1. A is r.e.
2. A is the range of a partial recursive function φ
3. $A = \emptyset$ or A is the range of a recursive function f.

Proposition 2.18. The following are equivalent [26] p. 139:

1. A is recursive
2. $A = \emptyset$ or A is the range of a nondecreasing, recursive function f.

The following theorem 2.19 is sometimes called *Post's Theorem.*

Theorem 2.19. (Post [1943], Kleene [1943], Mostowski [1947] cited in [26] p. 140) A set is recursive if and only if the set and its complement are recursively enumerable.

The proposition 2.20 follows from the proposition 2.18.

Proposition 2.20. (Post [1944] cited in [26] p. 141) Every infinite r.e. set has an infinite recursive subset.

From the previous results we get the defining properties of recursively enumerable and recursive sets.

Proposition 2.21. *Set-theoretical properties of r.e. sets and recursive sets* (Post [1943], Mostowski [1947] cited in [26] pp. 141-142).

- With respect to set-theoretical inclusion, the r.e. sets form a distributive lattice with smallest and greatest element, and with the recursive sets as the only complemented elements.
- The property of being r.e. is preserved under images and inverse images via partial recursive functions.
- With respect to set-theoretical inclusion, the recursive sets form a Boolean algebra.

- The property of being recursive is preserved under inverse images via recursive functions.

If A and f are recursive, then $f(A)$ is r.e. by the proposition 2.21, but it is not necessarily recursive. Any nonempty r.e. set A is the range of a recursive function f and therefore the image of $\mathbf{N}$.

2.3 Diagonalization

Georg Cantor introduced the switching function d in 1874 and proved that the set of subsets of $\mathbf{N}$ is not countable. The following theorem 2.22 introduces the often used technique called diagonalization.

Theorem 2.22. *Cantor's Theorem*
Given a set S, a function $d : S \to S,\ d(s) \neq s, \forall s \in S$ and an infinite matrix of elements on S

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & \cdots \\ s_{1,0} & s_{1,1} & s_{1,2} & \cdots \\ s_{2,0} & s_{2,1} & s_{2,2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

we get a transformed diagonal sequence of elements of S

$$d(s_{0,0})\ d(s_{1,1})\ d(s_{2,2})\ldots$$

which is not equal to any row of the matrix, because it differs from the n-th row on the n-th element by the hypothesis of d.

In the next proposition 2.23 we apply the Cantor's Theorem to recursive functions.

Proposition 2.23. *Recursive version of Cantor's Theorem* (Kleene [1936], Turing [1936] cited in [26] p. 146).
There is no recursive function which enumerates (at least one index of) each recursive $(0, 1)$-valued function.

Proof. Let f be a recursive function such that $\varphi_{f(x)}$ is total for every x, and define

$$g(x) \simeq 1 - \varphi_{f(x)}(x).$$

Then g is a $0, 1$-valued function, which is partial by Enumeration Theorem 2.12 and total by the hypothesis on f. Moreover, g is different from $\varphi_{f(x)}$ for every x, and thus no index of g is in the range of f. $\square$

We are now ready to prove one of the most important results of the Recursion Theory.

Theorem 2.24. *Combinatorial core of the undecidability results* (Post [1922], Gödel [1931], Kleene [1936] cited in [26] p. 147).
There is an r.e. nonrecursive set. Explicitly, the following set is r.e. and nonrecursive:

$$x \in \mathcal{K} \Leftrightarrow x \in \mathcal{W}_x \Leftrightarrow \varphi_x(x) \downarrow$$

Proof. By the Enumeration theorem 2.12, there is a partial recursive function φ such that

$$\varphi(x) \simeq \varphi_x(x).$$

Then $\mathcal{K}$ is r.e., because

$$x \in \mathcal{K} \Leftrightarrow \varphi(x) \downarrow .$$

To show that $\mathcal{K}$ is not recursive we give two different proofs based on the two equivalent definitions of $\mathcal{K}$.

- if $\mathcal{K}$ were recursive, the following function would be partial recursive

$$\varphi(x) = \begin{cases} 0 & \text{if } x \in \overline{\mathcal{K}} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

  Then, for some e, $\varphi \simeq \varphi_e$ and $\varphi_e(e) \downarrow \Leftrightarrow e \in \overline{\mathcal{K}}$. This contradicts the definition of $\mathcal{K}$.

- If $\mathcal{K}$ were recursive, then $\overline{\mathcal{K}}$ would be r.e. But

$$x \in \overline{\mathcal{K}} \Leftrightarrow x \notin \mathcal{W}_x,$$

  so $\overline{\mathcal{K}}$ differs on the element x from the x-th r.e. set, and cannot itself be r.e.

□

In the previous proof $\mathcal{K}$ is the diagonal r.e. set. $\overline{\mathcal{K}}$ is the set of numbers not belonging to the r.e. sets they code.

In the next theorem 2.25 it is proved that there does not exist a recursive procedure that decides whether a partial recursive function converges for given arguments. This is a reformulation of the previous theorem 2.24.

Theorem 2.25. *Unsolvability of the Halting Problem* (Turing [1936] cited in [26] p. 150).
The following set is r.e. and nonrecursive

$$\langle x, e\rangle \in \mathcal{K}_0 \Leftrightarrow x \in \mathcal{W}_e \Leftrightarrow \varphi_e(x) \downarrow .$$

Proof. $\mathcal{K}_0$ is shown to be r.e. by the Enumeration Theorem 2.12. If $\mathcal{K}_0$ were recursive so would be $\mathcal{K}$, because

$$x \in \mathcal{K} \Leftrightarrow \langle x, x \rangle \in \mathcal{K}_0$$

□

2.4 Oracle computations and reductions

The class of recursive functions 2.5 is defined as a set of initial functions and a set of operations transforming given functions into new functions. In the next definition 2.26 a function g is added to the initial functions. If g is recursive, then the class obtained is the same as in 2.5, but otherwise it is more comprehensive.

Definition 2.26. (Turing [1939] cited in [26] p. 175) If g is a total function, the class of *functions recursive in* g is the smallest class of functions

1. containing the initial functions and g
2. closed under composition, primitive recursion and restricted μ-recursion.

If S is a set, the class of *functions recursive in* S is the class of functions recursive in c_S. A predicate is recursive in g or S if its characteristic function is recursive.

The functions recursive in g are not computable unless g itself is, but they are still 'computable module g'. They are computable with the help of an *oracle*. The oracle is an extrarecursive entity which helps the computation of any function recursive in g when a call to g is made. The oracle supplies the answer to any such call for free.

Definition 2.27. Given two functions f and g, we say that [26] p. 176:

- f is *T-reducible (Turing reducible)* to g ($f \leq_T g$) if f is recursive in g
- f is *T-equivalent (Turing equivalent)* to g ($f \equiv_T g$) if $f \leq_T g$ and $g \leq_T f$.

The relation $\leq_T$ is both reflexive and transitive making $\equiv_T$ an equivalence relation. The relation $\equiv_T$ partitions the class of total functions into equivalence classes, called *Turing degrees* or *degrees of unsolvability*. The degrees can be partially ordered with $\leq_T$. Two functions are Turing equivalent (in the same Turing degree) when they are recursive in each other.

The definition 2.28 introduces the simplest special case of Turing reducibility.

Definition 2.28. (Post [1944] cited in [26] p. 257) A is *m-reducible* to B ($A \leq_m B$) if, for some recursive function f, the following equivalent conditions are satisfied:

1. $\forall x \; (x \in A \Leftrightarrow f(x) \in B)$

2. $A = f^{-1}(B)$

3. $f(A) \subseteq B \wedge f(\overline{A}) \subseteq \overline{B}$.

A is *m-equivalent* to B ($A \equiv_m B$) if $A \leq_m B$ and $B \leq_m A$.

Similarly, with $\leq_T$, $\leq_m$ is reflexive and transitive so $\equiv_m$ is an equivalence relation.

Proposition 2.29 clarifies the difference between Turing- and m-reductions with regard to recursively enumerable sets.

Proposition 2.29. (Post [1944] cited in [26] p. 252 and p. 258)

1. If S is any r.e. set then $S \leq_T \mathcal{K}$

2. A set S is r.e. if and only if $S \leq_m \mathcal{K}$.

2.5 Other reductions

In m-reducibility only one positive query to the oracle was allowed. Next step is to relax the requirements. We can allow a fixed bounded number or unboundedly many queries to the oracle. The nature of questions asked may be modified to asking whether some elements are in the oracle or not in the oracle. Also, we can restrict the way the questions are combined with logical operations.

Let $\{\sigma_n\}_{n\in\mathbf{N}}$ be an effective enumeration of all the propositional formulas built from the atomic ones '$m \in X$', for $m \in \mathbf{N}$. These are the so called *truth table conditions*, since they can be arranged in truth-tables. Given a set B, $B \models \sigma_n$ means that B satisfies σ_n, i.e. that the propositional formula σ_n becomes true when X in the atomic formulas is interpreted as B.

Definition 2.30. (Post [1944] cited in [26] p. 268 and p. 331) A is *tt-reducible (truth table reducible)* to B ($A \leq_{tt} B$) if, for some recursive function f,

$$x \in A \Leftrightarrow B \models \sigma_{f(x)}.$$

A is *btt-reducible (bounded truth table reducible)* to B ($A \leq_{btt} B$) if $\sigma_{f(x)}$ uses at most m elements, where the number m is called the *norm* of the reduction. If m is the norm, we write $A \leq_{btt(m)} B$.

If we limit the nature of what kinds of questions we can ask the oracle we get the

$$\begin{tikzcd}
 & & & Q \arrow[ddrr] & & \\
 & c \arrow[urr] \arrow[dr] & & & & \\
m \arrow[ur] \arrow[dr] \arrow[ddr] & & p \arrow[r] & tt \arrow[r] & wtt \arrow[r] & T \\
 & d \arrow[ur] & & & & \\
 & btt(1) \arrow[r] & l \arrow[uur] & & &
\end{tikzcd}$$

Figure 2.1: Implication hierarchy between reducibilities [19] p. 101, [26] p. 341.

following reductions:

$\leq_{tt}$	$= \{\neg, \wedge, \vee\}$	truth-table reducibility
$\leq_p$	$= \{\wedge, \vee\}$	positive reducibility
$\leq_c$	$= \{\wedge\}$	conjunctive reducibility
$\leq_d$	$= \{\vee\}$	disjunctive reducibility
$\leq_{btt(1)}$	$= \{\neg\}$	bounded truth-table reducibility with norm 1
$\leq_l$	$= \{\oplus\}$	linear reducibility

The previous six reductions together with $\leq_m$ are the only possible truth-table-like reducibilities. On the non-trivial r.e. sets $\leq_m$ and $\leq_{btt(1)}$ coincide [19] p. 100. Structures induced by conjunctive and disjunctive reductions are isomorphic because [26] p. 591

$$A \leq_c B \Leftrightarrow \overline{A} \leq_d \overline{B}$$

In *wtt-reduction (weak truth-table reduction)* the reductions may diverge. In *Q-reduction* the T-reduction is strengthened so that we ask the oracle only questions about singletons. Figure 2.1 shows the hierarchy between reductions. There are also many other reductions not mentioned here.

3. TURING MACHINES AND THE MODELS OF COMPUTATION

In this chapter some familiarity with Turing machines is presumed. Most of the definitions are from the book "Introduction to the Theory of Computation" by Michael Sipser. [30]

A Turing machine can be tought of as a typewriter which has an infinite tape serving as memory. Furthermore, there are instructions according to which the machine operates. Turing machines are formally defined in 3.1. [30] p. 140.

Definition 3.1. *Turing Machine (TM)*
A *Turing machine* is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets.

1. Q is the set of states,
2. Σ is the *input alphabet* not containing the *blank symbol* $\sqcup$
3. Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.

The following theorem 3.2 links the recursive functions and Turing machines together.

Theorem 3.2. (Turing [1936] cited in [26] p. 54) Every recursive function is Turing machine computable.

To prove the theorem 3.2 we need to construct Turing machines that compute the conditions that define class of recursive functions 2.5. The different properties of recursive functions can be defined as separate Turing machines. The Turing machine that computes $\mathcal{O}$ just takes the input and moves one step to the right in the tape and prints a tally. The machine computing $\mathcal{S}$ makes a copy of the input to the right and then adds one more tally. Similarly, more complicated machines can be defined to compute $\mathcal{I}_n$, composition, primitive recursion and μ-recursion.

Theorem 3.3 combined with the previous theorem 3.2 proves that Turing machines compute exactly the recursive functions.

Theorem 3.3. (Turing [1936], [1937]) cited in [26] p. 99) Every Turing machine computable function is recursive.

Proof. By arithmetization we can define $\mathcal{I}_n(e, x_1, \ldots, x_n, y)$ primitive recursive.

- y codes a computation carried out by the TM coded by e on inputs $x_1, \ldots, x_n$.

Let $\mathcal{U}$ be a primitive recursive function such that

- if y codes a computation then $\mathcal{U}(y)$ is the value of the number written on the tape to the left of the head, in the last configuration of the computation coded by y.

If f is computed by the Turing machine coded by e then f is recursive, because

$$f(x_1, \ldots, x_n) = \mathcal{U}(\mu y\ \mathcal{I}_n(e, x_1, \ldots, x_n, y)).$$

□

Programming languages use the concepts of *for* and *while* which are introduced formally in the next two definitions.

Definition 3.4. Consider the programming language whose statements are the following:

1. assignment statements ($X := 0,\ X := X + 1$ and $X := X - 1$)
2. 'for' statements ('*for Y do S*', with S arbitrary statement (meaning: iterate S for Y times)
3. compound statements (*begin* $S_1, \ldots, S_n$ *end*, with S_i arbitrary statements).

A *'for' program* is any compound statement. A function is 'for' computable if and only if it is primitive recursive. [26] p. 70.

"While" programs are strictly more powerful than "for" programs as they compute the recursive functions.

Definition 3.5. Consider the programming language whose statements are the following:

1. assignment statements ($X := 0,\ X := X + 1$ and $X := X - 1$)
2. 'while' statements (*while* $X \neq Y$ *do* S, with S arbitrary statement)
3. compound statements (*begin* $S_1, \ldots, S_n$ *end*, with S_i arbitrary statements).

A *'while' program* is any compound statement and 'while' programs generate all the recursive functions. [26] p. 68.

There are numerous ways to define the recursive functions. Some of these are named in the next theorem 3.6.

Theorem 3.6. *Basic result.* The following are equivalent:

1. f is recursive
2. f is Turing computable
3. f is flowchart (or 'while') computable
4. f is finitely definable
5. f is Herbrand-Gödel computable
6. f is representable in a consistent formal system extending $\mathcal{R}$
7. f is λ-definable.

The class of recursive functions arises in different fields such as mathematics, logic, computer science and linguistics which turn out to be equivalent. This has led to the following thesis proposed by Alonzo Church and Alan Turing in 1936.

Thesis. *Church-Turing Thesis.* Every effectively computable function is recursive.

3.1 Variants

In this section some variations to Turing machines are introduced. They compute the same functions as regular Turing machines but they are useful in classifying different complexity classes.

Nondeterministic Turing machine is a Turing machine whose instructions are not required to be consistent, so that more than one could be applicable in a given situation. It is formally defined in 3.7.

Definition 3.7. *Nondeterministic Turing machine* differs from the deterministic version (definition 3.1) in transition function which is the power set of the transition function from the original definition: [30] p. 150.

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

The nondeterministic Turing machine has a set of possible computations on a given input. Every nondeterministic Turing machine has an equivalent deterministic Turing machine. If the use of resources is not considered, then nondeterministic machines have the same power as deterministic ones. [11] p. 28.

Definition 3.8. *Oracle Turing machine*
An *oracle* for a language A is an external device that is capable of telling whether any string w is a member of A. An oracle Turing machine is a Turing machine that has the additional capability of querying an oracle. Querying the oracle is like invoking a subroutine for solving A without counting the time required by the subroutine. [30] p. 232, [22] p. 74.

Complexity class of decision problems solvable by on algorithm in class A with an oracle for language L is called A^L. This can be extended to a set of languages or a complexity class B:

$$\mathrm{A}^{\mathrm{B}} = \bigcup_{L \in \mathrm{B}} \mathrm{A}^L$$

A language or complexicity class A is *low* for a class C if $\mathrm{C}^A = \mathrm{C}$ meaning that A is powerless as an oracle for C.

The following classes of languages are defined mostly for sake of completeness. For a more detailed study see for example [30].

Definition 3.9.

- REG is the class of regular languages. There exists many equivalent definitions for REG for example using finite automata or regular expressions.

$$\textsc{parity} \in \mathrm{REG} \Rightarrow \mathrm{REG} \neq \mathrm{AC}^0$$

- CFL is the class of context-free languages. There exists different ways to define them for example using context-free grammars or nondeterminic pushdown automata. DCFL is the class of deterministic context-free languages. They are defined using a deterministic pushdown automata. The following proper inclusions between the languages are known:

$$\mathrm{REG} \subset \mathrm{DCFL} \subset \mathrm{CFL}$$

3.2 Time and space hierarchies

Limiting the amount of time and space resources available to deterministic and nondeterministic Turing machines new complexity classes can be defined. In this section some of these classes and their properties are introduced. The main definitions are from Algorithms and Theory of Computation Handbook [8] ch. 27.2.

First we define the basic notions of deterministic and nondeterminic time and space.

Definition 3.10. Given functions $t(n)$ and $s(n)$

- DTIME$[t(n)]$ is the class of functions decided by a deterministic Turing machine of time complexity $t(n)$.
- NTIME$[t(n)]$ is the class of functions decided by a nondeterministic Turing machine of time complexity $t(n)$.
- DSPACE$[s(n)]$ is the class of functions decided by a deterministic Turing machine of space complexity $s(n)$.
- NSPACE$[s(n)]$ is the class of functions decided by a nondeterministic Turing machine of space complexity $s(n)$.

Theorem 3.11 states that there is a relationship between deterministic and nondeterministic space.

Theorem 3.11. Savitch's Theorem [28]
Let $s(n) \geq \log_2 n$ be a space-constructible function. Then,

$$\text{NSPACE}[s(n)] \subseteq \text{DSPACE}[s(n)^2].$$

The complement of a nondeterministic space complexity class is the same as the original class. This is formalized in theorem 3.12

Theorem 3.12. Immerman - Szelepcsényi Theorem [21] [31]
For any function $s(n) \geq \log n$

$$\text{NSPACE}[s(n)] = \text{coNSPACE}[s(n)]$$

Using different values for the function $t(s)$ we can define time complexity classes.

Definition 3.13. Time complexity classes

- DLOGTIME is the complexity class of all computational problems solvable in a logarithmic amount of computation time on a deterministic Turing machine. It must be defined on a special Turing machine with direct access to its input, since otherwise the machine does not have time to read the entire input tape.
- $\text{P} = \bigcup_{k \geq 1} \text{DTIME}[n^k] = \text{DTIME}[n^{O(1)}]$ (polynomial time)

- NP $= \bigcup_{k\geq 1}$ NTIME$[n^k] =$ NTIME$[n^{O(1)}]$ (nondeterministic polynomial time)

It is obvious that GI $\in$ NP, we just "guess" the isomorphism with an NP-machine if it exists. The question whether or not P $=$ NP is a famous one. It is commonly believed that P $\subset$ NP.

More time complexity classes above NP can be constructed by giving DTIME and NTIME exponential arguments. It has been proved that those classes can never be solved assuming Church-Turing thesis so they have been left out.

Space complexity classes are defined similarly.

Definition 3.14. Space complexity classes

- L $= \bigcup_{c\geq 1}$ DSPACE$[c \log n] =$ DSPACE$[O(\log n)]$ (logarithmic space)
- NL $= \bigcup_{c\geq 1}$ NSPACE$[c \log n] =$ NSPACE$[O(\log n)]$ (nondeterministic logarithmic space)
  Using theorem 3.12 we can see that NL $=$ coNL.
- polyL $= \bigcup_{k>1}$ DSPACE$[\log^k n] =$ DSPACE$[\log^{O(1)} n]$ (polylogarithmic space)
  polyL $\neq$ P because polyL does not have complete problems under many-one log-space reductions.
- LINSPACE $= \bigcup_{c>0}$ DSPACE$[cn] =$ DSPACE$[O(n)]$ (linear space)
- NLINSPACE $= \bigcup_{c>0}$ NSPACE$[cn] =$ NSPACE$[O(n)]$ (nondeterminstic linear space)
- PSPACE $= \bigcup_{k\geq 1}$ DSPACE$[n^k] =$ DSPACE$[n^{O(1)}]$ (polynomial space)

Savitch's Theorem 3.11 states that nondeterministic polynomial space equals PSPACE. Also, the same theorem can be applied to polyL to get the same result. L $\subseteq$ NL but it is not known whether they are different or not. L, NL and polyL are all proper subclasses of PSPACE.

LINSPACE and NLINSPACE are both subclasses of PSPACE. Unlike PSPACE however, they are not known to contain NP or even P. We do know that they are not equal to P or NP since LINSPACE and NLINSPACE are not closed under polynomial transformations whereas P and NP both are. [27]

Combining the m- and T-reductions defined in 2.28 and 2.27 with the time and space limits we can define more restrictive reductions. Two of them have been given a special name:

- A problem A is *Karp reducible* to a problem B ($A \leq_m^{\mathrm{P}} B$) if the m-reduction uses polynomial time resources.
- Similarly, a problem A is *Cook reducible* to a problem B ($A \leq_T^{\mathrm{P}} B$) if the T-reduction uses polynomial time resources.

The following complexity classes are defined mostly for the sake of completeness.

Definition 3.15. Additional complexity classes

- The class SC named after Stephen Cook is the class of all decision problems solvable by DTMs that simultaneously obey polynomial time bounds and poly-logarithmic space bounds:

$$\begin{aligned}\mathrm{SC} &= \mathrm{DTIME}[n^{O(1)}] \wedge \mathrm{DSPACE}[\log^{O(1)} n]\\ \mathrm{SC}^k &= \mathrm{DTIME}[n^{O(1)}] \wedge \mathrm{DSPACE}[\log^{k} n]\end{aligned}$$

- The class LOGCFL consists of all those decision problems that are log-space reducible to a context free language CFL defined in 3.9. It is closed under complementation. [12]

$$\begin{aligned}\mathrm{CFL} \subset \mathrm{LOGCFL} &\subseteq \mathrm{AC}^1\\ \mathrm{NL} &\subseteq \mathrm{LOGCFL}\end{aligned}$$

- Similarly, LOGDCFL consists of all those decision problems that are log-space reducible to a deterministic context free language DCFL.

$$\begin{aligned}\mathrm{DCFL} \subset \mathrm{LOGDCFL} &\subseteq \mathrm{LOGCFL}\\ \mathrm{L} \subseteq \mathrm{LOGDCFL} &\subseteq \mathrm{SC}^2 \text{ [13]}\end{aligned}$$

It is an open problem whether graph isomorphism is contained in the class LOGCFL.

4. BOOLEAN CIRCUITS AND CIRCUIT COMPLEXITY

The best known hardness result for graph isomorphism is related to Boolean circuits and circuit complexity. In this chapter we define the basic theory of Boolean functions and circuits and related complexity classes. The definitions used in this chapter are mainly from the book "Introduction to Circuit Complexity" by Heribert Vollmer. [33]

The basic properties of Boolean functions are defined in 4.1. A Boolean function is a function, the input of which is a binary vector and output is a scalar 0 or 1. For example the logical operations $\vee$ and $\wedge$ are Boolean functions.

The family of Boolean functions is a sequence of functions in which the first function takes 0 elements as its input, the second one takes 1 element and so on. Therefore we get an infinite number of functions where every function takes a different number of elements as its argument.

Definition 4.1.

- A *Boolean function* is a function $f : \{0,1\}^m \to \{0,1\}$ for some $m \in \mathbf{N}$.
- A *family of Boolean functions* is a sequence $f = (f^n)_{n \in \mathbf{N}}$, where f^n is an n-ary Boolean function.
- A *basis* is a finite set consisting of Boolean functions and families of Boolean functions.

The previous definition 4.1 is now used to define the concept of a circuit.

Definition 4.2. Let B be a basis. A *Boolean circuit* over B with n inputs and m outputs is a 5-tuple

$$\mathcal{C} = (V, E, \alpha, \beta, \omega),$$

where

- (V, E) is a finite directed acyclic graph,
- $\alpha : E \to \mathbf{N}$ is an injective function,

- $\beta : V \to B \cup \{x_1, \ldots, x_n\}$, and
- $\omega : V \to \{y_1, \ldots, y_m\} \cup \{*\}$

such that the following conditions hold:

1. If $v \in V$ has in-degree 0, then $\beta(v) \in \{x_1, \ldots, x_n\}$ or $\beta(v)$ is a 0-ary Boolean function (i.e. Boolean constant) from B.
2. If $v \in V$ has in-degree $k > 0$, then $\beta(v)$ is a k-ary Boolean function from B or a family of Boolean functions from B.
3. $\forall i \in \{1, \ldots, n\}$, there is at most one node $v \in V$ such that $\beta(v) = x_i$.
4. $\forall i \in \{1, \ldots, m\}$, there is exactly one node $v \in V$ such that $\omega(v) = y_i$.

A *gate* $v \in \mathcal{C}$ is a node $v \in V$ which has an in-degree (*fan-in*) k_0 and an out-degree (*fan-out*) k_1. A *wire* is an edge $e = (u, v) \in E$. The function β tells us whether the gate is an input node or a computation node. The function ω designates certain nodes $\omega(v) \neq *$ as output nodes.

The definition 4.2 can be intuitively understood as a collection of *gates* and *inputs* that are connected by *wires* [30] p. 352. A circuit $\mathcal{C} = (V, E, \alpha, \beta, \omega)$ with n inputs and m outputs computes a function

$$f_{\mathcal{C}} : \{0,1\}^n \to \{0,1\}^m.$$

An example circuit computing the parity function is pictured in figure 4.1.

The definition 4.3 expands the definition of a circuit by allowing additional oracle gates which do not have to calculate a Boolean function. Additional complexity classes can be defined using the oracle circuits.

Definition 4.3. An *oracle-augmented Boolean circuit* is a Boolean circuit with an additional class of "oracle" gates allowed, where the latter can have any number of inputs and outputs. The input string for such a gate is the sequence of the values on its input gates. The output string is the sequence of values on its output gates. Given a search problem X as oracle, the output string of an oracle gate with input string x is any y that is an answer for x in X. If no answer exists, the circuit containing the gate fails. [22] p. 136.

In order to consider infinite functions we have to build an infinite family of circuits. One circuit is given for each input length. This is formalized in the definition 4.4.

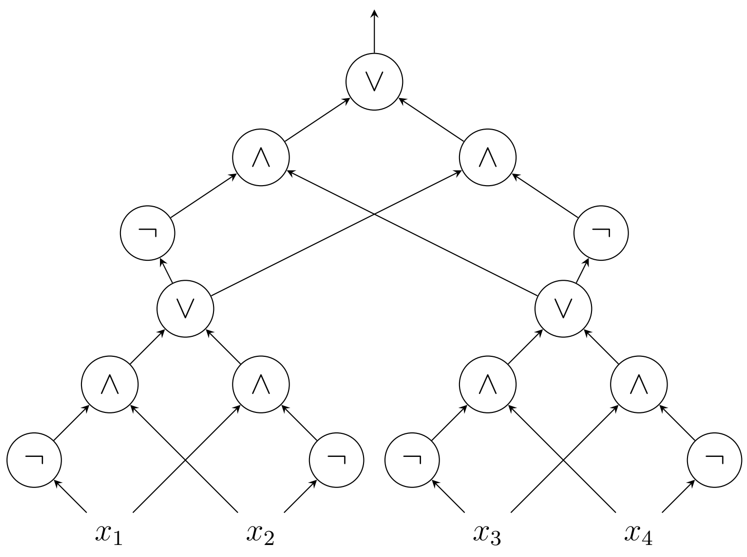

Figure 4.1: A Boolean circuit computing the parity function $\oplus$ on four variables $x_1, \ldots, x_4$.

Definition 4.4. Let B be a basis. A *circuit family* over B is a sequence $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \ldots)$, where $\forall n \in \mathbf{N}$, $\mathcal{C}_n$ is a circuit over B with n inputs. Let f^n be the function computed by $\mathcal{C}_n$. Then we say that $\mathcal{C}$ computes function $f : \{0,1\}^* \to \{0,1\}^*$, defined by

$$\forall s \in \{0,1\}^* : \ f(s) = f^{|s|}(s).$$

To be able to define complexity classes using circuit families we have to define the *size* and the *depth* of circuits. In addition, we need to be able to construct the circuits using Turing machines. These are defined in 4.5.

Definition 4.5. Let $\mathcal{C} = (V, E, \alpha, \beta, \omega)$ be a circuit over B. The *size* of $\mathcal{C}$ is defined to be the number of non-input gates in V, and the *depth* of $\mathcal{C}$ is defined to be the length of the longest directed path in the graph (V, E).

- Let $\mathcal{C} = (\mathcal{C}_n)_{n \in \mathbf{N}}$ be a circuit family, and let $s, d : \mathbf{N} \to \mathbf{N}$. $\mathcal{C}$ has size s and depth d if $\forall n$, $\mathcal{C}_n$ has size $s(n)$ and depth $d(n)$.
- A family $\mathcal{C} = (\mathcal{C}_n)_{n \in \mathbf{N}}$ of Boolean circuits is *log-space uniform* if there is a DTM M that, given n, constructs $\mathcal{C}_n$ using space $O(\log n)$.

Next, in 4.6, we define the class NC named after Nicholas Pippinger. It is the same class as AC. The only difference between the two classes is that in AC we accept unbounded fan-in whereas in NC the fan-in for every gate is 2.

Definition 4.6.

- For each $k \geq 1$ the class NC^k consists of all languages recognizable by log-space uniform family of Boolean circuits having polynomial size $O(n^{O(1)})$ and depth $O(\log^k n)$.

- In the class AC^k the Boolean circuits have *unbounded fan-in.*

$$\forall\ k \geq 0, \quad \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1} \subseteq \mathrm{AC}^{k+1}$$

$$\mathrm{AC} = \bigcup_{k=0}^{\infty} \mathrm{AC}^k = \bigcup_{k=0}^{\infty} \mathrm{NC}^k = \mathrm{NC}$$

The following classes in 4.7 are defined mostly for the sake of completeness. Their relationships are pictured in figure 1.

Definition 4.7.

1. AC^0 is the class of problems solvable by polynomial-size, constant-depth circuits of AND, OR and NOT gates of unbounded fan-in. AC^0 corresponds to $O(1)$-time computation on a parallel computer, and it also consists exactly of the languages that can be specified in first-order logic. AC^0-circuits are powerful enough to add and subtract n-bit numbers. The XOR function is not in AC^0. [17]

2. NC^1 is the class of problems solvable by circuits of AND, OR and NOT gates of fan-in two, size $O(n^{O(1)})$ and depth $O(\log n)$.

3. TC^0 is the class of problems solvable by polynomial-size, constant-depth threshold circuits. TC^0 captures exactly the complexity of integer multiplication, division and sorting. Also, TC^0 is a good complexity-theoretical model for *neural net* computation.

4. $\mathrm{AC}^0(m)$ is the class of problems solvable by polynomial-size, constant-depth circuits of AND, OR, NOT and MOD_m gates of unbounded fan-in. A MOD_m gate takes inputs $x_1, \ldots, x_n$ and determines if the number of 1's among these inputs is a multiple of m. If p is a prime number, then

$$\mathrm{AC}^0 \subset \mathrm{AC}^0[p] \subset \mathrm{TC}^0 \subseteq \mathrm{NC}^1.$$

   PARITY $\oplus$ is contained in $\mathrm{AC}^0[p]$. [33]

5. The class ACC^0 (AC with circuits) is defined as: $\mathrm{ACC}^0 = \bigcup_m \mathrm{AC}^0(m)$, where ACC^0 corresponds to computation in any *solvable monoid.* Computing the *permanent* is not possible for logspace-uniform ACC^0 circuits, thus $\mathrm{ACC}^0 \subset$ PP.

6. The MAJORITY-function is defined as follows:

MAJORITY: $\{0,1\}^n \to \{0,1\}$

$$\text{MAJORITY}(x_1,\dots,x_n) = \begin{cases} 0 & \sum x_i < \frac{n}{2} \\ 1 & \sum x_i \geq \frac{n}{2} \end{cases}$$

MAC^0 is the class of polynomial-size, constant-depth circuits of AND, OR, NOT and a single MAJORITY gate at the root. $\text{MAC}^0 \subset \text{TC}^0$ [7]. It is conjegured that ACC^0 can not compute the MAJORITY. [1]

4.1 Graph Isomorphism is hard for DET

The main result of this chapter is that graph isomorphism is *hard* for the class DET of integer determinant. In this section we introduce the basic elements needed for the proof. A reader interested in the details of the proof is referred to the original publication by Jacobo Toran. [32]

To define the necessary complexity classes we have the following definition 4.8.

Definition 4.8. The *st-connectivity* is a decision problem asking, for vertices s and t in a directed graph, if t is reachable from s. It is known to be complete for NL, the class of languages accepted by nondeterministic Turing machines using logarithmic space. [28]

By using the previous definition 4.8 we can now define three different complexity classes. The definition uses the fact that a computation of a Turing machine can be understood as a directed graph of different computation paths.

Definition 4.9. #L defined in [3] is the class of functions $f : \Sigma^* \to \mathbf{N}$ that count the number of accepting paths of a log-space nondeterministic Turing machine M on input x. See definition 5.15 for the details of counting Turing machines. The computation of #L function on input x can be reduced to the st-connectivity problem defined in 4.8. Using the #L functions we can define the classes PL (probabilistic logarithmic space), $\text{C}_=\text{L}$ (exact threshold in logarithmic space) and Mod_kL (modular counting in logarithmic space, $k \geq 2$):

$$\begin{aligned} \text{PL} &= \{A : \exists p \in \text{Poly}, f \in \#\text{L},\ x \in A \Leftrightarrow f(x) \geq 2^{p(|x|)}\} \\ \text{C}_=\text{L} &= \{A : \exists p \in \text{Poly}, f \in \#\text{L},\ x \in A \Leftrightarrow f(x) = 2^{p(|x|)}\} \\ \text{Mod}_k\text{L} &= \{A : \exists f \in \#\text{L},\ x \in A \Leftrightarrow f(x) = 1 \bmod k\} \end{aligned}$$

Mod_k circuits $k \geq 2$ are circuits where the input variables can take values in $\mathbf{Z}_k$, and the gates compute addition in $\mathbf{Z}_k$. [32]

The class PL can be alternatively defined using probabilistic Turing machines defined in 5.4. PL is the class of languages A for which there exists a probabilistic Turing machine such that on input x the machine never uses more than $\log |x|$ space, and $x \in A$ if and only if the probability that the machine reaches an accepting configuration is $> \frac{1}{2}$. [2]

Definition 4.10. DET is a class of problems NC^1 Turing reducible to the *integer determinant*, the problem of computing the determinant of an n by n matrix of n-bit integers. In other words, the class of problems that can be solved by NC^1 circuits with additional oracle gates defined in 4.3 that can compute the determinant of integer matrices. It was first defined by Cook. [14]

The known relationships of the classes from defitions 4.9 and 4.10 are the following:

$$\mathrm{Mod}_k\mathrm{L} \subseteq \mathrm{DET},$$
$$\mathrm{NL} \subseteq \mathrm{C}_{=}\mathrm{L} \subseteq \mathrm{PL} \subseteq \mathrm{DET} \subseteq \mathrm{NC}^2$$

The AC^0 many-one reductions are used in the proofs.

Definition 4.11. A set A is DLOGTIME uniform AC^0 many-one reducible ($\leq_m^{\mathrm{AC}^0}$) to another set B if there is a family of AC^0 circuits $\{\mathcal{C}_n |\ n \in \mathbf{N}\}$ and

$$\forall x,\ |x| = n,\ x \in A \ \Leftrightarrow\ \mathcal{C}_n(x) \in B.$$

The proof of the hardness results starts by showing that the graph isomorphism problem has enough structure to encode a modular addition gate. For any ($k \in \mathbf{N}$) circuit value problem for addition mod k gates is Mod$_k$L-complete. This problem is then reduced to graph isomorphism via AC^0 many-one reductions.

The idea is to simulate a modular gate with a graph gadget and then combine the gadgets for the different gates into a graph. Any certain automorphism maps a special vertex encoding output gate to a vertex encoding the output of the circuit. This simulates the behaviour of the modular circuit. See [32] for details of the graph gadget.

The Chinese representation theorem 4.12 is used in the reduction of graph isomorphism to the class NL.

Theorem 4.12. A Chinese remainder representation base is a set $m_1, \ldots, m_n$ of pairwise coprime integers. Let $M = \prod_{i=1}^{m} m_n$. By the CRT, every integer $0 \leq x < M$

is uniquely represented by its Chinese remainder representation $(x_1, \ldots, x_n)$, where $0 \leq x_i < m_i$ and $x_i = x \mod m_i$.

NL is closed under complementation and the graph accessibilty problem 4.8 is complete for NL. The complement of the theorem 4.8 is reduced to graph isomorphism via $\leq_m^{\mathrm{AC}^0}$-reductions using the Chinese remainder theorem 4.12.

The next step is to show that any logarithmic space counting function $f \in \#\mathrm{L}$ can be reduced to graph isomorphism. This implies that GI is hard for $\mathrm{C_{=}L}$ and PL. The proof uses the result that division can be computed by uniform TC^0 circuits [20]. Also, the fact that NC^1 circuit with fixed values in the input nodes can be encoded as a graph isomorphism question is needed.

DET can be alternative defined as $\mathrm{NC}^1(\#\mathrm{L})$ the class of problems computed by an AC^0-uniform family of polynomial size and logarithmic depth circuits with oracle gates to a function $f \in \#\mathrm{L}$. Using this alternative definition we get the best known hardness result for graph isomorphism. [2] [32]

Theorem 4.13. Graph isomorphism is hard for the class DET under $\leq_m^{\mathrm{AC}^0}$-reductions. [32]

5. POLYNOMIAL HIERARCHY AND PROBABILISTIC CLASSES

In the previous chapter we introduced the best known hardness results for graph isomorphism. In this chapter we introduce the complexity classes which are known to contain GI and also argue why it is not believed that the GI is NP-complete.

We start by introducing the polynomial hierarchy which is an infinite hierarchy of classes defined with oracles. The infinite hierarchy would collapse to its second level if GI were NP-complete.

To prove other known containment results for GI we first need to define machine models for probabilistic computation. Using these models and Arthur-Merlin hierarchies - a restricted version of interactive proof systems - it can be shown that GI $\in$ coAM.

Finally, we define the complexity class SPP and introduce the concepts needed for proving that GI $\in$ SPP.

5.1 Polynomial hierarchy

The *polynomial hierarchy* PH generilizes P, NP and coNP using oracles. It is formally defined as an infinite hierarchy in 5.1. [25]

Definition 5.1. Polynomial hierarchy (PH) is defined as follows. It is pictured in figure 5.1.

$$\Delta_0^{\mathrm{P}} = \Sigma_0^{\mathrm{P}} = \Pi_0^{\mathrm{P}} = \mathrm{P}$$
$$\forall k \geq 0: \quad \Delta_{k+1}^{\mathrm{P}} = \mathrm{P}^{\Sigma_k^{\mathrm{P}}}, \quad \Sigma_{k+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_k^{\mathrm{P}}}, \quad \Pi_{k+1}^{\mathrm{P}} = \mathrm{co}\Sigma_{k+1}^{\mathrm{P}}$$

In particular

$$\Sigma_1^{\mathrm{P}} = \mathrm{NP}, \quad \Pi_1^{\mathrm{P}} = \mathrm{coNP} \quad \text{and} \quad \Delta_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P}} \cap \Pi_k^{\mathrm{P}}, \quad \Sigma_k^{\mathrm{P}} \cup \Pi_k^{\mathrm{P}} \subseteq \Delta_{k+1}^{\mathrm{P}}.$$

The polynomial hierarchy is equal to the union:

$$\mathrm{PH} = \bigcup_{k=0}^{\infty} \Sigma_k^{\mathrm{P}}.$$

$$\begin{tikzcd}
 & \mathrm{PH} & \\
\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}} = \Sigma_3^{\mathrm{P}} \arrow[ur, dotted] & & \Pi_3^{\mathrm{P}} = \mathrm{coNP}^{\mathrm{NP}^{\mathrm{NP}}} \arrow[ul, dotted] \\
 & \Delta_3^{\mathrm{P}} = \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}} \arrow[ul] \arrow[ur] & \\
\mathrm{NP}^{\mathrm{NP}} = \Sigma_2^{\mathrm{P}} \arrow[ur] & & \Pi_2^{\mathrm{P}} = \mathrm{coNP}^{\mathrm{NP}} \arrow[ul] \\
 & \Delta_2^{\mathrm{P}} = \mathrm{P}^{\mathrm{NP}} \arrow[ul] \arrow[ur] & \\
\mathrm{NP} = \Sigma_1^{\mathrm{P}} \arrow[ur] & & \Pi_1^{\mathrm{P}} = \mathrm{coNP} \arrow[ul] \\
 & \Delta_0^{\mathrm{P}} = \mathrm{P} = \Delta_1^{\mathrm{P}} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 5.1: Polynomial hierarchy

The polynomial hierarchy gives a more detailed way of classifying NP-hard decision problems which are not as "easy" as NP-complete problems. Every element in PH can be solved by exhaustive search in DTIME$[O(2^{p(n)})]$, where p is a polynomial. The following inclusions are known:

$$\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{PP}} \subseteq \mathrm{PSPACE}.$$

For any $k > 0$, $\Sigma_k^{\mathrm{P}} = \Sigma_{k+1}^{\mathrm{P}}$ implies $\mathrm{PH} = \Sigma_k^{\mathrm{P}}$. Graph isomorphism is contained in the polynomial hierarchy (section 5.4.1) and if GI were NP-complete then the polynomial hierarchy would collapse to Σ_2^{P}. It is commonly believed that PH does not collapse so there is strong evidence that GI is not NP-complete. GI is also *low* for Σ_2^{P} meaning that it is powerless as an oracle for Σ_2^{P}. [8]

5.2 Probabilistic classes

In this section two different models for random and probabilistic computation are introduced. Without the loss of generality it is assumed that the underlying nondeterministic Turing machine is normalized so that all computations are finite, have the same length and that every internal node in the computation tree has either one or two successors. We first define the *random Turing machine* in 5.2.

Definition 5.2. A *random Turing machine* (RTM) is a nondeterministic Turing machine such that, for each possible input string, either there are no accepting computations or else at least half of all computations are accepting.

Starting from the initial configuration the random Turing machine proceeds deterministically until it reaches a branch point. At each branch point it randomly picks one of the two alternatives for the next move and then proceeds. The probability that the computation ends up in an accept state is the ratio of the number of accepting computations to the total computations. The probability that the machine answers "yes" when the answer is "no" is 0 and the probability of correctness can be arbitrarily increased by repeating the experiment. [22] p. 114.

By limiting the resources available to a random Turing machine we can define new complexity classes.

Definition 5.3.

- The class R or *random polynomial time* consist of all decision problems solved by polynomial-time RTMs. It is a natural formulation of *Monte Carlo algorithms* since the output 1 is absolutely correct while an output 0 is correct only with the probability of at least $\frac{1}{2}$. R is not obviously closed under complement due to its asymmetric definition.

- The class ZPP or *zero-error probability polynomial time* is defined as an intersection of R and coR
$$\mathrm{ZPP} = \mathrm{R} \cap \mathrm{coR}.$$
ZPP consists of all those problems solved by polynomial-time *Las Vegas algorithms*, feasible randomized algorithms that never lie, but may not answer less than half the time.

A randomized machine uses different acceptance criteria for "yes" and "no". By relaxing this we can define the *probabilistic Turing machines* with the similar assumptions as with the random Turing machines.

Definition 5.4. *Probabilistic Turing machine* (PTM) is a nondeterministic Turing machine whose output for a given input string x is "yes" if more than half of the computations terminate in "yes" states and is "no" if more than half of the computations terminate in "no" states. If the number of yes-computations equals the number of no-computations, the output is "don't know". A PTM *solves* a problem X if and only if the PTM outputs the correct answer for each instance of the problem.

New complexity classes can be defined using the PTMs. In addition to the ones below we have already defined the class PL in 4.9. [22] p. 119.

Definition 5.5.

- The Class PP is the set of all decision problems that can be solved by polynomial-time PTMs.
- BPP is the class of languages recognized by polynomial time probabilistic Turing machines whose error probability is bounded above by some positive constant $\epsilon < \frac{1}{2}$.

The inclusions between the different classes can be found in figure 1.

5.3 Interactive proof systems

In this section *interarctive proof systems* are briefly defined. An interactive proof system is a machine that models computation as the exchange of messages between two parties. The two different parties are called the *verifier* and the *prover*. They interact by exchanging messages in order to ascertain whether a given string x belongs to a language L or not. The complexity class NP can be viewed as a simple proof system where a P-machine checks a certificate produced by the prover in deterministic polynomial time.

We first define formally the interaction between the two parties.

Definition 5.6. Interaction of deterministic functions [4] p. 128.
Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be functions and $k \geq 0$ be an integer which can depend upon the input size. A *k-round interaction* of f and g on input $x \in \{0,1\}^*$, denoted

by $\langle f, g\rangle(x)$ is the sequence of strings $a_1, \dots a_k \in \{0,1\}^*$ defined as follows:

$$\begin{aligned} a_1 &= f(x) \\ a_2 &= g(x, a_1) \\ &\dots \\ a_{i+1} &= f(x, a_1, \dots, a_{2i}) \quad \text{for } 2i < k \\ a_{i+2} &= g(x, a_1, \dots, a_{2i+1}) \quad \text{for } 2i+1 < k \end{aligned}$$

The class IP of interactive proof systems uses probabilistic polynomial-time Turing machine as its machine model.

Definition 5.7. Interactive proof systems [4] p. 130.
For an integer $k \geq 1$ that may depend on the input length, we say that a language L is in IP$[k]$ if there is a probabilistic polynomial-time Turing machine V that can have a k-round interaction with a function $P : \{0,1\}^* \to \{0,1\}^*$ such that

$$\begin{aligned} x \in L &\Rightarrow \exists P \ \text{Prob}[out_V\langle V, P\rangle(x) = 1] \geq \frac{2}{3} \quad \text{(Completeness)} \\ x \notin L &\Rightarrow \forall P \ \text{Prob}[out_V\langle V, P\rangle(x) = 1] \leq \frac{1}{3} \quad \text{(Soundness)} \end{aligned}$$

The probabilities $\frac{1}{3}$ and $\frac{2}{3}$ can be amplified arbitrary close to 0 or 1.

The complexity class IP characterizes the set of languages that have interactive proofs. It is known that IP = PSPACE.

5.4 Arthur-Merlin games

Babai was first to introduce the Arthur-Merlin games in [9]. They are a restricted version of interactive proof systems. The coin tosses are constrained to be *public* meaning that they are known also to the prover.

King Arthur recognizes the supernatural intellectual abilities of Merlin but does not trust him. How should Merlin convince the intelligent but impatient king that a string x belongs to a given language L?

Arthur (A = BPP) has the polynomial-time bounded computational resources and the ability to flip unbiased coins. Merlin's (M = NP) goal is to convince Arthur that a given string x is a yes-instance of decision problem L. When Arthur speaks, he is limited to simply telling Merlin the outcome of some number of coin flips polynomial in $|x|$. When Merlin speaks, his message (also polynomial in $|x|$) can depend on x and all the previous messages. This is an interactive proof system which uses public coins instead of private.

If $L \in$ NP, Merlin will be able to present a *witness* which Arthur can check in polynomial time. The complexity class MA is the set of decision problems that can be decided in polynomial-time by an Arthur-Merlin protocol where Merlin's only move precedes any computation by Arthur. [9]

Definition 5.8. A language L is in MA if there exists a polynomial-time deterministic Turing machine M and polynomials p and q such that for every input string x, $|x| = n$:

$$\begin{aligned} x \in L &\Rightarrow \exists z \in \Sigma^{p(n)} \text{ Prob } [\ M(x,y,z) = 1\ ] \ \geq \frac{2}{3} \\ x \notin L &\Rightarrow \forall z \in \Sigma^{p(n)} \text{ Prob } [\ M(x,y,z) = 1\ ] \ \leq \frac{1}{3} \end{aligned}$$

where y is chosen uniformly at random from $\Sigma^{q(n)}$.

In other words, z is the alleged proof from Merlin and y is the random string Arthur uses.

Definition 5.9. A language L is in AM if there exists a polynomial-time deterministic Turing machine M and polynomials p and q such that for every input string x, $|x| = n$:

$$\begin{aligned} x \in L &\Rightarrow \text{Prob } [\ \exists z \in \Sigma^{p(n)}\ M(x,y,z) = 1\ ] \ \geq \frac{2}{3} \\ x \notin L &\Rightarrow \text{Prob } [\ \forall z \in \Sigma^{p(n)}\ M(x,y,z) = 1\ ] \ \leq \frac{1}{3} \end{aligned}$$

where y is chosen uniformly at random from $\Sigma^{q(n)}$.

The Arthur-Merlin hierarchy collapses to AM if we add more interaction. That is, MA$[k]$ = AM$[k]$ = AM $\forall k > 2$. The relationships between Arthur-Merlin and other classes are as follows. See figure 1 for a bigger picture.

$$\begin{aligned} &\text{NP} &&\subseteq \text{MA} &&\subseteq \text{AM} &&\subseteq \Pi_2^{\text{P}} \\ &\text{coNP} &&\subseteq \text{coMA} &&\subseteq \text{coAM} &&\subseteq \Sigma_2^{\text{P}} \\ &\text{BPP} &&\subseteq \text{MA} &&\subseteq \Pi_2^{\text{P}} \cap \Sigma_2^{\text{P}} \\ &\text{BPP} &&\subseteq \text{coMA} \end{aligned}$$

5.4.1 Graph isomorphism is in coAM

There are different ways of proving that GI $\in$ coAM. This text follows the one by Uwe Schöning that does not use the interactive proof systems. [29]

The following combinatorial definition and theorem are needed for the proof.

Definition 5.10. For two graphs with n vertices define $num(G_1, G_2)$ to be the number of different triples of the form (H, p, i) where H is an isomorphic graph to either G_1 or G_2, and p is an automorphism for G_i, $i \in \{1, 2\}$.

Theorem 5.11. If G_1 and G_2 are isomorphic, then $num(G_1, G_2) = 2n!$, otherwise $num(G_1, G_2) \geq 4n!$.

The objects (H, p, i) that are counted in $num(G_1, G_2)$ can be nondeterministically generated in polynomial time.

Next we define the *random hash function.*

Definition 5.12. A random hash function $H : \Sigma^t \to \Sigma^m$ is given by a Boolean (t, m)-matrix whose elements $h_{ij} \in \{0, 1\}$ are picked uniformly random and independently. Then the j:th bit of $H(a_1, a_2, \ldots, a_t) \in \Sigma^m$ is calculated by

$$(h_{1j} \wedge a_1) \oplus (h_{2j} \wedge a_2) \oplus \ldots \oplus (h_{tj} \wedge a_t),$$

ie. matrix multiplication modulo 2.

A *collision* in a hash function means that

$$\exists x \; \forall H_i : (x \neq y) \wedge (H_i(x) = H_i(y))$$

The next theorem 5.13 states that on every fixed "small" subset of Σ^t a randomly selected collection of hash functions $\{H_i\}$ is likely to be collision free.

Theorem 5.13. Hashing Lemma

- If $X \subseteq \Sigma^t$ has cardinality at most 2^{m-1}, then for randomly selected hash functions $H_1, \ldots, H_{m+1} : \Sigma^t \to \Sigma^m$, with probability at most $\frac{1}{3}$,

$$(\exists\, x \in X)\ (\forall\, i \leq m+1)\ (\exists\, y \in X)\ [\ y \neq x \ \wedge\ H_i(x) = H_i(y)\ ]. \qquad (*)$$

- If X has more than $(m + 1)2^m$ elements, the probability for $(*)$ is 1.

The set $X = \{(H, p, i)\}$ is defined as in the definition 5.10. Now we apply the hashing lemma 5.13 to the set $Y = X^k$. In this case $|Y| = (2n!)^k$ for isomorphic graphs and $|Y| = (4n!)^k$ for non-isomorphic graphs.

Theorem 5.14. There is a set $B \in \mathrm{NP}$ and a polynomial p such that for every pair of graphs G_1, G_2 with n vertices,

$$\mathrm{Prob}\ [\ (G_1, G_2, w) \in B\ ]\ \leq \frac{1}{3}$$

if G_1, G_2 are isomorphic, and

$$\mathrm{Prob}\ [\ (G_1, G_2, w) \in B\ ] = 1$$

if G_1, G_2 are non-isomorphic, where w is chosen uniformly at random from $\Sigma^{p(n)}$. Hence, Graph isomorhism GI $\in$ coAM.

The set B is essentially the collision predicate from theorem 5.13 with w being an encoding of the collection of random hash functions. The fact that $B \in \mathrm{NP}$ follows from the observation that the elements in $X = Y^k$ can be nondeterministically generated. [29]

The best known location of graph isomorphism is pictured in figure 5.2.

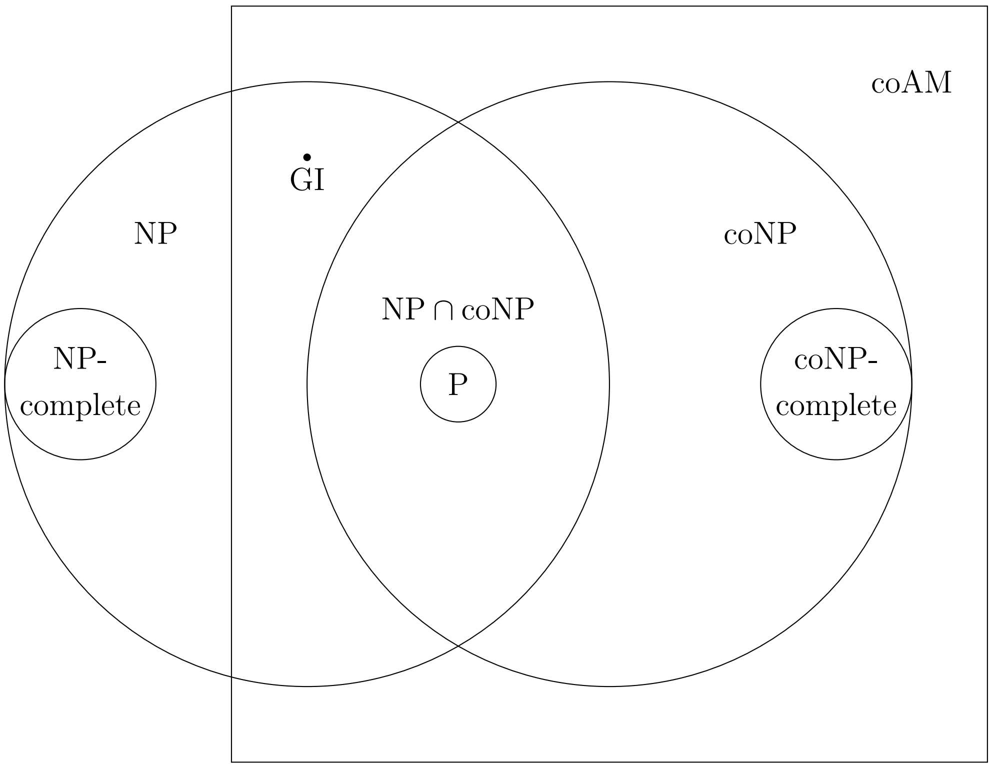

Figure 5.2: GI $\in$ NP $\cap$ coAM

5.5 Graph Isomorphism is in SPP

In this section the complexity class SPP is defined. The graph isomorphism is known to be in this class and a rough sketch of the definitions and theorems needed to prove this are introduced. This text follows the original proof by Arvind and Kurur. [5]

First we introduce a new machine model, the *counting Turing machine*.

Definition 5.15. A non-deterministic Turing Machine is called a *counting Turing machine* (CTM) if its acceptance criterion is based on the *number* of accepting and/or rejecting paths. Let M be a CTM. The function $\#M : \Sigma^* \to Z^+$ is defined such that $\forall x \in \Sigma^*$, $\#M(x)$ is the number of accepting computation paths of M on input x. The machine $\overline{M}$ is the machine identical to M but with the accepting and rejecting states interchanged. $\#\overline{M}(x)$ is the number of rejecting paths of M on input x.

By adding restrictions to the counting Turing machine we can define new complexity classes. To do that we need the following definition.

Definition 5.16. If M is a CTM, define the function $gap_M : \Sigma^* \to Z$ as follows:

$$gap_M = \#M - \#\overline{M}$$

A class C of languages is *gap-definable* if there exists disjoint sets $A, R \subset \Sigma^* \times Z$ such that, for any language L, $L \in C$ if and only if there exists a CTM M with

$$x \in L \Rightarrow (x, gap_M(x)) \in A$$
$$x \notin L \Rightarrow (x, gap_M(x)) \in R$$

The sets A and R are called the *accepting* and *rejecting* sets.

The complexity class SPP is now defined using the counting Turing machine and gap-definability. It is the smallest reasonable gap-definable class.

Definition 5.17. SPP is the class of all languages L such that there exists M such that $\forall x$,

$$x \in L \Rightarrow gap_M(x) = 1$$
$$x \notin L \Rightarrow gap_M(x) = 0.$$

In other words, if $x \in L$ there is one more accepting than rejecting paths. If $x \notin L$ there are the same number of each.

The graph isomorphism problem can be reduced to a related *functional problem* AUTO: given a graph X as an input, the problem is to output a strong generator set $Aut(X)$. The $Aut(X)$ means the underlying *automorphism group* of the graph X. The automorphisms are permutations on the vertex set and they form a group under permutation composition.

Finding the symmetric group that is essential in solving the AUTO can be reduced to a more general problem FIND-GROUP. To each instance $(x, 0^n)$ there is associated an unknown subgroup $G_x \leq S_n$ for which there is a polynomial time test. That is, the polynomial time membership function is given and it takes x and $g \in S_n$ as input and evaluates "true" if and only if $g \in G_x$. The FIND-GROUP problem is to compute a strong generator set for G_x given $(x, 0^n)$ as input.

In order to prove that GI $\in$ SPP, it suffices to show that AUTO $\in$ $\mathrm{FP}^{\mathrm{SPP}}$. To prove this we need to show that there exists a deterministic Turing machine M, with oracle $A \in$ SPP which takes a graph X as an input and which outputs a strong generator set for $Aut(X)$. This is done by showing that the generic problem FIND-GROUP has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm.

Theorem 5.18. Graph isomorphism is in SPP. [5]

BIBLIOGRAPHY

[1] Eric Allender. Circuit complexity before the dawn of the new millennium. Technical report, 1997.

[2] Eric Allender and Mitsunori Ogihara. Relationships among PL, #L, and the determinant. In *Structure in Complexity Theory Conference, 1994., Proceedings of the Ninth Annual*, pages 267–278, 1994.

[3] Carme Álvarez and Birgit Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3 – 30, 1993.

[4] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[5] V. Arvind and P.P. Kurur. Graph isomorphism is in SPP. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 743 – 750, 2002.

[6] V. Arvind and Jacobo Torán. Isomorphism testing: Perspective and open problems. *Bulletin of the EATCS*, pages 66–84, 2005.

[7] J. Aspnes, R. Beigel, M. Furst, and S. Rudich. The expressive power of voting polynomials. *Combinatorica*, 14(2):135–148, 1994.

[8] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[9] László Babai. Trading group theory for randomness. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 421–429, New York, NY, USA, 1985. ACM.

[10] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 171–183, New York, NY, USA, 1983. ACM.

[11] J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I.* EATCS Monographs on Theoretical Computer Science. Springer London, Limited, 1988.

[12] Allan Borodin, S.A. Cook, P.W. Dymond, W.L. Ruzzo, and M. Tompa. Two applications of complementation via inductive counting. In *Structure in Complexity Theory Conference, 1988. Proceedings., Third Annual*, pages 116–125, Jun 1988.

[13] Stephen A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 338–345, New York, NY, USA, 1979. ACM.

[14] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–22, March 1985.

[15] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367 –1372, October 2004.

[16] Scott Fortin. The graph isomorphism problem. Technical report, 1996.

[17] Merrick Furst, JamesB. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

[18] Mark Goldberg. The graph isomorphism problem. In *Handbook of Graph Theory*, pages 68–78. CRC Press, 2003.

[19] E.R. Griffor. *Handbook of Computability Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1999.

[20] William Hesse. Division is in uniform TC^0. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, ICALP '01, pages 104–114, London, UK, UK, 2001. Springer-Verlag.

[21] N. Immerman. Nondeterministic space is closed under complementation. In *Structure in Complexity Theory Conference, 1988. Proceedings., Third Annual*, pages 112–115, Jun 1988.

[22] David S. Johnson. Handbook of theoretical computer science (vol. a). chapter A catalog of complexity classes, pages 67–161. MIT Press, Cambridge, MA, USA, 1990.

[23] José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *CoRR*, abs/1108.1060, 2011.

[24] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *CoRR*, abs/1301.1493, 2013.

[25] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th*

Annual Symposium on Switching and Automata Theory (swat 1972), SWAT '72, pages 125–129, Washington, DC, USA, 1972. IEEE Computer Society.

[26] Piergiorgio Odifreddi. *Classical Recursion Theory : The Theory of Functions and Sets of Natural Numbers.* Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, New-York, Oxford, Tokyo, 1989.

[27] Piergiorgio Odifreddi. *Classical Recursion Theory, vol II.* Studies in Logic and the Foundations of Mathematics. Elsevier, 1999.

[28] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[29] Uwe Schöning. Graph isomorphism is in the low hierarchy. In Franz Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 114–124. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0039599.

[30] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 2nd edition, 1996.

[31] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[32] Jacobo Torán. On the hardness of graph isomorphism. *SIAM J. Comput.*, 33:1093–1108, May 2004.

[33] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

INDEX