Computable Isomorphisms of Directed Graphs and Trees

by Hakim J. Walker

B.A. in Mathematics, May 2009, Boston University
M.A. in Mathematics, May 2014, The George Washington University

A Dissertation submitted to

The Faculty of
The Columbian College of Arts and Sciences
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2017

Dissertation directed by

Valentina Harizanov
Professor of Mathematics

The Columbian College of Arts and Sciences of The George Washington University certifies that Hakim J. Walker has passed the Final Examination for the degree of Doctor of Philosophy as of May 1, 2017. This is the final and approved form of the dissertation.

Computable Isomorphisms of Directed Graphs and Trees

Hakim J. Walker

Dissertation Research Committee:

Valentina Harizanov, Professor of Mathematics, Dissertation Director

Jozef Przytycki, Professor of Mathematics, Committee Member

Wesley Calvert, Associate Professor of Mathematics, Southern Illinois University, Committee Member

Dedication

For my grandmother, Fidelia Walker, and my aunts, Nicole Reid and Maxine Archer.

Acknowledgements

First, I must express my sincere and eternal gratitude to my advisor, Valentina Harizanov, who has unconditionally supported me in every way imaginable over the last five years. Valentina has taught me so many invaluable and unforgettable lessons about mathematics, logic, computability theory, and academia as a whole. She has selflessly invested an immeasurable amount of time, energy, and resources into my education and professional development. She has consistently inspired me, pushed me, promoted me, and defended me, despite considerable conflicts and obstacles along the way. Valentina made me the mathematician that I am today, and she did it all with warmth, kindness, and patience. I owe all of this to her.

This dissertation would not have been possible without the wisdom and guidance of other mathematicians and students, so I would like to recognize them here. Thanks to Wesley Calvert, Tim MicNicholl, and Russell Miller for providing me with critical feedback and corrections, and for exposing me to a wealth of new ideas, solutions, and research questions. I would especially like to thank my three academic sisters, Iva Bilanovic, Trang Ha, and Leah Marshall. Thanks to Iva for always giving me thoughtful constructive criticism and positive reinforcement, for having hundreds of interesting discussions with me about mathematics (and everything else), and for being a wonderful office mate and friend. Thanks to Trang for being a constant in my graduate mathematical life since day one, from qualifying exams to conferences, and also for being a wonderful office mate and friend. Lastly, thanks to Leah for being the best deputy advisor I could have had, for being an excellent role model to

me, both professionally and personally...and also for being a wonderful office mate and friend.

Speaking of friends, there are a number of them to whom I am forever indebted, as I would not have survived graduate school without their loyalty, love, and support. They are, in no particular order: Francis Pina, SarahEmily Lekberg, Adrienne Baker, Jamie Perkins, Anna Bakanova, James Dargan, Nathan Chow, Andrew Jerell Jones, Abaigeal Pacholk, Giselle Robleto, Dara Randall, Arjona Andoni, Morgan Smith, Jonathan Remple, Jennifer Kaizer, Marianna Breytman, Andrea Atehortua, Taylor Hartwin, James Conkell, Kresenda Keith, and Jessica Indarte.

As much as my colleagues and friends have helped me to succeed in graduate school, I would not have made it even that far without my family. Thanks to my mom for giving me exactly what I needed to discover and follow my passion, and thanks to my dad for always being my biggest cheerleader. They have both sacrificed so much for my success, and I am happy to be able to give them a return on their investment. Thanks to my brother, Nashan, and my sisters, Tiana and Kenya, for putting up with me, and for constantly reminding me where I come from. Thanks to all of my aunts, uncles, and cousins who have reached out to me over the last few years, even when I failed to do the same.

Finally, to my late grandmother, Fidelia Walker, who fed and sheltered me for a week while I studied to enter graduate school...thank you for always taking care of me, and for believing in me. I hope that I have made you proud.

Abstract

Computable Isomorphisms of Directed Graphs and Trees

One of the main goals of computable model theory is to classify mathematical structures up to computable isomorphism. Two computable structures $\mathcal{A}$ and $\mathcal{B}$ are computably isomorphic if there exists a computable bijection from $\mathcal{A}$ to $\mathcal{B}$ that preserves all of the functions and relations in the structure. Furthermore, we say that $\mathcal{A}$ is computably categorical if every two computable copies of $\mathcal{A}$ are computably isomorphic. Significant work on computable categoricity has been done for a variety of mathematical structures, including linear orders, abelian groups, Boolean algebras, trees, and many others.

We define a (2,1):1 structure, which consists of a countable set A, together with a function f on that set such that for every element x in A, f maps either exactly one or exactly two elements of A to x. These structures are in a similar class as the injection structures, two-to-one structures, and (2,0):1 structures studied by Cenzer, Harizanov, and Remmel, all of which can be thought of as directed graphs. In particular, (2,1):1 structures can contain two types of connected components: K-cycles, which consist of a directed cycle of k elements, each of which has an infinite (or empty) binary tree, and $\mathbb{Z}$-chains, which consist of an infinite directed chain of elements, each of which has attached an infinite (or empty) binary tree.

We investigate the isomorphism problem for computable (2,1);1 structures, and show that deciding if two such structures are isomorphic is Π^0_4 in the arithmetical hierarchy. Furthermore, we prove that all (2,1):1 structures are Δ^0_4-categorical. We

then introduce two additional functions on our structures: the *branching function*, which determines if an element has one or two pre-images, and the *branch isomorphism function*, which determines if an element with two branches has isomorphic trees. We prove that all computable (2,1):1 structures with a computable branching function in every computable copy are Δ^0_3-categorical, and that all such structures without $\mathbb{Z}$-chains are Δ^0_2-categorical. We also give examples of a (2,1):1 structure with no computable branching function, and a structure with computable branching but no computable branch isomorphism function.

We make significant progress toward classifying computable categoricity for (2,1):1 structures, by providing sufficient and necessary conditions for such graphs to be computably categorical. We construct a computable (2,1):1 structure that is computably categorical, but not relatively computably categorical, demonstrating the difficulty of classifying computable categoricity for such structures. Lastly, we present an interesting connection between our theory and the Collatz conjecture, also known as the $3n+1$ conjecture. We observe that the Collatz graph is also a computable (2,1):1 structure with computable branching and branch isomorphism functions. Moreover, we use this result to prove that the connected component of the Collatz graph containing 1 is computably categorical, and that if the full Collatz structure is not computably categorical, then the Collatz conjecture does not hold.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Foundations of Computability Theory

The idea of an algorithm in mathematics has existed since antiquity, with the Euclidean algorithm being one of the earliest and most famous documented examples. The word *algorithm* itself is derived from the name of the 9^{th} century Persian mathematician al-Khwarizmi, who is credited with the first systematic solution to linear and quadratic equations. Consequently, an algorithm is informally defined as a step-by-step procedure to perform a given task, or to solve a problem. However, a widely-accepted formal mathematical definition of an algorithm was not introduced until the 20^{th} century.

In 1928, David Hilbert asked if there exists an algorithm to decide if a given statement in first-order logic is *universally valid*, i.e., true in every universe in which the axioms of first-order logic are satisfied. This question is known as the *Entscheidungsproblem*, or *decision problem*. (Due to Gödel's Completeness Theorem, this

problem is equivalent to asking for an algorithm to decide if a given first-order sentence is *provable* from the axioms and rules of first-order logic.) It soon became evident that the problem required a rigorous mathematical definition of the concept of an algorithm. In the 1930's, there were numerous attempts at a formulation, including Kurt Gödel's general recursive functions, Stephen Kleene's μ-recursive functions, and Alonzo Church's λ-calculus. However, it was Alan Turing who, in his seminal 1936 paper *On Computable Numbers, with an Application to the Entscheidungsproblem* [46], introduced the definitive notion of an algorithm. (Amusingly, Turing's notion turned out to be equivalent to the aforementioned formulations.)

Turing conceived of an abstract computing device without time or space limitations, now referred to as a *Turing machine*. A Turing machine consists of a doubly-infinite tape divided into cells, on which could appear a 0 or a 1, and a head capable of moving left and right along the tape and reading and writing symbols on one cell at a time. Each machine is programmable in the sense that it can be given a finite list of states in which the machine could be (including a HALT state), and a finite list of instructions for the head to carry out, given the state of the machine and the symbols on the tape. This list of instructions is known as a *Turing program*.

It is easy to show that a Turing machine can calculate every elementary operation on natural numbers, i.e., addition, multiplication, exponentiation, etc. However, as the complexity of the operations increases, it becomes less obvious that the function can be calculated on such a simplistic machine. Thus, the question arises whether all functions that we *believe* to be calculable can be *formally* calculated on a Turing machine. Alan Turing, along with his advisor Alonzo Church, espoused the philo-

sophical stance that a function is intuitively calculable if and only if it is calculable by a Turing program. This postulate became known as the *Church-Turing Thesis*. Although it cannot be formally proven, as there is no formal definition of "intuitively calculable", there are no known counterexamples to the thesis, and it is widely accepted as true among mathematicians and computer scientists. Therefore, we ascribe to the following foundational definition.

Definition 1.1.1. A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if there is a Turing program P that computes f. That is, given $x \in \mathbb{N}$ as the starting input on the tape of the Turing machine, when the program P is run on input x, P halts in a finite number of steps and outputs $f(x)$ on the tape.

There are a few things to note about Definition 1.1.1. First, we do not strictly require that the function f be from $\mathbb{N}$ to $\mathbb{N}$. Indeed, we may use any countable set for the domain and codomain, provided we have a systematic Gödel coding of the countable set using elements from $\mathbb{N}$. Consequently, we often just use $\mathbb{N}$ as the domain and codomain for convenience. Second, the word "computable" is frequently replaced with the following synonyms: *decidable, recursive, effective,* and *algorithmic*. Thus, a Turing program becomes our formal equivalent of an algorithm. Third, although the cells on a Turing machine can only contain 0's and 1's, we may still represent all inputs and outputs in $\mathbb{N}$ by simply representing them as binary digits. (Some variants of Turing machines allow any finite set of symbols to be the underlying alphabet. A more advanced version of a Turing machine, called an *Unlimited Register Machine* (URM), allows any natural number to be written in a cell.) Finally, each "step" in

the computation of P refers to the carrying out of one instruction in P, and we say that P halts when it either reaches the HALT state or when no further instructions can be carried out.

Since each Turing program consists of a finite list of finite instructions, there are only countably many Turing programs, which can be effectively coded using natural numbers. Thus, we can algorithmically list *all* possible Turing programs in the following manner:

$$\varphi_0, \varphi_1, \varphi_2, \cdots$$

where φ_e represents the e^{th} Turing program, and e is the *index* of the program. Moreover, since each computable function from $\mathbb{N}$ to $\mathbb{N}$ is computable by some program, we can associate computable functions with Turing programs, and think of the list above as a list of all computable unary functions on the natural numbers.

This list is not ideal in the sense that it will contain many "nonsense" functions that do not compute much of anything, or functions that do not halt on certain inputs and instead get stuck in an infinite loop of computation. Thus, we distinguish *total* computable functions that halt on all inputs, from *partial* computable functions that may or may not halt on all inputs. Then the list above becomes a list of all *partial* computable unary functions on the natural numbers. Interestingly, it is impossible to create an effective list of only the *total* computable functions.

Also, our list may not be ideal in the sense that each computable function may not have a *unique* Turing program that computes it. Indeed, in any standard enumeration of the Turing programs, every computable function will have *infinitely many* programs that compute it, by simply "padding" one program with useless instruc-

tions. Although Friedberg [17] showed that it is possible to effectively enumerate all Turing programs without repetition, we often use the standard enumeration in practice.

Now that we have defined a computable function, we can define computable *sets* and *relations* via their characteristic functions.

Definition 1.1.2. A set $X \subseteq \mathbb{N}$ is *computable* if its characteristic function χ_A is computable, where:

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

Furthermore, let $\vec{x}$ be an n-tuple of natural numbers. An n-ary relation R is *computable* if its characteristic function χ_R is computable, where:

$$\chi_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \text{ holds}, \\ 0 & \text{if } R(\vec{x}) \text{ does not hold}. \end{cases}$$

Informally, a set A is computable if there is some effective procedure to decide if a given object is an element of A. It should be noted that one could have also begun by first defining a computable set, and then defining computable functions and relations by interpreting them as sets.

Since there are uncountably many functions $f : \mathbb{N} \to \mathbb{N}$, but only countably many computable unary functions, we may conclude that "most" functions on the natural numbers are not computable. Similarly, since there are uncountably many sets $X \subseteq$

$\mathbb{N}$, most subsets of the natural numbers are not computable either. Ironically, non-computable sets are harder to come by, despite being more abundant. Most sets that we can "naturally" think of will be computable, since we would have some intuitive, effective way of describing that set. For example, all finite sets are computable, as is the complement of any computable set. However, in [46], Turing gave an explicit description of a non-computable set.

Definition 1.1.3. The *Halting Set,* denoted by K, is the set of all Turing programs that eventually halt when given its own index as the input. That is,

$$K = \{e : \varphi_e(e) \downarrow\},$$

where $\downarrow$ denotes that the program halts in a finite number of steps and outputs a natural number.

We often make use of a *time bound* t on partial computable functions, and write $\varphi_{e,t}(x)$ to denote the output when program e is run on input x for exactly t steps of its computation. Consequently, we can present the Halting Set as:

$$K = \{e : (\exists t)(\varphi_{e,t}(e) \downarrow)\}.$$

Turing proved that K is not computable using a similar diagonal argument as the one introduced by Cantor to show that the reals are uncountable. He then extrapolated this idea to show that the set of universally valid first-order sentences is also not computable, thus providing a negative solution to the Entscheidungsproblem.

It should make intuitive sense that the Halting Set is not decidable. Given a program e, we run $\varphi_e(e)$ to see if it will eventually halt. If we wait for a very long time and it doesn't seem to halt, we have no way of effectively knowing in general if

the program will halt if we wait a few more steps, or if the program will never halt and our waiting is in vain. However, if $\varphi_e(e)$ *does* halt, we will see this at some point during the computation, and we can then say with certainty that $e \in K$. This leads to the idea of *computable enumerability*.

Definition 1.1.4. A set $A \subseteq \mathbb{N}$, or an n-ary relation R, is *computably enumerable* (abbreviated *c.e.*) if its *partial characteristic function,* given by:

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ \uparrow & \text{if } x \notin A \end{cases} \qquad \chi_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \text{ holds}, \\ \uparrow & \text{if } R(\vec{x}) \text{ does not hold} \end{cases}$$

is computable, where $\uparrow$ denotes "does not halt" or "computes forever".

Equivalently, a set is c.e. if it is the domain of some partial computable function φ_e. More informally, a set A is c.e. if there is some effective procedure to enumerate the elements that are in A.

Using the Church-Turing Thesis and Definition 1.1.4, we see that K is a c.e. set. It can also be easily shown that all computable sets are c.e. sets, and that a set A is computable if and only if both A and $\overline{A}$ are c.e. sets. Thus, we can deduce that $\overline{K}$ is not even computably enumerable. The c.e. subsets of $\mathbb{N}$, usually denoted by $\mathcal{E}$, form a countable lattice under union and intersection, and this structure remains a rich area of study today. See [43] and [44] for a comprehensive treatment of computably enumerable sets.

We think of non-computable sets as having more information content than computable ones. This gives rise to the idea of *Turing reducibility* and *Turing equivalence*.

Definition 1.1.5. Let A and B be subsets of $\mathbb{N}$.

(a) We say that A is *Turing reducible to* B, denoted by $A \leq_T B$, if there is a Turing program that can compute A, given membership information about B.

(b) We say that A is *Turing equivalent to* B, denoted by $A \equiv_T B$, if $A \leq_T B$ and $B \leq_T A$.

(c) We say that A is *strictly Turing reducible to* B, denoted by $A <_T B$, if $A \leq_T B$ and $A \not\equiv_T B$.

Informally, $A <_T B$ means that B has more information content than A, $A \leq_T B$ means that B has at least as much information content as A, and $A \equiv_T B$ means that A and B have the same information content. We often describe the set B in Definition 1.1.5(a) as an *oracle*, of which we can ask any membership question in order to determine whether a given number x is in A. Given B as such an oracle, we can make an effective list of all B*-computable functions*, that is, all unary functions that are partial computable from B:

$$\varphi_0^B, \varphi_1^B, \varphi_2^B, ...$$

Thus, $A \leq_T B$ if and only if $\chi_A = \varphi_e^B$ for some e. We also say that A is *computable in* B.

It is obvious that $A \leq_T A$ and $A \leq_T \overline{A}$ for all $A \subseteq \mathbb{N}$, that if A is computable and X is any set, then $A \leq_T X$, and that if A is computable and $X \leq_T A$, then X is computable. It is equally obvious that $A \equiv_T A$ and $A \equiv_T \overline{A}$ for all $A \subseteq \mathbb{N}$, and that if A and B are computable sets, then $A \equiv_T B$. It can also be shown that if A is

any c.e. set, then $A \leq_T K$. (The converse, however, is not true, since $\overline{K} \leq_T K$ but $\overline{K}$ is not c.e.) Hence, K is the most complicated c.e. set in the sense that any other c.e. set can be computed from it. Lastly, it is straightforward to check that $\leq_T$ is a partial order on subsets of $\mathbb{N}$, $<_T$ is a strict partial order, and $\equiv_T$ is an equivalence relation.

However, $\leq_T$ is not a *total* linear order. Kleene and Post constructed two sets A and B that are *Turing incomparable*, i.e., $A \not\leq_T B$ and $B \not\leq_T A$ (see [10] and [43] for details). Later, Friedberg [18] and Muchnik [37] independently improved this result by constructing two c.e. sets A and B that are incomparable.

Since $\equiv_T$ is an equivalence relation on subsets of natural numbers, we can discuss the equivalence classes of $\mathbb{N}$ under $\equiv_T$, which we refer to as *Turing degrees*.

Definition 1.1.6. Let $A \subseteq \mathbb{N}$. The *Turing degree of A,* denoted by $deg(A)$ or $\mathbf{a}$, is defined as:

$$deg(A) = \mathbf{a} = \{X \subseteq \mathbb{N} : A \equiv_T X\}.$$

Turing degrees are often referred to as *degrees of unsolvability*, as each degree informally describes how non-computable a set is. We denote the degree of all computable sets as $\mathbf{0}$. Furthermore, we denote the set of all Turing degrees by $\mathcal{D}$, just as we denote the set of all computably enumerable degrees by $\mathcal{E}$. Naturally, the Turing reducibility ordering $\leq_T$ induces a partial ordering $<$ on $\mathcal{D}$, and Turing equivalence $\equiv_T$ induces an equivalence relation $=$ on $\mathcal{D}$. So, if $A \in \mathbf{a}$ and $B \in \mathbf{b}$, then $A \leq_T B$ if and only if $\mathbf{a} \leq \mathbf{b}$, and $A \equiv_T B$ if and only if $\mathbf{a} = \mathbf{b}$.

Using a basic counting argument, one can prove that there are uncountably many

Turing degrees in $\mathcal{D}$, and only countably many degrees below any given degree $\mathbf{d}$. Furthermore, we can define the *join of two sets* $A, B \subseteq \mathbb{N}$, given by:

$$A \oplus B = \{2x : x \in A\} \cup \{2x+1 : x \in B\},$$

and thus define the *join of two degrees* $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a} \vee \mathbf{b} = deg(A \oplus B)$, where $A \in \mathbf{a}$ and $B \in \mathbf{b}$. Thus, it can be shown that the Turing degrees form an upper semilattice, where the supremum of two degrees $\mathbf{a}$ and $\mathbf{b}$ is given by $\mathbf{a} \vee \mathbf{b}$. However, $\mathcal{D}$ is not a full lattice, as Lachlan and Yates independently proved that there are incomparable c.e. degrees $\mathbf{a}$ and $\mathbf{b}$ with no infimum (see [43]).

Although the Turing degrees are not totally ordered, we can describe infinite chains of degrees by defining the *Turing jump* of a set.

Definition 1.1.7. Let $A \subseteq \mathbb{N}$. The *Turing jump of A,* denoted by A', is the following set:

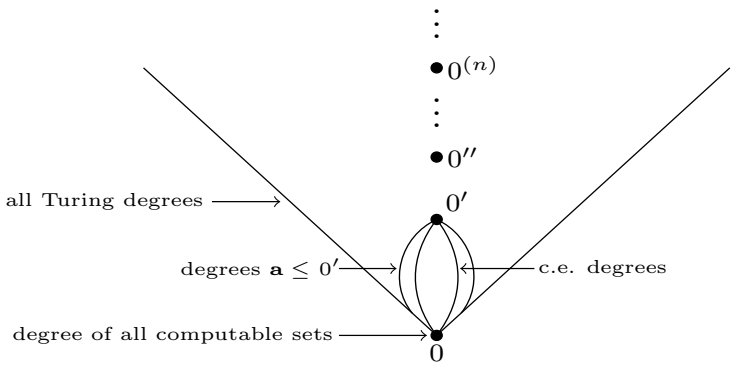
$$A' = \{e : \varphi_e^A(e) \downarrow\}$$

Naturally, if $\mathbf{a}$ represents the degree of A, then $\mathbf{a}'$ represents the degree of A'. In particular, by Definition 1.1.3 and 1.1.7, the jump of any computable set is simply the Halting set, and we denote its degree by $\mathbf{0}'$. (We often refer to the Halting set as $\emptyset'$, using the empty set as a representative for all computable sets.) For all sets A, we have that $A <_T A'$. Although A' is not computable from A, A' is computably enumerable in A. Also, the jump operator preserves the Turing order and equivalence of sets; that is, $A \leq_T B \implies A' \leq_T B'$. The converse, however, does not hold.

Inductively, given a set A and $n \geq 1$, we may define the $(n+1)^{th}$ *jump of* A as: $A^{(n+1)} = \{e : \varphi_e^{A^{(n)}}(e) \downarrow\}$. It can then be shown that for all n, $A^{(n)} <_T A^{(n+1)}$.

Specifically, $\emptyset^{(n)} <_T \emptyset^{(n+1)}$ and thus $\mathbf{0}^{(n)} < \mathbf{0}^{(n+1)}$. This generates the hierarchy of Turing degrees pictured below.

Figure 1.1: The Turing degree hierarchy.



Another common way to describe the complexity of subsets of natural numbers is via syntactic definability.

Definition 1.1.8. Let $A \subseteq \mathbb{N}$.

(a) A is called a Σ_n^0 *set* if A can be expressed as:

$$A = \{y \in \mathbb{N} : (\exists x_1)(\forall x_2)(\exists x_3)...(Qx_n)[R(\vec{x}, y)]\},$$

where Q is either $\exists$ or $\forall$ (depending on n), and $R(\vec{x}, y)$ is a computable $(n+1)$-ary relation.

(b) A is called a Π_n^0 *set* if A can be expressed as:

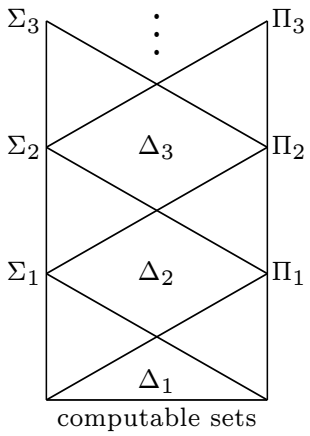
$$A = \{y \in \mathbb{N} : (\forall x_1)(\exists x_2)(\forall x_3)...(Qx_n)[R(\vec{x}, y)]\},$$

where Q is either $\exists$ or $\forall$ (depending on n), and $R(\vec{x}, y)$ is a computable $(n+1)$-ary relation.

(c) A is called a Δ_n^0 *set* if A is both a Σ_n^0 set and a Π_n^0 set.

The scheme described in Definition 1.1.8 gives rise to the *arithmetical hierarchy*, which is pictured in Figure 1.2.

Figure 1.2: The arithmetical hierarchy.



There is a deep connection between the Turing degree hierarchy and the arithmetical hierarchy, discovered by Emil Post. It is captured by the following theorem.

Theorem 1.1.9 (Post's Theorem)**.** *For every $A \subseteq \mathbb{N}$ and $n \geq 0$,*

(1) *A is a Σ^0_{n+1} set $\iff$ A is computably enumerable in $\emptyset^{(n)}$.*

(2) *A is a Δ^0_{n+1} set $\iff$ $A \leq_T \emptyset^{(n)}$.*

Post's Theorem actually says more than this (see [43]), but the version given in Theorem 1.1.9 is sufficient for our needs. One consequence of Post's Theorem is that if a set A is Σ^0_1, then A is computably enumerable, and thus Turing reducible to $\emptyset'$. Another consequence is that A is a Δ^0_2 set if and only if $A \leq_T \emptyset'$. We will often utilize these facts without explicitly stating them.

We conclude this section with one more computability-theoretic result, known as *the Limit Lemma*, that we will frequently use implicitly in future proofs. First, we need the following definition.

Definition 1.1.10. Let $\{f_s(x)\}_{s\in\omega}$ be a sequence of functions $f_s : \mathbb{N} \to \mathbb{N}$, and let $A \subseteq \mathbb{N}$.

(a) The sequence $\{f_s(x)\}_{s\in\omega}$ *converges* (*pointwise*) *to* $f(x)$, written as $f = \lim_s f_s$, if for all x, $f_s(x) = f(x)$ for all but finitely many s.

(b) The sequence $\{f_s(x)\}_{s\in\omega}$ is *A-computable* if there is an A-computable function $\hat{f}(x, s)$ such that $f_s(x) = \hat{f}(x, s)$ for all x, s.

Theorem 1.1.11 (The Limit Lemma)**.** *Let $f : \mathbb{N} \to \mathbb{N}$. Then $f \leq_T A'$ iff there is an A-computable sequence $\{f_s\}_{s\in\omega}$ such that $f = \lim_s f_s$.*

We refer the reader to [10], [12], [43], and [44] for further background on computability theory, including definitions, notations, theorems, and proofs.

1.2 Computable Model Theory

Before computability theory, mathematicians were primarily concerned with studying the properties of classical mathematical structures, such as groups and vector spaces, and the theories derived from those structures. We understand a *structure*, or *model*, $\mathcal{A}$ to be a set, usually called the *domain* or *universe* of $\mathcal{A}$, together with certain functions, relations, and special named constants. Every structure is equipped with a language $\mathcal{L}$ containing the symbols of $\mathcal{A}$ and the symbols from the underlying logic being used (usually first-order logic), as well as syntactic rules determining how sentences are formed in the language. We also generally assume that there is some deductive calculus with which we may prove sentences that are true in $\mathcal{A}$, thus

developing the theory of $\mathcal{A}$. These basic characteristics laid the foundation of classical model theory.

However, the advent of computability theory led to natural questions about the *algorithmic* properties of such structures. One of the earliest results in this direction is due to van der Waerden [47]. In 1930, he showed that an "explicit" field $\mathcal{F} = (F, +, \cdot, \mathbf{0}, \mathbf{1})$ does not necessarily have a splitting algorithm, i.e., an algorithm to split polynomials in $F[x]$ into irreducible factors. Although it was not understood in this way at the time, van der Waerden's theorem about splitting algorithms was a computability-theoretic result about a type of mathematical structure. This idea eventually motivated the notion of a *computable structure*.

Definition 1.2.1. Let $\mathcal{A}$ be a structure. We say that $\mathcal{A}$ is *computable* if the domain of $\mathcal{A}$ is computable and the functions and relations in $\mathcal{A}$ are uniformly computable. Equivalently, $\mathcal{A}$ is computable if the domain of $\mathcal{A}$ is computable and the *atomic diagram of* $\mathcal{A}$, i.e., the set of all quantifier-free sentences that are true in $\mathcal{A}$, is computable.

Yet another way to define a computable structure is via its characteristic function. The *characteristic function of a structure* $\mathcal{A}$ is the characteristic function of its atomic diagram. That is, the characteristic function of $\mathcal{A}$ takes an atomic sentence in the language of $\mathcal{A}$ (coded by a natural number) as input, and outputs 1 if the sentence is true in $\mathcal{A}$ and 0 otherwise. Then we may say that $\mathcal{A}$ is computable if its characteristic function is computable. Furthermore, for a class of computable structures $\mathcal{C}$, we define the *index set of* $\mathcal{C}$ as: $\{e \in \mathbb{N} : \varphi_e$ is the characteristic function of a structure in $\mathcal{C}\}$.

With computable structures, we generally assume that the structure is countable, which means that it has a countable domain and only countably many function and relation symbols. Moreover, through an algorithmic encoding of the domain via natural numbers, we also usually assume that the domain of a computable structure is ω. Additionally, we assume that a computable structure is over some computable language; in fact, most standard examples of computable models have a finite language.

Broadly speaking, computable model theory studies the algorithmic content and properties of computable structures. This investigation was largely motivated by the discovery that computable structures often have surprising non-computable characteristics. For example, van der Waerden's aforementioned result, stated in modern computability-theoretic terms, says that a field may not have a computable splitting algorithm, even if the field itself is computable. Another more famous example is given by Gödel's remarkable incompleteness theorems of 1931. By proving that the standard model of number theory given by Peano's axioms is incomplete, Gödel essentially showed that although arithmetic itself, and the axioms governing it, are computable, the set of true statements of arithmetic are not even computably enumerable.

Throughout the twentieth century, computable model theory continued to develop into a tool powerful enough to solve many important problems across mathematics. For example, the *word problem* in group theory is to determine if an arbitrary word in a finitely-generated group is equivalent to the identity. In the late 1950's, Boone [3] and Novikov [38] independently proved the existence of a finitely presented group whose word problem is undecidable. Perhaps the most notable example involves yet

another question posed by David Hilbert. In 1900, Hilbert announced an infamous list of 23 unresolved mathematical problems, the tenth one asking for a general algorithm to determine if a given Diophantine equation with integer coefficients has integer solutions. Seventy years later, Matiyasevich gave a negative solution to Hilbert's Tenth Problem, a problem in number theory, by using computability-theoretic tools and building off of work done by Davis, Putnam, and Robinson (see [35]).

One of the key foundational results in computable model theory is due to Fröhlich and Shepherdson. They showed in [19] that there are two "explicit" fields that are isomorphic but not "explicitly" isomorphic. Like van der Waerden, they used "explicit" in place of the more modern word "computable". Thus, Fröhlich and Shepherdson proved that it is possible for us to know that two computable structures are isomorphic, and yet not know how to compute an isomorphism between them. This astonishing result, along with similar discoveries by Mal'cev [33] and Rabin [39] led to the next two definitions.

Definition 1.2.2. We say that two computable structures $\mathcal{A}$ and $\mathcal{B}$ that are isomorphic to each other are *computably isomorphic* if there exists a computable isomorphism h from $\mathcal{A}$ to $\mathcal{B}$.

Computable isomorphisms between structures are important because they preserve not only the functions and relations of a structure, but also the algorithmic properties of the structure. As shown in [19], it is quite possible for two isomorphic computable structures to not be computably isomorphic. Thus, we have the notion of *computable categoricity*.

Definition 1.2.3. A computable structure $\mathcal{A}$ is *computably categorical* if every two computable copies of $\mathcal{A}$ are computably isomorphic.

Computable categoricity is a central concept in computable model theory that has been studied extensively over the last fifty years. Due to its independent development in the east and the west, some older Russian texts refer to the same concept as *autostability*. See [16] or [24] for a more detailed exposition of computable model theory and computable categoricity.

We generally seek to classify computable structures up to computable isomorphism. That is, within a class of structures, we wish to provide characterizations of those computable structures that are computably categorical. This has been done already for various classes of mathematical structures. For example, Goncharov and Dzgoev [15], and independently Remmel [41], proved that a computable linear order is computably categorical if and only if it has only finitely many successor pairs. Additionally, Goncharov and Dzgoev [15], LaRoche [30], and Remmel [40] independently proved that a computable Boolean algebra is computably categorical if and only if it has only finitely many atoms. Goncharov, Lempp, and Solomon [23] characterized computably categorical ordered abelian groups as those with finite rank. Calvert, Cenzer, Harizanov, and Morozov gave a full characterization of computably categorical equivalence structures (structures consisting of a countable set and an equivalence relation on that set) in [4].

Unfortunately, it can be extremely difficult (or even impossible) to establish a "nice" characterization of computable categoricity for a given class of computable

structures. In fact, it was proven in [13] that the index set complexity of the class of computably categorical structures is Π^1_1-complete in the *analytical hierarchy*, in which quantifiers can range over sets and functions, instead of just natural numbers. This sharp bound demonstrates that computable categoricity does not have a simple syntactic characterization in general.

One way to address this problem is through *relative computable structure theory*, developed by Ash, Knight, Manasse, and Slaman [2], and independently by Chisholm [8]. First, we must define the *Turing degree of a structure*.

Definition 1.2.4. Let $\mathcal{A}$ be a structure with computable domain $A \subseteq \omega$. The *Turing degree of* $\mathcal{A}$, denoted by $deg(\mathcal{A})$, is the degree of the atomic diagram of $\mathcal{A}$.

While computable structure theory is primarily concerned with properties of *computable* structures, relative computable structure theory investigates structures of *any* Turing degree. In particular, we have the following relativized notion of computable categoricity.

Definition 1.2.5. Let $\mathcal{A}$ be a computable structure. Then $\mathcal{A}$ is *relatively computably categorical* if, for every (possibly non-computable) structure $\mathcal{B}$ isomorphic to $\mathcal{A}$, there exists an isomorphism $h : \mathcal{A} \to \mathcal{B}$ that is computable in the atomic diagram of $\mathcal{B}$.

Equivalently, $\mathcal{A}$ is relatively computably categorical if, for every two (possibly non-computable) copies $\mathcal{A}_1$ and $\mathcal{A}_2$ of $\mathcal{A}$, there exists an isomorphism from $\mathcal{A}_1$ to $\mathcal{A}_2$ that is computable in the join of their degrees.

Every structure that is relatively computably categorical is also computably categorical. To see this, let $\mathcal{A}$ be a computable structure that is relatively computably

categorical, and let $\mathcal{B}$ be a computable copy of $\mathcal{A}$. By Definition 1.2.5, there is an isomorphism h from $\mathcal{A}$ to $\mathcal{B}$ that is computable in $deg(\mathcal{B})$, which is simply $\mathbf{0}$. Thus, h is a computable isomorphism, ensuring that $\mathcal{A}$ is computably categorical.

Downey, Kach, Lempp, and Turetsky [14] established that the index set complexity of the relatively computably categorical structures is Σ^0_3-complete, a much simpler bound than that of computably categorical structures. This is mostly due to the fact that relative computably categoricity has a nice syntactic characterization. Before we can state it, we need the following definition.

Definition 1.2.6. Let $\mathcal{A}$ be a countable structure with domain A. A *formally c.e. Scott family for* $\mathcal{A}$ is a computably enumerable collection Φ of existential formulas over some fixed finite tuple $\vec{c}$ of constants from A such that:

(a) every tuple of elements $\vec{a}$ from A satisfies some formula $\phi(\vec{x}, \vec{c})$ in Φ, and

(b) if two tuples $\vec{a}$ and $\vec{b}$ from A satisfy the same formula $\phi(\vec{x}, \vec{c})$ in Φ, then there is an automorphism $h : \mathcal{A} \to \mathcal{A}$ such that $h(\vec{a}) = \vec{b}$. (In this case, we say that $\vec{a}$ and $\vec{b}$ are *automorphic*.)

Essentially, a Scott family for $\mathcal{A}$ captures the automorphism orbits of $\mathcal{A}$. Note that for a Scott family Φ, a tuple $\vec{a}$ may satisfy more than one formula in Φ, and that not every formula in Φ needs to be satisfied by some tuple from A.

Goncharov [20] was the first to connect relative computable categoricity to the syntactic notion in Definition 1.2.6 by proving the following elegant theorem.

Theorem 1.2.7 ([20])**.** *A computable structure $\mathcal{A}$ is relatively computably categorical if and only if $\mathcal{A}$ has a formally c.e. Scott family.*

Most "natural" classes of computably categorical structures are also relatively computably categorical. However, using an elaborate construction, Goncharov proved in [22] that there is a computably categorical structure that is not relatively computably categorical. This result demonstrates that relative computable categoricity is a strictly stronger notion than that of computable categoricity. (We shall return to this topic in Section 3.2).

Later, Ash, Knight, Manasse, and Slaman [2] and independently Chisholm [8] established a generalized version of Theorem 1.2.7 for all levels of the *hyperarithmetical hierarchy*. See the monograph by Ash and Knight [1] for the requisite background and other important ideas in computable and relative computable model theory.

1.3 Computable Graphs

The remainder of the introduction will be devoted to a discussion on graphs. A *graph* $\mathcal{G} = (V, E)$ consists of a set of vertices V together with a binary edge relation E. If $\mathcal{G}$ is undirected, then E is symmetric. If $\mathcal{G}$ is a simple graph (without loops), then E is irreflexive. Since E is a set of ordered pairs without repeats, our definition of a graph does not allow for duplicated edges, although it is possible to define such a structure. Naturally, a *computable graph* is a graph whose vertex set and edge relation are computable. For computable graphs, we generally assume that V is a countable set, in keeping with our assumptions about structures from Section 1.2.

For finite graphs, one of the most critical areas of study is the computational complexity of the *graph isomorphism problem* (GI), which is the problem of deter-

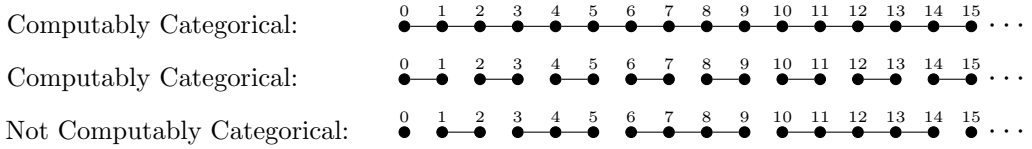
mining if two given finite graphs are isomorphic. For many types of finite graphs, GI is solvable in polynomial time, but the question of whether this is true for all finite graphs remains open. However, given any two finite graphs $\mathcal{G}$ and $\mathcal{H}$, we can always computably determine if they are isomorphic, regardless of the speed of the computation. Furthermore, if $\mathcal{G}$ and $\mathcal{H}$ are indeed isomorphic, we can always compute an isomorphism between them.

The same is not true of infinite graphs. Given two countably infinite computable graphs $\mathcal{G}$ and $\mathcal{H}$, there may be no effective procedure to decide if they are isomorphic to each other. Moreover, even if we are given the information that $\mathcal{G}$ and $\mathcal{H}$ are isomorphic, we may still be unable to compute an isomorphism from $\mathcal{G}$ to $\mathcal{H}$. In fact, in [25], Hirschfeldt, Khoussainov, Shore, and Slinko showed that directed graphs are "universal" in the sense that they can effectively encode certain algorithmic properties from any countable structure. More informally, this result demonstrated that countable graphs, and the isomorphisms between them, can be arbitrarily complex. Thus, an important goal is to determine the computability-theoretic complexity of isomorphisms for various classes of computable graphs. In particular, we would like to identify the types of computable graphs for which computable isomorphisms must exist.

To that end, computable categoricity of graphs has been intensively studied in recent years. Lempp, McCoy, Miller, and Solomon [31] completely characterized computable trees of finite height that are computably categorical, and Miller [36] showed that no computable tree of infinite height is computably categorical. In [11], Csima, Khoussainov, and Liu partially characterized computable categoricity

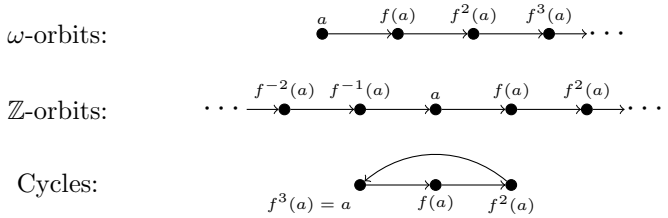
for *strongly locally finite graphs* (graphs which have countably many finite connected components), by investigating additional functions and relations on the structures and proper embeddability of the components (see also [32]). As an example, it can be shown that the first two graphs in Figure 1.3 are computably categorical, whereas the third graph is not.

Figure 1.3: Computable graphs that are and are not computably categorical.



Then Cenzer, Harizanov, and Remmel looked at different types of infinite directed graphs whose edge relations are derived from computable functions. They first studied directed graphs of this kind in [6], where they defined *injection structures*. An **injection structure** $\mathcal{A} = (A, f)$ is a countable set A (usually ω) together with an injective function $f : A \to A$. Injection structures can be completely classified up to isomorphism by the number, type, and size of its *orbits*, which are informally defined as the types of connected components the structure may have.

Figure 1.4: The three types of orbits in an injection structure.

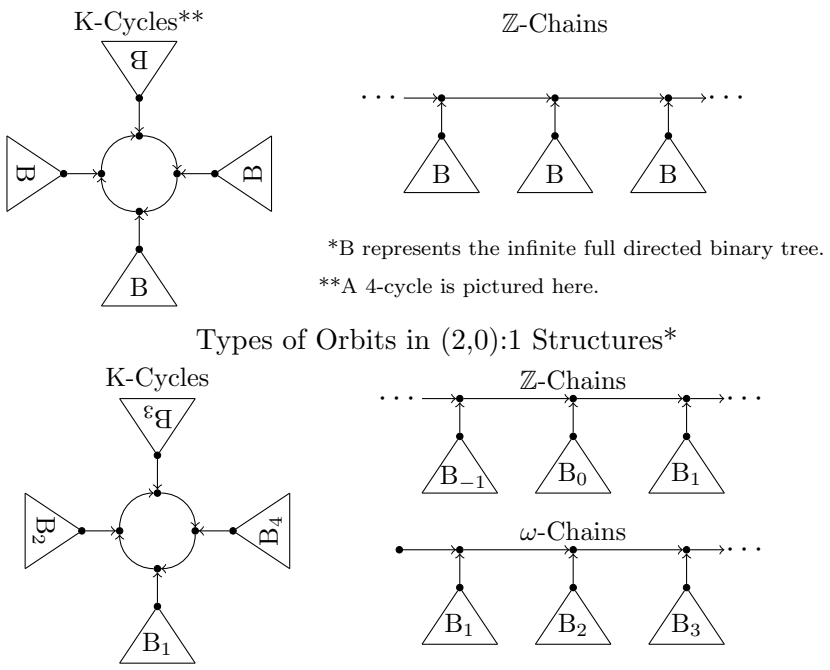


Cenzer, Harizanov, and Remmel proved the following characterization theorem for injection structures.

Theorem 1.3.1 ([6]). *A computable injection structure $\mathcal{A}$ is computably categorical if and only if $\mathcal{A}$ has only finitely many infinite orbits, that is, only finitely many ω-orbits and only finitely many $\mathbb{Z}$-orbits.*

Next, Cenzer, Harizanov, and Remmel [7] looked at **two-to-one (2:1) structures** $\mathcal{A} = (A, f)$, where $|f^{-1}(a)| = 2$ for all $a \in A$, as well as **(2,0):1 structures** $\mathcal{A} = (A, f)$, where $|f^{-1}(a)| \in \{0, 2\}$ for all $a \in A$. Thus, in a 2:1 structure, every element has exactly two pre-images under f, and in a (2,0):1 structure, every element has either exactly two pre-images or no pre-images under f. The types of orbits for these structures are shown below.

Figure 1.5: Types of orbits in two-to-one structures and (2,0):1 structures.



Cenzer, Harizanov, and Remmel partially characterized computably categorical (2,0):1 structures by considering additional structural and algorithmic properties.

However, they fully characterized the computably categorical two-to-one structures by proving the following theorem.

Theorem 1.3.2 ([7])**.** *A computable two-to-one structure $\mathcal{A}$ is computably categorical if and only if $\mathcal{A}$ has only finitely many $\mathbb{Z}$-chains.*

The goal of this dissertation is to explore computable categoricity and other related topics for a similar class of infinite directed graphs called (2,1):1 *structures*. In Chapter 2, we give the defining properties of (2,1):1 structures, as well as address the associated isomorphism problem and higher levels of categoricity. We also discuss two additional functions on (2,1):1 structures, and how they affect the complexity of computable isomorphisms. In Chapter 3, we give necessary and sufficient conditions for a (2,1):1 structure to be computably categorical. Additionally, we investigate the related concept of relative computable categoricity. Lastly, we present a connection between computable categoricity of (2,1):1 structures and an open problem in number theory.

Chapter 2

(2,1):1 Structures

2.1 Basic Notions

Definition 2.1.1. A (2,1):1 *structure* (read as "two-or-one-to-one structure") $\mathcal{A} = (A, f)$ is a countable set A (usually the natural numbers) together with a function $f : A \to A$ such that $|f^{-1}(a)| \in \{1, 2\}$ for all $a \in A$. That is, every element in A has either exactly two pre-images or exactly one pre-image under f.

Naturally, we say that a (2,1):1 structure $\mathcal{A} = (A, f)$ is *computable* if A is a computable set and f is a uniformly computable function.

Before we begin analyzing the computability-theoretic properties of these structures, we must first introduce and discuss some of their basic structural properties. We start with the *orbit of an element*.

Definition 2.1.2. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, and let $x \in A$. The *orbit of x in $\mathcal{A}$,* denoted by $\mathcal{O}_{\mathcal{A}}(x)$, is defined as follows:

$$\mathcal{O}_{\mathcal{A}}(x) = \{y \in A | (\exists m, n)(f^m(x) = f^n(y))\}$$

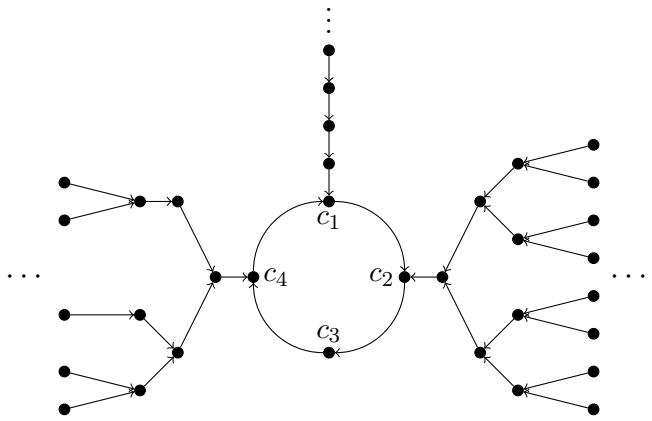
Here, $f^m(x)$ denotes the result of iterating the function m times on x. If we think of (2,1):1 structures as directed graphs, we can think of orbits as the connected components of the graph.

It is not hard to see that a (2,1):1 structure can only consist of two general types of orbits. We refer to them as **K-cycles** and **$\mathbb{Z}$-chains**, following the naming conventions for the orbits of 2:1 structures used by Cenzer, Harizanov, and Remmel in [7]. We describe these orbits below.

K-Cycles

A **K-cycle** is a directed cycle with k elements, where every element in the cycle has a directed binary tree attached, each of which is either infinite or empty.

Figure 2.1: An example of a 4-cycle.



An element x of a K-cycle is called a **cyclic element** if there exists an $n > 0$ such that $f^n(x) = x$. We denote the cyclic elements of a K-cycle as c_1, c_2,...,c_K, where $c_i \neq c_j$ for $1 \leq i < j \leq K$, $f(c_r) = c_{r+1}$ for $1 \leq r < K$, and $f(c_K) = c_1$. Since each K-cycle consists of only one directed cycle, we can uniquely specify a particular

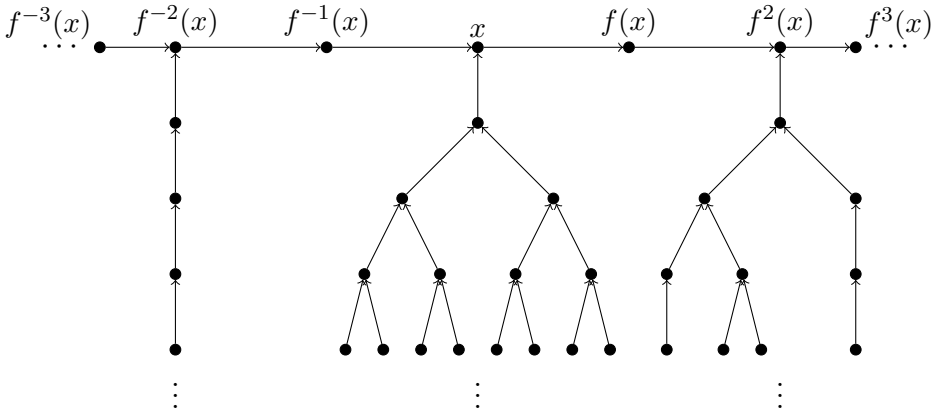
K-cycle within a (2,1):1 structure by listing its K cyclic elements.

In Figure 2.1, each of the cyclic elements c_1, c_2, c_3, and c_4 has a different type of binary tree attached. The tree attached to c_1 is often referred to as a **degenerate tree**, where every element in the tree has exactly one pre-image; the tree attached to c_2 is called a **full binary tree**, where every element has exactly two pre-images; the tree attached to c_3 is the **empty tree**; and the tree attached to c_4 is an arbitrary infinite binary tree that is neither empty, degenerate, nor full.

$\mathbb{Z}$-Chains

A **$\mathbb{Z}$-chain** consists of a $\mathbb{Z}$-orbit of elements, where every element in the orbit has a directed binary tree attached, each of which is either infinite or empty.

Figure 2.2: An example of a $\mathbb{Z}$-chain.



Here, a $\mathbb{Z}$-orbit refers to an infinite set of elements $\{..., f^{-1}(x), x, f(x), ...\}$ such that for all $m, n \in \mathbb{Z}$ with $m \neq n$, $f^m(x) \neq f^n(x)$. Unlike with cyclic elements in a K-cycle, a $\mathbb{Z}$-orbit within a $\mathbb{Z}$-chain does not necessarily *uniquely* determine the $\mathbb{Z}$-chain, since a $\mathbb{Z}$-chain may contain more than one different $\mathbb{Z}$-orbit. Indeed, if a $\mathbb{Z}$-chain contains any element with two pre-images, then that $\mathbb{Z}$-chain will contain more than one distinct $\mathbb{Z}$-orbit. However, given a $\mathbb{Z}$-chain, we can establish a **canonical**

$\mathbb{Z}$-orbit $\{..., f^{-2}(x), f^{-1}(x), x, f(x), f^2(x), ...\}$, where x is the least element in the $\mathbb{Z}$-chain (under the usual ordering on $\mathbb{N}$), and $f^{-(n+1)}(x)$ is the least pre-image of $f^{-n}(x)$ for all $n \geq 0$. Thus, in Figure 2.2, if we take the labeled elements to be the canonical $\mathbb{Z}$-orbit of the $\mathbb{Z}$-chain, then $f^{-1}(x)$ is the least pre-image of x, $f^{-3}(x)$ is the least pre-image of $f^{-2}(x)$, and so on.

As we can see from Figures 2.1 and 2.2, the orbits of a (2,1):1 structure are essentially directed graphs. However, this is not quite correct, as the orbit of an element x is only defined to be the *set* of elements in the same connected component as x, and does not include any additional structure specifying an edge relation. It will be advantageous later on to be able to refer to a connected component in a (2,1):1 structure as a directed graph instead of just a set of vertices. So we formalize this notion in the following definition.

Definition 2.1.3. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, and let $x \in A$. The *connected component of x in $\mathcal{A}$,* denoted by $C_{\mathcal{A}}(x)$, is the directed graph associated with $\mathcal{O}_{\mathcal{A}}(x)$. That is,

$$C_{\mathcal{A}}(x) = (V, E)$$

where $V = \mathcal{O}_{\mathcal{A}}(x)$ and $E = \{(x, f(x)) : x \in V\}$.

Since a (2,1):1 structure can have at most countably many connected components, we shall often enumerate the connected components of a structure as $C_0, C_1, C_2, ...$

Lemma 2.1.4. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure, and let $x \in A$. Then the following are all $\emptyset'$-computable:*

(1) $\{x : \mathcal{O}_{\mathcal{A}}(x)$ *is a K-cycle*$\}$

(2) $\{x : \mathcal{O}_{\mathcal{A}}(x)$ *is a* $\mathbb{Z}$*-chain*$\}$

(3) $\{(x, y) : \mathcal{O}_{\mathcal{A}}(x) = \mathcal{O}_{\mathcal{A}}(y)\}$

(4) $C_{\mathcal{A}}(x)$.

Proof. To see that (1) is $\emptyset'$-computable, observe that $\mathcal{O}_{\mathcal{A}}(x)$ is a K-cycle if and only if the following statement holds:

$$(\exists c, n)[f^n(x) = c \wedge f^K(c) = c \wedge (\forall m < K)(m \neq 0 \implies f^m(c) \neq c)] \qquad (*)$$

The relation inside of the bracket is computable since f is computable and the universal quantifier is bounded. So $(*)$ is a Σ^0_1-formula, and thus (1) is a Σ^0_1-set. Therefore, (1) is $\emptyset'$-computable.

The set (2) is the complement of (1) in A. Thus, (2) is also $\emptyset'$-computable.

By Definition 2.1.2, $\mathcal{O}_{\mathcal{A}}(x) = \mathcal{O}_{\mathcal{A}}(y)$ if and only if the following statement holds:

$$(\exists m, n)(f^m(x) = f^n(y)) \qquad (**)$$

Since $(**)$ is also a Σ^0_1-formula, (3) is a Σ^0_1-set, and thus it is also computable in $\emptyset'$.

Finally, (4) is a directed graph, whose set of vertices V is $\{y : \mathcal{O}_{\mathcal{A}}(x) = \mathcal{O}_{\mathcal{A}}(y)\}$ and whose edges E are $\{(a, b) : a \in V \wedge f(a) = b\}$. Since (3) is computable in $\emptyset'$, the vertex set V is also computable in $\emptyset'$. E is also $\emptyset'$-computable, since V is $\emptyset'$-computable and f is a computable function. Therefore, (4) is an $\emptyset'$-computable graph. $\square$

To further analyze our structures as graphs, we explore another fundamental property that we will refer to often: the *tree of an element*.

Definition 2.1.5. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, and let $x \in A$. The *tree of x in $\mathcal{A}$,* denoted by $tree_{\mathcal{A}}(x)$, is defined as:

$$tree_{\mathcal{A}}(x) = \{a \in A : (\exists n)(f^n(a) = x)\}$$

Furthermore, the *Tree of x in $\mathcal{A}$,* denoted by $Tree_{\mathcal{A}}(x)$, is the directed graph associated with $tree_{\mathcal{A}}(x)$. That is,

$$Tree_{\mathcal{A}}(x) = (V, E)$$

where $V = tree_{\mathcal{A}}(x)$ and $E = \{(x, f(x)) : x \in V \wedge f(x) \in V\}$.

Intuitively, we can think of $tree_{\mathcal{A}}(x)$ as the set of all *predecessors* of x (or the set of all elements that will eventually *lead* to x), and we can think of $Tree_{\mathcal{A}}(x)$ as a rooted binary tree with x as its root. It is apparent that if c_i is a cyclic element, then $tree_{\mathcal{A}}(c_i) = \mathcal{O}_{\mathcal{A}}(c_i)$, which is the entire K-cycle containing c_i. However, we often wish to refer to those elements in a K-cycle that are connected to a cyclic element via a directed path that does not contain other cyclic elements. So we introduce the notion of an *exclusive tree*.

Definition 2.1.6. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, and let c_i be a cyclic element on a K-cycle in A. The *exclusive tree of c_i in $\mathcal{A}$,* denoted by $extree_{\mathcal{A}}(c_i)$, is the following set:

$$extree_{\mathcal{A}}(c_i) = \{a \in A : (\exists n)[f^n(a) = c_i \wedge (\forall m < n)(f^{m+K}(a) \neq f^m(a))]\}$$

The *exclusive Tree of c_i in $\mathcal{A}$,* denoted by $exTree_{\mathcal{A}}(c_i)$ is the directed graph associated with $extree_{\mathcal{A}}(c_i)$.

Thus, in Figure 2.1, the exclusive Trees of c_1, c_2, and c_3 are degenerate, full, and empty, respectively.

All of the structural properties of (2,1):1 structures presented so far are generally infinite sets, or infinite substructures, which can be very complex. To begin to examine the computability-theoretic properties of such structures, it is easier to look at some important types of *finite* components of (2,1);1 structures. We list them below.

Definition 2.1.7. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, let $x \in A$, let $A_0 \subseteq A$ be the elements of a K-cycle in $\mathcal{A}$ with cyclic elements $c_1, c_2, ..., c_i, ..., c_K$, and let $n \in \mathbb{N}$.

(a) The n^{th} *level of the tree of* x, denoted by $tree_{\mathcal{A}}(x|n)$, is defined as:

$$tree_{\mathcal{A}}(x|n) = \{a \in A : f^n(a) = x\}.$$

Similarly, the n^{th} *level of the exclusive tree of* c_i is defined as:

$$extree_{\mathcal{A}}(c_i|n) = \{a \in A : a \in extree_{\mathcal{A}}(c_i) \wedge f^n(a) = c_i\}.$$

(b) The *tree of* x, *truncated at level* n, denoted by $tree_{\mathcal{A}}(x, n)$, is defined as:

$$tree_{\mathcal{A}}(x, n) = \{a \in A : (\exists m \leq n)(f^m(a) = x)\},$$

and $Tree_{\mathcal{A}}(x, n)$ is its associated directed graph.

Similarly, the *exclusive tree of* c_i, *truncated at level* n is defined as:

$$extree_{\mathcal{A}}(c_i, n) = \{a \in A : a \in extree_{\mathcal{A}}(c_i) \wedge (\exists m \leq n)(f^m(a) = x)\},$$

and $exTree_{\mathcal{A}}(c_i, n)$ is its associated directed graph.

(c) The n^{th} *level of the K-cycle* A_0, denoted by $(A_0|n)$, is defined as:

$$(A_0|n) = \{a \in A : (\exists i \leq K)(a \in extree_{\mathcal{A}}(c_i|n)\}.$$

The *K-cycle* A_0, *truncated at level* n, denoted by (A_0, n), is defined as:

$$(A_0, n) = \{a \in A : (\exists i \leq K)(a \in extree_{\mathcal{A}}(c_i, n)\},$$

and (C_0, n) is its associated directed graph.

We conclude this section with some basic computability facts, which will be useful in the upcoming sections.

Lemma 2.1.8. *Let $\mathcal{A} = (A, f)$ be a computable (2,1):1 structure, let $x \in A$, let $A_0 \subseteq A$ be the elements of a K-cycle in $\mathcal{A}$ with cyclic elements $c_1, c_2, ..., c_i, ..., c_K$, and let $n \in \mathbb{N}$. Then:*

(a) *$tree_{\mathcal{A}}(x|n)$, $extree_{\mathcal{A}}(c_i|n)$, and $(A_0|n)$ are computable.*

(b) *$tree_{\mathcal{A}}(x, n)$, $extree_{\mathcal{A}}(c_i, n)$, and (A_0, n) are computable. Also, $Tree_{\mathcal{A}}(x, n)$, $exTree_{\mathcal{A}}(c_i, n)$, and (C_0, n) are computable as well.*

(c) *$tree_{\mathcal{A}}(x)$ and $extree_{\mathcal{A}}(c_i)$ are $\mathbf{0}'$-computable. Also, $Tree_{\mathcal{A}}(x)$ and $exTree_{\mathcal{A}}(c_i)$ are $\mathbf{0}'$-computable.*

Proof. Let $a, b \in A$ be given.

(a) Since $a \in tree_{\mathcal{A}}(x|n)$ if and only if $f^n(a) = x$, we can decide membership of $tree_{\mathcal{A}}(x|n)$ by simply calculating $f^n(a)$, which is an effective procedure since f is computable. Next, $a \in extree_{\mathcal{A}}(c_i|n)$ if and only if the following statement holds:

$$(\forall m < n)(\forall j \leq K)[a \in tree_{\mathcal{A}}(c_i|n) \wedge f^m(a) \neq c_j] \tag{1}$$

Since the predicate in brackets is computable (as previously established) and the universal quantifiers are bounded, (1) is computable, and thus so is $extree_{\mathcal{A}}(c_i|n)$.

Finally, $a \in (A_0|n)$ if and only if:

$$(\exists j \leq K)[a \in extree_{\mathcal{A}}(c_j|n)] \tag{2}$$

Since $extree_{\mathcal{A}}(c_j|n)$ is computable, and the existential quantifier is bounded, (2) is computable, and thus so is $(A_0|n)$.

(b) Observe that $a \in tree_{\mathcal{A}}(x, n)$ if and only if:

$$(\exists m \leq n)[a \in tree_{\mathcal{A}}(x|m)] \tag{3}$$

By part (a), (3) is computable, and hence $tree_{\mathcal{A}}(x, n)$ is computable. Similarly, $a \in extree_{\mathcal{A}}(c_i, n)$ if and only if:

$$(\exists m \leq n)[a \in extree_{\mathcal{A}}(c_i|m)] \tag{4}$$

Again, by part (a), (4) is computable, and hence so is $extree_{\mathcal{A}}(c_i, n)$. Next, $a \in (A_0, n)$ if and only if:

$$(\exists m \leq n)[a \in (A_0|m)] \tag{5}$$

By part (a), (5) is computable. Hence, (A_0, n) is computable.

To show that $Tree_{\mathcal{A}}(x, n)$ is computable, it suffices to show that the edge set E of $Tree_{\mathcal{A}}(x, n)$ is computable, since the vertex set V is simply $tree_{\mathcal{A}}(x, n)$, which

has already been shown to be computable. Given an ordered pair (a, b), we can decide if $(a, b) \in E$ by simply checking if $a \in V$, and $b \in V$, and $f(a) = b$, all of which are computable. A similar argument shows that $exTree_{\mathcal{A}}(c_i, n)$ and (C_0, n) are computable as well.

(c) Recall that $a \in tree_{\mathcal{A}}(x)$ if and only if:

$$(\exists n)[f^n(a) = x] \tag{6}$$

Since (6) is a Σ^0_1-formula (the predicate in brackets is computable), $tree_{\mathcal{A}}(x)$ is a Σ^0_1-set, and is therefore computable in $\mathbf{0}'$. Similarly, $a \in extree_{\mathcal{A}}(c_i)$ if and only if:

$$(\exists n)[a \in extree_{\mathcal{A}}(c_i|n)] \tag{7}$$

Again, (7) is a Σ^0_1-formula, and so $extree_{\mathcal{A}}(c_i)$ is also a Σ^0_1-set, and is therefore computable in $\mathbf{0}'$. A similar argument as the one used in part (b) shows that $Tree_{\mathcal{A}}(x)$ and $exTree_{\mathcal{A}}(c_i)$ are $\mathbf{0}'$-computable graphs.

□

2.2 The Isomorphism Problem for (2,1):1 Structures

Given a class of structures $\mathcal{C}$, the **isomorphism problem for $\mathcal{C}$** is the problem of deciding membership of the following set:

$$\{(i,j) : \varphi_i \text{ and } \varphi_j \text{ are the characteristic functions of isomorphic structures in } \mathcal{C}\},$$

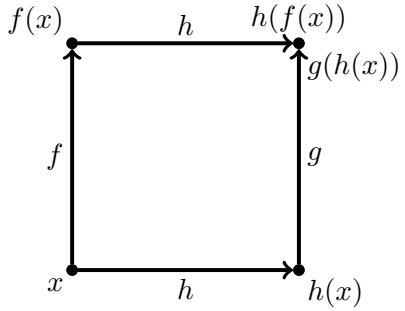
where the characteristic function of a structure is simply the characteristic function of its atomic diagram.

More informally, the isomorphism problem asks whether or not two given structures within a class are isomorphic to each other. We generally wish to determine the difficulty of the isomorphism problem for a class of structures. That is, for a given class $\mathcal{C}$, we want to establish an upper bound within the Arithmetical Hierarchy for the complexity of the isomorphism problem for $\mathcal{C}$. This has been done for a variety of structures, including certain types of graphs and trees. For example, Steiner proved in [45] that the isomorphism problem for computable finite-branching trees under a predecessor function is Π^0_2. In this section, we establish a corresponding result for directed graphs associated with (2,1):1 structures. First, we start with a natural definition.

Definition 2.2.1. Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two (2,1):1 structures. Then a function $h : A \to B$ is an *isomorphism* from $\mathcal{A}$ to $\mathcal{B}$ if h is a bijection and for all $x \in A$, $h(f(x)) = g(h(x))$. That is, h is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$ if h is a bijection that preserves the function.

We can also visualize Definition 2.2.1 by stating that h is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$ if h is a bijection and, for all $x \in A$, the following diagram commutes:

Figure 2.3: Commutative diagram for isomorphic (2,1):1 structures.



Naturally, we say that $\mathcal{A}$ and $\mathcal{B}$ are *isomorphic* if there exists an isomorphism h from $\mathcal{A}$ to $\mathcal{B}$, and we write $\mathcal{A} \cong \mathcal{B}$. Since we can now think of our structures as directed graphs, we can say that two (2,1):1 structures are isomorphic if their associated directed graphs are isomorphic. We shall often appeal to this intuition from now on, rather than referring to the more technical statement given in Definition 2.2.1.

We now establish the main result of this section, that the isomorphism problem for computable (2,1):1 structures is Π^0_4. First, we need the following lemmas.

Lemma 2.2.2. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with non-cyclic elements $a_0 \in A$ and $b_0 \in B$, and let $n > 0$ be given. Then the isomorphism problem for $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ is decidable via a Δ^0_2-oracle.*

Proof. We first show by induction that for any $n > 0$, we can reveal all vertices and edges of $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ via a Δ^0_2-oracle.

Given a Δ^0_2-oracle, we can use it to compute all vertices of $Tree_{\mathcal{A}}(a_0, 1)$ by deter-

mining if the following Σ_1^0 statement is true:

$$(\exists a_1, a_2)[f(a_1) = f(a_2) = a_0 \wedge a_1 \neq a_2] \qquad (*)$$

If $(*)$ holds, then we search until we find such elements a_1 and a_2. Otherwise, we end our search after finding one such element a_1 with $f(a_1) = a_0$. In either case, we can effectively reveal $Tree_{\mathcal{A}}(a_0, 1)$. In a similar manner, we can reveal $Tree_{\mathcal{B}}(b_0, 1)$. Now assume inductively that we can reveal $Tree_{\mathcal{A}}(a_0, m)$ and $Tree_{\mathcal{B}}(b_0, m)$ via a Δ_2^0-oracle, where $m \geq 1$. From Lemma 2.1.8(a), we know that $tree_{\mathcal{A}}(a_0|m)$ is computable. So for each element $x \in tree_{\mathcal{A}}(a_0|m)$ (of which there are only finitely many), we use our Δ_2^0-oracle to determine if the following statement holds:

$$(\exists x_1, x_2)[f(x_1) = f(x_2) = x \wedge x_1 \neq x_2] \qquad (**)$$

If $(**)$ holds for a particular x, we search for and find both pre-images of x; otherwise, we stop searching after finding the first pre-image of x. So with our oracle, we can effectively find the pre-images of all elements in the m^{th} level of the tree of a_0, giving us all elements in the $(m+1)^{th}$ level. And since we can reveal $Tree_{\mathcal{A}}(a_0, m)$ with a Δ_2^0-oracle by the inductive hypothesis, we have that $Tree_{\mathcal{A}}(a_0, m+1)$ can also be revealed using a Δ_2^0-oracle. By the same argument, we have the same result for $Tree_{\mathcal{B}}(b_0, m+1)$, thus completing the induction.

Now recall that $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ are *finite* directed graphs. So once we know all of the vertices and edges of both graphs (which is Δ_2^0-computable from

our induction argument), we only need to check finitely many possible isomorphisms from $Tree_{\mathcal{A}}(a_0, n)$ to $Tree_{\mathcal{B}}(b_0, n)$ to decide if the two trees are isomorphic. Therefore, the isomorphism problem for $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ is decidable via a Δ^0_2-oracle. □

Lemma 2.2.3. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with non-cyclic elements $a_0 \in A$ and $b_0 \in B$. Then the isomorphism problem for $Tree_{\mathcal{A}}(a_0)$ and $Tree_{\mathcal{B}}(b_0)$ is Π^0_2.*

Proof. Note that $Tree_{\mathcal{A}}(a_0)$ and $Tree_{\mathcal{B}}(b_0)$ are isomorphic if and only if the following statement holds:

$$(\forall n)[Tree_{\mathcal{A}}(a_0, n) \cong Tree_{\mathcal{B}}(b_0, n)] \qquad (*)$$

By Lemma 2.2.2, the statement in brackets is Δ^0_2, and thus it is Π^0_2. So the bracketed statement can be written as a $\forall\exists$-formula, which implies that $(*)$ can be written as a $\forall\forall\exists$-sentence, which in turn can be written as a $\forall\exists$-sentence. Therefore, $(*)$ is a Π^0_2 sentence, and the isomorphism problem for $Tree_{\mathcal{A}}(a_0)$ and $Tree_{\mathcal{B}}(b_0)$ is Π^0_2. □

Lemma 2.2.4. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with cyclic elements $a_0 \in A$ and $b_0 \in B$. Then the isomorphism problem for $exTree_{\mathcal{A}}(a_0)$ and $exTree_{\mathcal{B}}(b_0)$ is also Π^0_2.*

Proof. Observe that $exTree_{\mathcal{A}}(a_0) \cong exTree_{\mathcal{B}}(b_0)$ if and only if the following statement holds:

$$(\forall n)[exTree_{\mathcal{A}}(a_0, n) \cong exTree_{\mathcal{B}}(b_0, n)] \qquad (*)$$

So it suffices to show that the statement in brackets in Δ_2^0, as we can use the same argument from the proof of Lemma 2.2.3 to show that $(*)$ is Π_2^0.

Let n be given. To decide if $exTree_{\mathcal{A}}(a_0, n)$ is isomorphic to $exTree_{\mathcal{B}}(b_0, n)$, we first check if a_0 and b_0 have two predecessors. This can be computed using a Δ_2^0-oracle, as was established in the proof of Lemma 2.2.2. If neither a_0 nor b_0 has two predecessors, then $exTree_{\mathcal{A}}(a_0, n) \cong exTree_{\mathcal{B}}(b_0, n)$, in a trivial way. If a_0 and b_0 have different numbers of predecessors, then $exTree_{\mathcal{A}}(a_0, n) \ncong exTree_{\mathcal{B}}(b_0, n)$.

If both a_0 and b_0 have two predecessors, let a_1 and b_1 be the respective predecessors of a_0 and b_0 that are not cyclic elements. We can computably determine a_1 and b_1 by first searching for both predecessors of a_0 and b_0, then iterating f and g on a_0 and b_0, respectively, until we find the cyclic predecessors of both a_0 and b_0; then the remaining predecessors of a_0 and b_0 must be the non-cyclic elements a_1 and b_1.

Then, for $n \geq 1$, $exTree_{\mathcal{A}}(a_0, n) \cong exTree_{\mathcal{B}}(b_0, n)$ if and only if $Tree_{\mathcal{A}}(a_1, n-1) \cong Tree_{\mathcal{B}}(b_1, n-1)$ (for $n = 0$, $exTree_{\mathcal{A}}(a_0, n)$ and $exTree_{\mathcal{B}}(b_0, n)$ are trivially isomorphic). By Lemma 2.2.2, the statement $Tree_{\mathcal{A}}(a_1, n-1) \cong Tree_{\mathcal{B}}(b_1, n-1)$ is Δ_2^0. Hence, the bracketed statement in $(*)$ is also Δ_2^0, establishing that $(*)$ is Π_2^0. Therefore, the isomorphism problem for $exTree_{\mathcal{A}}(a_0)$ and $exTree_{\mathcal{B}}(b_0)$ is also Π_2^0 $\square$

Lemma 2.2.5. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures. Suppose that $A_0 \subseteq A$ and $B_0 \subseteq B$ consist of all of the elements in a K-cycle of A and an L-cycle of B respectively, with cyclic elements $c_1, ..., c_K \in A_0$ and $d_1, ...d_L \in B_0$. Then the statement "$Tree_{\mathcal{A}}(c_1) \cong Tree_{\mathcal{B}}(d_1)$" is Π_2^0. That is, the isomorphism problem for K-cycles is Π_2^0.*

Proof. Observe that $Tree_{\mathcal{A}}(c_1) \cong Tree_{\mathcal{B}}(d_1)$ if and only if $|\{c_1, ..., c_K\}| = |\{d_1, ..., d_L\}|$ and the following sentence is true:

$$(\exists j \leq L)[exTree_{\mathcal{A}}(c_1) \cong exTree_{\mathcal{B}}(d_j) \wedge (\forall n \leq K)(exTree_{\mathcal{A}}(f^n(c_1)) \cong exTree_{\mathcal{B}}(g^n(d_j)))]$$

The existential quantifier is bounded, and thus does not add any complexity to the bracketed statement. The first conjunct in brackets is Π_2^0 by Lemma 2.2.4. The universal quantifier in front of the second conjunct is also bounded, and thus does not change the complexity of the statement inside the quantifier, which is also Π_2^0 by Lemma 2.2.4. Hence, the whole sentence above is Π_2^0. Also, the statement "$|\{c_1, ..., c_K\}| = |\{d_1, ..., d_L\}|$" is easily seen to be computable, as we can simply iterate f on c_1 and g on d_1 and count the number of cyclic elements in both cycles until we arrive at c_1 and d_1 again. Therefore, the isomorphism problem for K-cycles is Π_2^0. □

Lemma 2.2.6. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with elements $a_0 \in A$ and $b_0 \in B$ such that both a_0 and b_0 are in $\mathbb{Z}$-chains. Then the statement "$C_{\mathcal{A}}(a_0) \cong C_{\mathcal{B}}(b_0)$" is Σ_3^0. That is, the isomorphism problem for $\mathbb{Z}$-chains is Σ_3^0.*

Proof. Observe that the $\mathbb{Z}$-chain containing a_0 and the $\mathbb{Z}$-chain containing b_0 are isomorphic if and only if the following is true:

$$(\exists p, q)(\forall t)[Tree_{\mathcal{A}}(f^{p+t}(a_0)) \cong Tree_{\mathcal{B}}(g^{q+t}(b_0))] \qquad (*)$$

By Lemma 2.2.3, the statement in brackets is Π^0_2. So, $(*)$ can be written as a $\exists\forall\forall\exists$-sentence, and thus it can be written as a $\exists\forall\exists$-sentence. Therefore, $(*)$ is a Σ^0_3-sentence, and the isomorphism problem for $\mathbb{Z}$-chains is Σ^0_3. □

Lemma 2.2.7. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with elements $a_0 \in A$ and $b_0 \in B$. Then the statement "$C_{\mathcal{A}}(a_0) \cong C_{\mathcal{B}}(b_0)$" is Σ^0_3. That is, determining if two arbitrary connected components are isomorphic is Σ^0_3.*

Proof. To determine if $C_{\mathcal{A}}(a_0)$ is isomorphic to $C_{\mathcal{B}}(b_0)$, we first decide if a_0 and b_0 are in the same type of orbit. By Lemma 2.1.4, deciding the type of orbit of an element is Σ^0_1, and thus certainly Σ^0_3. If a_0 and b_0 are in different types of orbits, then $C_{\mathcal{A}}(a_0)$ and $C_{\mathcal{B}}(b_0)$ are automatically non-isomorphic. If a_0 and b_0 are both in K-cycles, then by Lemma 2.2.5 the isomorphism problem for $C_{\mathcal{A}}(a_0)$ and $C_{\mathcal{B}}(b_0)$ is Π^0_2, and thus Σ^0_3. If a_0 and b_0 are both in $\mathbb{Z}$-chains, then by Lemma 2.2.6 the isomorphism problem for the two connected components is also Σ^0_3. Therefore, the isomorphism problem for two arbitrary connected components is Σ^0_3. □

With these lemmas in place, we can now establish the main theorem for this section.

Theorem 2.2.8. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures, with connected components G_i in $\mathcal{A}$ and H_i in $\mathcal{B}$. Then the statement "$\mathcal{A} \cong \mathcal{B}$" is Π^0_4.*

Proof. The theorem follows from the key observation that $\mathcal{A}$ and $\mathcal{B}$ are isomorphic if and only if the number of connected components in $\mathcal{A}$ with any given isomorphism

type is equal to the number of connected components in $\mathcal{B}$ with that isomorphism type.

Define formulas **P, Q, R,** and **S** in the following manner:

$$\mathbf{P} : [\bigwedge_{1\leq i<j\leq k} (\mathcal{O}_{\mathcal{A}}(a_i) \neq \mathcal{O}_{\mathcal{A}}(a_j)) \wedge \bigwedge_{1\leq i<j\leq k} (C_{\mathcal{A}}(a_i) \cong C_{\mathcal{A}}(a_j) \cong G_n)]$$

$$\mathbf{Q} : [\bigwedge_{1\leq i<j\leq k} (\mathcal{O}_{\mathcal{B}}(b_i) \neq \mathcal{O}_{\mathcal{B}}(b_j)) \wedge \bigwedge_{1\leq i<j\leq k} (C_{\mathcal{B}}(b_i) \cong C_{\mathcal{B}}(b_j) \cong G_n)]$$

$$\mathbf{R} : [\bigwedge_{1\leq i<j\leq k} (\mathcal{O}_{\mathcal{B}}(b_i) \neq \mathcal{O}_{\mathcal{B}}(b_j)) \wedge \bigwedge_{1\leq i<j\leq k} (C_{\mathcal{B}}(b_i) \cong C_{\mathcal{B}}(b_j) \cong H_n)]$$

$$\mathbf{S} : [\bigwedge_{1\leq i<j\leq k} (\mathcal{O}_{\mathcal{A}}(a_i) \neq \mathcal{O}_{\mathcal{A}}(a_j)) \wedge \bigwedge_{1\leq i<j\leq k} (C_{\mathcal{A}}(a_i) \cong C_{\mathcal{A}}(a_j) \cong H_n)]$$

Then $\mathcal{A} \cong \mathcal{B}$ if and only if the following statement holds:

$$(\forall n, k)[((\exists a_1, ..., a_k)(P) \implies (\exists b_1, ..., b_k)(Q)) \wedge ((\exists b_1, ..., b_k)(R) \implies (\exists a_1, ..., a_k)(S))]$$

By Lemmas 2.1.4 and 2.2.7, formulas **P, Q, R,** and **S** are all Σ_3^0. Thus, the existential formulas in brackets above are also Σ_3^0, and so the universal statement is Π_4^0. Therefore, the isomorphism problem for (2,1):1 structures is Π_4^0. $\square$

2.3 Δ^0_n-Categoricity of (2,1):1 Structures

We now turn our attention to establishing the complexity of isomorphisms between two isomorphic (2,1):1 structures. More specifically, given two computable isomorphic (2,1):1 structures, we wish to give an upper bound on the complexity of any isomorphism between the two structures. To that end, we start with a generalized notion of Definition 1.2.3.

Definition 2.3.1. A computable structure $\mathcal{A}$ is Δ^0_n-*categorical* if every two computable copies of $\mathcal{A}$ are isomorphic via a Δ^0_n function; that is, there is an isomorphism between any two computable copies that is computable in $\emptyset^{(n-1)}$.

Center, Harizanov, and Remmel proved in [6] that all computable injection structures are Δ^0_3-categorical, and in [34] Marshall proved the same result for partial injection structures. In [7], Cenzer, Harizanov, and Remmel showed that all computable 2:1 structures are Δ^0_2-categorical, and that all computable, locally finite (2,0):1 structures are Δ^0_3-categorical.

Our main result in this section is that all (2,1):1 structures are Δ^0_4-categorical. In other words, the isomorphism between any two computable isomorphic (2,1):1 structures is computable in $\emptyset'''$. Like the main theorem in the previous section, we prove this through a series of lemmas, building up from truncated trees to any arbitrary (2,1):1 structure.

Lemma 2.3.2. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with non-cyclic elements $a_0 \in A$ and $b_0 \in B$, and let $n > 0$ be given. If $Tree_{\mathcal{A}}(a_0, n) \cong$*

$Tree_{\mathcal{B}}(b_0, n)$, then they are isomorphic via a Δ^0_2-function.

Proof. In the proof of Lemma 2.2.2, we showed that we can reveal all vertices and edges of $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ with a Δ^0_2-oracle. Once we know all of the vertices and edges of the two finite truncated trees (which is Δ^0_2-computable), determining an isomorphism between them is a computable process. Therefore, $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ are Δ^0_2-isomorphic. □

Lemma 2.3.3. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with non-cyclic elements $a_0 \in A$ and $b_0 \in B$. If $Tree_{\mathcal{A}}(a_0) \cong Tree_{\mathcal{B}}(b_0)$, then they are Δ^0_3-isomorphic.*

Proof. We prove the lemma by constructing a Δ^0_3-isomorphism $h : Tree_{\mathcal{A}}(a_0) \to Tree_{\mathcal{B}}(b_0)$. We do this in stages as follows:

Stage 0: Define $h_0 : Tree_{\mathcal{A}}(a_0, 0) \to Tree_{\mathcal{B}}(b_0, 0)$ by simply letting $h_0(a_0) = b_0$.

Stage s+1: Assume that from stage s, we have a partial isomorphism h_s from $Tree_{\mathcal{A}}(a_0, s)$ to $Tree_{\mathcal{B}}(b_0, s)$. At stage $s + 1$, we seek to define h_{s+1}, a partial isomorphism from $Tree_{\mathcal{A}}(a_0, s + 1)$ to $Tree_{\mathcal{B}}(b_0, s + 1)$. Unfortunately, we cannot simply *extend* h_s to h_{s+1}, since there may not exist such a partial isomorphism h_{s+1} that extends h_s. For example, h_s may have mapped an element $x \in tree_{\mathcal{A}}(a_0|s)$ with one pre-image to an element $y \in tree_{\mathcal{B}}(b_0|s)$ with two pre-images. Then h_s would still be a partial isomorphism from $Tree_{\mathcal{A}}(a_0, s)$ to $Tree_{\mathcal{B}}(b_0, s)$, but once we reveal the $(s + 1)^{th}$ level of $Tree_{\mathcal{A}}(a_0)$ and $Tree_{\mathcal{B}}(b_0)$, we would see that h_s "made a mistake", and cannot be extended to a partial isomorphism h_{s+1}.

So at the beginning of stage $s+1$, we reveal $Tree_{\mathcal{A}}(a_0, s+1)$ and $Tree_{\mathcal{B}}(b_0, s+1)$,

and find the first partial isomorphism $p : tree_{\mathcal{A}}(a_0, s+1) \to tree_{\mathcal{B}}(b_0, s+1)$ such that $card[Eq(h_s, p \upharpoonright_{tree_{\mathcal{A}}(a_0,s)})]$ is maximal, where $Eq(f, g)$ is the *equalizer of f and g*, i.e., the set of all elements x such that $f(x) = g(x)$. Then define $h_{s+1} = p$ and go on to stage $s + 2$.

This ends the construction. Finally, let $h = \lim_s h_s$. To complete the proof of the lemma, we must prove two claims:

Claim 1. The function h is an isomorphism from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$. That is, $\lim_s h_s$ exists.

Proof of Claim 1. Let $x \in tree_{\mathcal{A}}(a_0)$. We need to show that there exists a stage s such that for all $t > s$, $h_s(x) = h_t(x)$. Suppose $x \in tree_{\mathcal{A}}(a_0|n)$. Then at the end of stage n, $h_n(x)$ is defined. If there exists an isomorphism $H : Tree_{\mathcal{A}}(a_0) \to Tree_{\mathcal{B}}(b_0)$ such that $H(x) = h_n(x)$, then we are done, since by the nature of the construction we will never have to redefine the partial isomorphism on x after stage n. Otherwise, no such isomorphism H exists, and there will be a stage $r > n$ such that $h_r(x) \neq h_{r-1}(x)$, that is, a stage when the partial isomorphism is redefined on x. However, this can happen at only finitely many stages, since there are only finitely many elements in $tree_{\mathcal{B}}(b_0|n)$ to which x can be mapped, and at least one of these elements represents the image of x under an isomorphism $H : Tree_{\mathcal{A}}(a_0) \to Tree_{\mathcal{B}}(b_0)$. Thus, there must be a stage $s \geq n$ such that for all $t > s$, $h_t(x) = h_s(x)$.

Claim 2. The isomorphism h is a Δ^0_3-function.

Proof of Claim 2. At each stage $s+1$ of the construction, we reveal $Tree_{\mathcal{A}}(a_0, s+1)$ and $Tree_{\mathcal{B}}(b_0, s+1)$, which, from the proof of Lemma 2.2.2, is Δ^0_2-computable. Then we can computably find all of the (finitely many) partial isomorphisms p from

$Tree_{\mathcal{A}}(a_0, s+1)$ to $Tree_{\mathcal{B}}(b_0, s+1)$, computably determine $card[Eq(h_s, p \restriction_{tree_{\mathcal{A}}(a_0,s)})]$ for each p, then define h_{s+1} accordingly. Hence, each partial isomorphism h_s is Δ^0_2-computable. Since $h = \lim_s h_s$ is the limit of Δ^0_2-computable functions, h is a Δ^0_3-function. □

Lemma 2.3.4. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures with cyclic elements $a_0 \in A$ and $b_0 \in B$. If $exTree_{\mathcal{A}}(a_0) \cong exTree_{\mathcal{B}}(b_0)$, then they are Δ^0_3-isomorphic.*

Proof. Given a Δ^0_3-oracle, we first decide if the following Σ^0_1-formula is true:

$$(\exists a_1, a_2)[f(a_1) = f(a_2) = a_0 \wedge a_1 \neq a_2] \qquad (*)$$

If $(*)$ does not hold, then a_0 only has one pre-image, which must also be a cyclic element. Thus, $exTree_{\mathcal{A}}(a_0)$ is the empty tree and so, by assumption, $exTree_{\mathcal{B}}(b_0)$ is the empty tree as well. So the two exclusive trees are trivially computably isomorphic, and thus clearly Δ^0_3-isomorphic.

If $(*)$ does hold, then there exists a non-cyclic element $a_1 \in exTree_{\mathcal{A}}(a_0)$ such that $f(a_1) = a_0$. Then by assumption, there is a non-cyclic element $b_1 \in exTree_{\mathcal{B}}(b_0)$ such that $g(b_1) = b_0$. Furthermore, $Tree_{\mathcal{A}}(a_1)$ and $Tree_{\mathcal{B}}(b_1)$ are isomorphic. So by Lemma 2.3.3, $Tree_{\mathcal{A}}(a_1)$ and $Tree_{\mathcal{B}}(b_1)$ are Δ^0_3-isomorphic. Therefore, by simply mapping a_0 to b_0 and using our Δ^0_3-oracle to map $Tree_{\mathcal{A}}(a_1)$ to $Tree_{\mathcal{B}}(b_1)$, we have that $exTree_{\mathcal{A}}(a_0)$ and $exTree_{\mathcal{B}}(b_0)$ are Δ^0_3-isomorphic. □

Lemma 2.3.5. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures. Suppose that $A_0 \subseteq A$ and $B_0 \subseteq B$ consist of the elements in a K-cycle of A and*

B respectively, with cyclic elements $c_1, ..., c_K \in A_0$ *and* $d_1, ...d_K \in B_0$. *If* $C_{\mathcal{A}}(c_1) \cong C_{\mathcal{B}}(d_1)$, *then they are* Δ^0_3*-isomorphic. That is, all K-cycles are* Δ^0_3*-categorical.*

Proof. Given a Δ^0_3-oracle, we wish to compute the isomorphism from $C_{\mathcal{A}}(c_1)$ to $C_{\mathcal{B}}(d_1)$. Clearly, any isomorphism between the two K-cycles must isomorphically map the cyclic elements from A_0 to those in B_0. There are only k such isomorphisms, each of which is completely determined by where it maps c_1. For example, if c_1 is mapped to d_2, then c_2 must be mapped to d_3, c_3 mapped to d_4, etc. If c_1 is mapped to d_K, then c_2 must be mapped to d_1, c_3 mapped to d_2, etc.

So define $h_j : \{c_1, ..., c_K\} \to \{d_1, ...d_K\}$ to be the isomorphism on the cyclic elements such that $h_j(c_1) = d_j$. (Thus, h_j is fixed for all other cyclic elements in A_0.) Then for each j with $1 \leq j \leq K$, by Lemma 2.2.4, we can use a Δ^0_3-oracle to check if $exTree_{\mathcal{A}}(c_i) \cong exTree_{\mathcal{B}}(h_j(c_i))$ for all i with $1 \leq i \leq K$. Since there are only finitely many j and finitely many i to check for each j, this process is in fact Δ^0_3-computable. And by assumption, there exists a j with $1 \leq j \leq K$ such that $exTree_{\mathcal{A}}(c_i) \cong exTree_{\mathcal{B}}(h_j(c_i))$ for all i with $1 \leq i \leq K$. So once we find such a j, we map the cyclic elements from A_0 to those in B_0 using h_j, then use our Δ^0_3-oracle to isomorphically map $exTree_{\mathcal{A}}(c_i)$ to $exTree_{\mathcal{B}}(h_j(c_i))$ for all i with $1 \leq i \leq K$. Therefore, we have a Δ^0_3-computable algorithm to determine the isomorphism from $C_{\mathcal{A}}(c_1)$ to $C_{\mathcal{B}}(d_1)$. □

Lemma 2.3.6. *Let* $\mathcal{A} = (A, f)$ *and* $\mathcal{B} = (B, g)$ *be two computable* (2,1):1 *structures with elements* $a_0 \in A$ *and* $b_0 \in B$ *such that both* a_0 *and* b_0 *are in* $\mathbb{Z}$*-chains. If* $C_{\mathcal{A}}(a_0) \cong C_{\mathcal{B}}(b_0)$, *then they are* Δ^0_4*-isomorphic. That is, all* $\mathbb{Z}$*-chains are* Δ^0_4*-categorical.*

Proof. Like the proof of Lemma 2.3.3, our goal is to construct a Δ^0_4-isomorphism h from the $\mathbb{Z}$-chain containing a_0 to the $\mathbb{Z}$-chain containing b_0. First, recall the observation from Lemma 2.2.6 that these two $\mathbb{Z}$-chains are isomorphic if and only if the following is true:

$$(\exists p, q)(\forall t)[Tree_{\mathcal{A}}(f^{p+t}(a_0)) \cong Tree_{\mathcal{B}}(g^{q+t}(b_0))] \qquad (*)$$

With this characterization in mind, we construct our isomorphism in stages as follows:

Stage 0: Assume we have an effective enumeration of all elements in $\mathbb{N} \times \mathbb{N}$. Find the first ordered pair (call it (p_0, q_0)) such that $Tree_{\mathcal{A}}(f^{p_0}(a_0)) \cong Tree_{\mathcal{B}}(g^{q_0}(b_0))$. Then construct a partial isomorphism h_0 from $Tree_{\mathcal{A}}(f^{p_0}(a_0))$ to $Tree_{\mathcal{B}}(g^{q_0}(b_0))$, and set $t_0 = 0$.

Stage s+1: Suppose that from stage s we are given a natural number t_s, an ordered pair (p_s, q_s), and a partial isomorphism h_s from $Tree_{\mathcal{A}}(f^{p_s+t_s}(a_0))$ to $Tree_{\mathcal{B}}(g^{q_s+t_s}(b_0))$. If $Tree_{\mathcal{A}}(f^{p_s+t_s+1}(a_0)) \cong Tree_{\mathcal{B}}(g^{q_s+t_s+1}(b_0))$, then we define h_{s+1} to be an isomorphism from $Tree_{\mathcal{A}}(f^{p_s+t_s+1}(a_0))$ to $Tree_{\mathcal{B}}(g^{q_s+t_s+1}(b_0))$, set $t_{s+1} = t_s + 1$, set $(p_{s+1}, q_{s+1}) = (p_s, q_s)$, and proceed to stage $s + 2$. Otherwise, we find the next ordered pair (x, y) in our enumeration such that $Tree_{\mathcal{A}}(f^x(a_0)) \cong Tree_{\mathcal{B}}(g^y(b_0))$, define an isomorphism h_{s+1} from $Tree_{\mathcal{A}}(f^x(a_0))$ to $Tree_{\mathcal{B}}(g^y(b_0))$, set $t_{s+1} = 0$, set $(p_{s+1}, q_{s+1}) = (x, y)$, and proceed to stage $s + 2$.

This ends the construction. Let $h = \lim_s h_s$. To verify the correctness of the construction, we must establish the following claims:

Claim 1. The function h is an isomorphism from $C_{\mathcal{A}}(a_0)$ to $C_{\mathcal{B}}(b_0)$. That is, $\lim_s h_s$ exists.

Proof of Claim 1. By the assumption that $C_{\mathcal{A}}(a_0) \cong C_{\mathcal{B}}(b_0)$, if we search through the ordered pairs in $\mathbb{N} \times \mathbb{N}$, then at some stage s we will eventually find the first pair (p_s, q_s) that satisfies $(*)$. Then for all stages $r > s$, we have that h_r is a proper extension of h_{r-1}. So given any $x \in \mathcal{O}_{\mathcal{A}}(a_0)$, once $h_r(x)$ is defined at a stage $r \geq s$, we will never redefine the image of x in B. Therefore, $\lim_s h_s$ exists for all $x \in \mathcal{O}_{\mathcal{A}}(a_0)$.

Claim 2. The isomorphism h is a Δ^0_4-function.

Proof of Claim 2. At stage 0, we check each ordered pair in $\mathbb{N} \times \mathbb{N}$ until we find a pair (p_0, q_0) with $Tree_{\mathcal{A}}(f^{p_0}(a_0)) \cong Tree_{\mathcal{B}}(g^{q_0}(b_0))$. By Lemma 2.2.3, determining if $Tree_{\mathcal{A}}(f^{p_0}(a_0))$ and $Tree_{\mathcal{B}}(g^{q_0}(b_0))$ are isomorphic is Π^0_2, and thus certainly Δ^0_3. Then we define an isomorphism h_0 from $Tree_{\mathcal{A}}(f^{p_0}(a_0))$ to $Tree_{\mathcal{B}}(g^{q_0}(b_0))$, which by Lemma 2.3.3 is Δ^0_3-computable. So the first stage of our construction is Δ^0_3-computable.

Then at each stage $s+1$ of the construction, we first check if $Tree_{\mathcal{A}}(f^{p_s+t_s+1}(a_0)) \cong Tree_{\mathcal{B}}(g^{q_s+t_s+1}(b_0))$. Again, this is Δ^0_3-computable. If the two Trees are isomorphic, we define an isomorphism between them, which again is Δ^0_3-computable. Otherwise, we search for a new pair (x, y) such that $Tree_{\mathcal{A}}(f^x(a_0)) \cong Tree_{\mathcal{B}}(g^y(b_0))$, then define an isomorphism between these two Trees. This is essentially the same computation done at stage 0, and is thus also Δ^0_3-computable.

So at every stage, we are defining a Δ^0_3-partial isomorphism h_s. Therefore, since h is the limit of a sequence of Δ^0_3-functions, the isomorphism h is a Δ^0_4-function. $\square$

We are now ready to prove the main theorem for this section.

Theorem 2.3.7. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable* (2,1):1 *structures. If $\mathcal{A} \cong \mathcal{B}$, then they are Δ^0_4-isomorphic. That is, all* (2,1):1 *structures are Δ^0_4-categorical.*

Proof. We will use a standard back-and-forth argument to construct a Δ^0_4-isomorphism $h : A \to B$ in stages as follows:

Stage 0: Choose an element $a_0 \in A$, and let $G_0 = C_{\mathcal{A}}(a_0)$. Search through the elements in B until we find an element b_0 such that $G_0 \cong C_{\mathcal{B}}(b_0)$. Let $H_0 = C_{\mathcal{B}}(b_0)$. Then define h_0 to be an isomorphism from G_0 to H_0.

Stage 1: Determine if the following statement holds:

$$(\exists b_1 \in B)(b_1 \notin H_0)$$

If the statement above is false, then our construction is done. Otherwise, find the least such b_1 and let $H_1 = C_{\mathcal{B}}(b_1)$. Then, find an element $a_1 \in A$ such that $a_1 \notin G_0$ and $C_{\mathcal{A}}(a_1) \cong H_1$. Let $G_1 = C_{\mathcal{A}}(a_1)$ and define h_1 to be an isomorphism from G_1 to H_1.

Stage 2s+1: Suppose that from stage $2s$ we have connected components $G_0, G_1, ..., G_{2s}$ from $\mathcal{A}$ and isomorphic connected components $H_0, H_1, ..., H_{2s}$ from $\mathcal{B}$, with isomorphisms $h_n : G_n \to H_n$ for $0 \leq n \leq 2s$. Determine if the following statement holds:

$$(\exists b_{2s+1} \in B)[\bigwedge_{0 \leq n \leq 2s} b_{2s+1} \notin H_n] \qquad (*)$$

If $(*)$ does not hold, then the construction is done. Otherwise, find the least

such b_{2s+1} and let $H_{2s+1} = C_{\mathcal{B}}(b_{2s+1})$. Then find an element $a_{2s+1} \in A$ such that $a_{2s+1} \notin \bigcup_{0 \leq n \leq 2s} G_n$ and $C_{\mathcal{A}}(a_{2s+1}) \cong H_{2s+1}$. Let $G_{2s+1} = C_{\mathcal{A}}(a_{2s+1})$ and define h_{2s+1} to be an isomorphism from G_{2s+1} to H_{2s+1}. Then proceed to the next stage.

Stage 2s+2: Suppose that from stage $2s+1$ we have connected components $G_0, G_1, ..., G_{2s+1}$ from $\mathcal{A}$ and isomorphic connected components $H_0, H_1, ..., H_{2s+1}$ from $\mathcal{B}$, with isomorphisms $h_n : G_n \to H_n$ for $0 \leq n \leq 2s+1$. Determine if the following statement holds:

$$(\exists a_{2s+2} \in A)[\bigwedge_{0 \leq n \leq 2s+1} a_{2s+2} \notin G_n] \qquad (**)$$

If $(**)$ does not hold, then the construction is done. Otherwise, find the least such a_{2s+2} and let $G_{2s+2} = C_{\mathcal{A}}(a_{2s+2})$. Then find an element $b_{2s+2} \in B$ such that $b_{2s+2} \notin \bigcup_{0 \leq n \leq 2s+1} H_n$ and $C_{\mathcal{B}}(b_{2s+2}) \cong G_{2s+2}$. Let $H_{2s+2} = C_{\mathcal{B}}(b_{2s+2})$ and define h_{2s+2} to be an isomorphism from G_{2s+2} to H_{2s+2}. Then proceed to the next stage.

Finally, let $h = \bigcup_s h_s$.

To see that h is a Δ^0_4-isomorphism from $\mathcal{A}$ to $\mathcal{B}$, first observe that at each even stage, we find a connected component in $\mathcal{B}$ isomorphic to some connected component in $\mathcal{A}$ and define an isomorphism between the two components, and at every odd stage we do essentially the same thing, but in reverse. So it suffices to show that each odd stage $2s+1$ is Δ^0_4-computable and thus, each isomorphism h_{2s+1} is a Δ^0_4-isomorphism. The same argument can be used for the even stages.

We start each odd stage by checking the truth of $(*)$. By Lemma 2.1.4(4), the bracketed part of $(*)$ is $\emptyset'$-computable, and thus a Δ^0_2-formula. So $(*)$ itself is a Σ^0_2-sentence, and thus certainly Δ^0_4. If $(*)$ does not hold, we are done. Otherwise, we can

search for and find the least element b_{2s+1}, which is also computable via a Δ^0_4-oracle.

Next, we find an element $a_{2s+1} \in A$ such that $a_{2s+1} \notin \bigcup_{0\leq n\leq 2s} G_n$ and $C_{\mathcal{A}}(a_{2s+1}) \cong H_{2s+1}$. The first of these conditions is again Δ^0_2. The second condition is Σ^0_3 by Lemma 2.2.7. So finding an element in A which is in a new connected component isomorphic to $C_{\mathcal{B}}(b_{2s+1})$ is also Δ^0_4-computable. Finally, we define the isomorphism h_{2s+1} from G_{2s+1} to H_{2s+1}. If G_{2s+1} and H_{2s+1} are both K-cycles, then h_{2s+1} is necessarily Δ^0_3 by Lemma 2.3.5, and if G_{2s+1} and H_{2s+1} are both $\mathbb{Z}$-chains, then h_{2s+1} is necessarily Δ^0_4 by Lemma 2.3.6. In either case, h_{2s+1} is a Δ^0_4-isomorphism for all s. (Similarly, h_{2s} is a Δ^0_4-isomorphism for all s.) So every stage of the construction is Δ^0_4-computable.

Therefore, h is a Δ^0_4-isomorphism, and all (2,1):1 structures are Δ^0_4-categorical. □

Corollary 2.3.8. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure. If $\mathcal{A}$ contains no $\mathbb{Z}$-chains, then $\mathcal{A}$ is Δ^0_3-categorical.*

Proof. Suppose we are given a computable (2,1):1 structure $\mathcal{B}$ isomorphic to $\mathcal{A}$. By Lemma 2.3.5, since every connected component in $\mathcal{A}$ is a K-cycle, we can isomorphically map every component in $\mathcal{A}$ to a component in $\mathcal{B}$ and vice-versa using a Δ^0_3-oracle. Then simply use the same back-and-forth argument in Theorem 2.3.7 to construct a Δ^0_3-isomorphism from $\mathcal{A}$ to $\mathcal{B}$. □

2.4 The Branching Function and the Branch Isomorphism Function

In the last section, we saw that all (2,1):1 structures are Δ^0_4-categorical, while all such structures without $\mathbb{Z}$-chains are Δ^0_3-categorical. We would like to explore some structural and computability-theoretic properties of (2,1):1 structures which would lower the complexity of isomorphisms between such structures. In this manner, we can begin to characterize the (2,1):1 structures that are computably categorical.

In this section, we define two additional functions on (2,1):1 structures, the *branching function* and the *branch isomorphism function*, and investigate how the computability of these functions affects the categoricity of our structures.

Definition 2.4.1. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure. The *branching function of* $\mathcal{A}$, denoted by $\beta_{\mathcal{A}} : \mathbb{N} \to \{1, 2\}$, is defined as:

$$\beta_{\mathcal{A}}(x) = \begin{cases} 1 & \text{if } (\forall x_1, x_2)(f(x_1) = f(x_2) = x \implies x_1 = x_2), \\ 2 & \text{if } (\exists x_1, x_2)(f(x_1) = f(x_2) = x \wedge x_1 \neq x_2). \end{cases}$$

Essentially, the branching function takes an element $x \in A$ as an input, and outputs the number of pre-images of x under f. By the definition of a (2,1):1 structure, the branching function is clearly defined for any (2,1):1 structure $\mathcal{A}$ and for all $x \in A$. Also, the branching function may not be computable, even if the underlying structure $\mathcal{A}$ is computable. (We construct an example of such a structure later in the section.) However, we can easily put an upper bound on the Turing degree of the branching

function of a computable (2,1):1 structure, which we do in the following lemma.

Lemma 2.4.2. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure, with branching function $\beta_{\mathcal{A}}$. Then $\beta_{\mathcal{A}} \leq_T \emptyset'$.*

Proof. By definition, $\beta_{\mathcal{A}}(x) = 2$ if and only if the following formula holds:

$$(\exists x_1, x_2)(f(x_1) = f(x_2) = x \wedge x_1 \neq x_2) \qquad (*)$$

Since $(*)$ is a Σ^0_1-sentence, we can compute if it holds for a given x using a $\mathbf{0}'$-oracle. Thus, the value of $\beta_{\mathcal{A}}(x)$ is also $\mathbf{0}'$-computable, and therefore $\beta_{\mathcal{A}} \leq_T \emptyset'$. □

For the sake of convenience, we wish to give a name to all of those elements in a (2,1):1 structure with only one pre-image, as well as a name to those elements with two pre-images. Hence, we have the following definition.

Definition 2.4.3. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, with branching function $\beta_{\mathcal{A}}$. Then the *hair set of $\mathcal{A}$,* denoted by $I_{\mathcal{A}}$, is defined as:

$$I_{\mathcal{A}} = \{x \in A : \beta_{\mathcal{A}}(x) = 1\}$$

Also, the *split hair set of $\mathcal{A}$,* denoted by $\Lambda_{\mathcal{A}}$, is defined as:

$$\Lambda_{\mathcal{A}} = \{x \in A : \beta_{\mathcal{A}}(x) = 2\}$$

We informally refer to elements in $I_{\mathcal{A}}$ as "hairs" and elements in $\Lambda_{\mathcal{A}}$ as "split hairs". It should be obvious that $I_{\mathcal{A}}$ and $\Lambda_{\mathcal{A}}$ are complements in A for all $\mathcal{A}$. Thus, $I_{\mathcal{A}}$, $\Lambda_{\mathcal{A}}$, and $\beta_{\mathcal{A}}$ are all Turing equivalent.

If we assume computability of the branching function, then we can lower the upper bound on the complexity of isomorphisms between two such (2,1):1 structures.

Proposition 2.4.4. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure, and suppose that the branching function is computable in every computable copy of $\mathcal{A}$. Then $\mathcal{A}$ is Δ^0_3-categorical.*

Proof. We prove the proposition by essentially running through the proof of Theorem 2.3.7 (as well as the lemmas preceding the theorem).

First, let $\mathcal{B} = (B, g)$ be a computable (2,1):1 structure isomorphic to $\mathcal{A}$. So, by assumption, $\mathcal{B}$ has a computable branching function. Let $a_0 \in A$ and $b_0 \in B$ be non-cyclic elements such that $Tree_{\mathcal{A}}(a_0, n) \cong Tree_{\mathcal{B}}(b_0, n)$ for some n. Since the branching function is computable in both structures, given all of the elements in the m^{th} level of the truncated trees, we can computably determine all of the elements in the $(m+1)^{th}$ level of both truncated trees. This is done by using the branching functions to determine if each element in the m^{th} level has one or two pre-images, then searching for all pre-images of all elements in that level. Thus, we can effectively reveal all vertices and edges in both truncated trees, then find an isomorphism between the two finite graphs. Hence, $Tree_{\mathcal{A}}(a_0, n)$ and $Tree_{\mathcal{B}}(b_0, n)$ are computably isomorphic.

Next, suppose that $Tree_{\mathcal{A}}(a_0) \cong Tree_{\mathcal{B}}(b_0)$. Then we can build an isomorphism h from one tree to another in stages in the same manner as in the proof of Lemma 2.3.3. The only difference here is that each partial isomorphism h_s from one truncated tree to another is a computable function, and thus $h = \lim_s h_s$ is the limit of computable functions. So h is a Δ^0_2-function, and therefore $Tree_{\mathcal{A}}(a_0)$ and $Tree_{\mathcal{B}}(b_0)$ are Δ^0_2-isomorphic.

A similar argument shows that if $c_1 \in A$ and $d_1 \in B$ are cyclic elements in $\mathcal{A}$ and

$\mathcal{B}$, then $exTree_{\mathcal{A}}(c_1)$ and $exTree_{\mathcal{B}}(d_1)$ are also Δ^0_2-isomorphic. Thus, by the same method used in the proof of Lemma 2.3.5, if $C_{\mathcal{A}}(c_1) \cong C_{\mathcal{B}}(d_1)$, we can isomorphically map the K-cycle containing c_1 to the one containing d_1 using a Δ^0_2-oracle.

If $a_0 \in A$ and $b_0 \in B$ are in isomorphic $\mathbb{Z}$-chains, then we use the same construction in Lemma 2.3.6 to build an isomorphism h from $C_{\mathcal{A}}(a_0)$ to $C_{\mathcal{B}}(b_0)$. Again, the only difference here is that each partial isomorphism h_s is a Δ^0_2-function. Thus, $h = \lim_s h_s$ is a Δ^0_3-isomorphism from one $\mathbb{Z}$-chain to the other.

Now, with the assumption that the branching functions are computable, we can modify the proofs of the lemmas in Section 2.2 to show that the isomorphism problem is computable for truncated Trees, Π^0_1 for full Trees, exclusive Trees, and K-cycles, Σ^0_2 for $\mathbb{Z}$-chains (and thus for arbitrary connected components), and Π^0_3 for (2,1):1 structures. So given any $a_0 \in A$ and $b_0 \in B$, the isomorphism problem for $C_{\mathcal{A}}(a_0)$ and $C_{\mathcal{B}}(b_0)$ is Σ^0_2. Thus, given a Δ^0_3-oracle, we can employ the same back-and-forth argument used in Theorem 2.3.7 to isomorphically map every connected component in $\mathcal{A}$ to one in $\mathcal{B}$, and vice-versa. Therefore, $\mathcal{A}$ is Δ^0_3-categorical. □

Corollary 2.4.5. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure without any $\mathbb{Z}$-chains, and suppose that the branching function is computable in every computable copy of $\mathcal{A}$. Then $\mathcal{A}$ is Δ^0_2-categorical.*

Proof. This proof is very similar to the one given for Corollary 2.3.8. The only difference is that if we are given $\mathcal{B}$, a computable copy of $\mathcal{A}$, then every connected component in one structure can be isomorphically mapped to a connected component in the other structure via a Δ^0_2-function now. Thus, using the same back-and-forth

argument, there is a Δ^0_2-isomorphism from $\mathcal{A}$ to $\mathcal{B}$. $\square$

We now give an example of our earlier claim that computability of a (2,1):1 structure does not imply computability of its branching function.

Proposition 2.4.6. *There exists a computable* (2,1):1 *structure* $\mathcal{A} = (A, f)$ *such that* $\beta_{\mathcal{A}}$ *is not computable.*

Proof. Our goal is to construct a computable (2,1):1 structure $\mathcal{A}$ such that $\Lambda_{\mathcal{A}}$ is not a computable set. So let C be some non-computable c.e. set containing 0. Then C has a partial computable characteristic function χ_C such that $\chi_C(x) = 1$ if $x \in C$, and $\chi_C(x) = \uparrow$ if $x \notin C$. We now build $\mathcal{A} = (A, f)$ to be a single 1-cycle in stages as follows:

Stage 0: Let $A_0 = \{0\}$, and let $f_0(0) = 0$.

Stage 1: Let $A_1 = \{0, 1\}$, and let $f_1(0) = 0$ and $f_1(1) = 0$.

Stage s+1: Suppose we have A_s and f_s from stage s. Find the least a such that:

(1) $a \in I_{\mathcal{A}_{s-1}}$, and

(2) $\chi_{C,s}(a) \downarrow = 1$.

If no such a exists, simply extend A_s to A_{s+1} and f_s to f_{s+1} by attaching one new number (not already in A_s) to each number in $extree_{\mathcal{A}_s}(0|s)$. Then move on to the next stage.

If such an a does exist, take the least number x_0 not already in A_s and define $f_s(x_0) = a$. Denote the level of x_0 in $extree_{\mathcal{A}_s}(0)$ by l, and extend the Tree of x_0 by attaching $s - l$ new numbers $x_1, x_2, ..., x_{s-l}$ to x such that $f_s(x_i) = x_{i-1}$ for

$1 \leq i \leq s - l$. Enumerate $x_1, x_2, ..., x_{s-l}$ into A_s. Then extend $exTree_{\mathcal{A}_s}(0, s)$ to $exTree_{\mathcal{A}_s}(0, s+1)$ by attaching one new number to each number in $extree_{\mathcal{A}_s}(0|s)$. Define $A_{s+1} = A_s \cup \{extree_{\mathcal{A}_s}(0|s+1)\}$ and extend f_s to f_{s+1} accordingly. Then move on to the next stage.

This completes the construction of $\mathcal{A}$. Let $A = \bigcup_s A_s$ and $f = \bigcup_s f_s$. We must now verify two claims.

Claim 1. $\mathcal{A} = (A, f)$ is a computable (2,1):1 structure.

Proof of Claim 1. $A = \omega$, and is thus clearly computable. To compute $f(x)$, we simply run through the construction until we reach the stage s where x appears, then determine $f_s(x)$. Due to the nature of the construction, once f_s is defined on an element, we never redefine it at a later stage. Thus, $f_s(x) = f(x)$, and f is computable. Therefore, $\mathcal{A}$ is computable.

To see that $\mathcal{A}$ is a (2,1):1 structure, first observe that 0 has exactly two pre-images (0 and 1). Also notice that at every stage, we extend the exclusive tree of 0 by one level, so every element has at least one pre-image. The only instance where an element is given an additional pre-image is if it had exactly one pre-image, so no element has more than two pre-images. Thus, every element either has exactly one or exactly two pre-images, making $\mathcal{A}$ a (2,1):1 structure.

Claim 2. $\beta_{\mathcal{A}}$ is not computable.

Proof of Claim 2. Observe that $x \in C$ if and only if x has two pre-images, which is if and only if $\beta_{\mathcal{A}}(x) = 2$. So C is Turing equivalent to $\beta_{\mathcal{A}}$. Therefore, since C is not a computable set, $\beta_{\mathcal{A}}$ cannot be computable. □

Although we know from Corollary 2.4.5 that computability of the branching function in all computable copies of a (2,1):1 structure and a lack of $\mathbb{Z}$-chains guarantees that the structure is Δ^0_2-categorical, these properties may still not be enough to ensure computable categoricity. So we introduce another function that provides us with even more crucial information about our structures.

Definition 2.4.7. Let $\mathcal{A} = (A, f)$ be a (2,1):1 structure, and let $x \in \Lambda_{\mathcal{A}}$ with distinct pre-images x_1 and x_2. The *branch isomorphism function of* $\mathcal{A}$, denoted by $iso_{\mathcal{A}} : \Lambda_{\mathcal{A}} \to \{0, 1\}$, is defined as:

$$iso_{\mathcal{A}}(x) = \begin{cases} 0 & \text{if } Tree_{\mathcal{A}}(x_1) \ncong Tree_{\mathcal{A}}(x_2), \\ 1 & \text{if } Tree_{\mathcal{A}}(x_1) \cong Tree_{\mathcal{A}}(x_2). \end{cases}$$

Thus, given an element x with two distinct pre-images, the branch isomorphism function tells us if the Trees of those pre-images are isomorphic to each other, i.e., if the branches stemming from x are isomorphic. Note that if $c_1 \in \Lambda_{\mathcal{A}}$ is a cyclic element, then $iso_{\mathcal{A}}(c_1) = 0$. This is because one pre-image of c_1 will be a cyclic element c_K while the other pre-image will be a non-cyclic element a, so $Tree_{\mathcal{A}}(c_K)$ will be the entire K-cycle containing c_1, while $Tree_{\mathcal{A}}(a)$ will be an infinite directed binary tree with no cycles. So $Tree_{\mathcal{A}}(c_K)$ is clearly not isomorphic to $Tree_{\mathcal{A}}(a)$.

It is also important to note that the domain of $iso_{\mathcal{A}}$ consists of exactly those elements with two distinct pre-images; therefore, the domain of $iso_{\mathcal{A}}$ may not be computable. In fact, if $\beta_{\mathcal{A}}$ is not computable, then neither is the domain of $iso_{\mathcal{A}}$. We generally avoid this issue by assuming that the branching function is computable

whenever discussing the branch isomorphism function. However, computability of the branching function does not guarantee computability of the branch isomorphism function, as we will see at the end of this section.

We now state an important theorem, which demonstrates the effect of the computability of the branch isomorphism function on the complexity of isomorphisms between our structures.

Theorem 2.4.8. *Let $\mathcal{A} = (A, f)$ and $\mathcal{B} = (B, g)$ be two computable isomorphic* (2,1):1 *structures, both with a computable branching function and a computable branch isomorphism function. If $a_0 \in A$ and $b_0 \in B$ are non-cyclic elements such that $Tree_{\mathcal{A}}(a_0) \cong Tree_{\mathcal{B}}(b_0)$, then the two Trees are computably isomorphic. Likewise, if $c_1 \in A$ and $d_1 \in B$ are cyclic elements such that $exTree_{\mathcal{A}}(c_1) \cong exTree_{\mathcal{B}}(d_1)$, then the two exclusive Trees are computably isomorphic.*

Proof. We construct a computable isomorphism h from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ in stages as follows:

Stage 0: Define $h_0(a_0) = b_0$.

Stage s+1: Suppose that from stage s we are given h_s, an isomorphism from $Tree_{\mathcal{A}}(a_0, s)$ to $Tree_{\mathcal{B}}(b_0, s)$. For all elements $w \in tree_{\mathcal{A}}(a_0, s)$, define $h_{s+1}(w) = h_s(w)$. Let $x \in tree_{\mathcal{A}}(a_0|s)$ and let $y = h_s(x)$. If $\beta_{\mathcal{A}}(x) = 1$, find x_1, the unique pre-image of x, and find y_1, the unique pre-image of y. Then define $h_{s+1}(x_1) = y_1$.

If $\beta_{\mathcal{A}}(x) = 2$, find x_1 and x_2, the distinct pre-images of x, then find y_1 and y_2, the distinct pre-images of y. If $iso_{\mathcal{A}}(x) = 1$, then define $h_{s+1}(min\{x_1, x_2\}) = min\{y_1, y_2\}$ and $h_{s+1}(max\{x_1, x_2\}) = max\{y_1, y_2\}$, where min and max are defined

under the usual ordering on $\mathbb{N}$. If $iso_{\mathcal{A}}(x) = 0$, then there exists a level n such that $Tree_{\mathcal{A}}(x_1, n) \ncong Tree_{\mathcal{A}}(x_2, n)$. In that case, use $\beta_{\mathcal{A}}$ to reveal the vertices and edges of $Tree_{\mathcal{A}}(x_1)$ and $Tree_{\mathcal{A}}(x_2)$ one level at a time until we find such a level n. Then, use $\beta_{\mathcal{B}}$ to reveal the vertices and edges of $Tree_{\mathcal{B}}(y_1, n)$ and $Tree_{\mathcal{B}}(y_2, n)$. If $Tree_{\mathcal{A}}(x_1, n) \cong Tree_{\mathcal{B}}(y_1, n)$, then define $h_{s+1}(x_1) = y_1$ and $h_{s+1}(x_2) = y_2$. Otherwise, define $h_{s+1}(x_1) = y_2$ and $h_{s+1}(x_2) = y_1$.

Repeat the procedure above for all $x \in tree_{\mathcal{A}}(a_0|s)$, so h_{s+1} is defined on all elements in $tree_{\mathcal{A}}(a_0|s+1)$.

Finally, let $h = \lim_s h_s$. We must now verify that h is a computable isomorphism from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$.

Claim 1. *h is an isomorphism from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$.*

Proof of Claim 1. Suppose h_s is an isomorphism from $Tree_{\mathcal{A}}(a_0, s)$ to $Tree_{\mathcal{B}}(b_0, s)$ such that there exists an isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ with $H \restriction_{tree_{\mathcal{A}}(a_0,s)} = h_s$. Let $x \in tree_{\mathcal{A}}(a_0|s)$ and let $y = h_s(x)$.

If $\beta_{\mathcal{A}}(x) = 1$, then it must be the case that $\beta_{\mathcal{B}}(y) = 1$ as well. So x and y each have a unique pre-image x_1 and y_1, respectively, and thus we can properly extend h_s to h_{s+1} by defining $h_{s+1}(x_1) = y_1$. Furthermore, any isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ that extends h_s must map x_1 to y_1. So there certainly exists an isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ extending h_s that agrees with h_{s+1} on x_1.

If $\beta_{\mathcal{A}}(x) = 2$, then again, it must be the case that $\beta_{\mathcal{B}}(y) = 2$ as well. So let x_1 and x_2 be the distinct pre-images of x, and let y_1 and y_2 be the distinct pre-images of y. If $iso_{\mathcal{A}}(x) = 1$, then it must be the case that $iso_{\mathcal{B}}(y) = 1$ as well. Also, the Trees of both x_1 and x_2 are isomorphic to the Trees of both y_1 and y_2. So regardless of where

h_{s+1} maps x_1 and x_2, there will exist an isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ extending h_s that agrees with h_{s+1} on both x_1 and x_2. And if $iso_{\mathcal{A}}(x) = 0$, then it must be the case that $iso_{\mathcal{B}}(y) = 0$ as well. So due to the construction, if h_{s+1} maps x_1 to y_1, then $Tree_{\mathcal{A}}(x_1) \cong Tree_{\mathcal{B}}(y_1)$, and thus it must be that $Tree_{\mathcal{A}}(x_2) \cong Tree_{\mathcal{B}}(y_2)$. (The reverse statement can also be said if h_{s+1} maps x_1 to y_2.) And since the branches of x (and y) are not isomorphic to each other, any isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ extending h_s must agree with h_{s+1} on x_1 and x_2.

Thus, it is apparent that h_{s+1}, once it is defined on all elements in $tree_{\mathcal{A}}(a_0|s+1)$, is an isomorphism from $Tree_{\mathcal{A}}(a_0|s+1)$ to $Tree_{\mathcal{B}}(b_0|s+1)$. Moreover, there must exist an isomorphism H from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$ that extends h_{s+1}, as the branching functions and the branch isomorphism functions prevent us from "making a mistake" throughout the construction. So, h_{s+1} is a proper extension of h_s for all stages s, and once $h_s(x)$ is defined at a stage s, it is never redefined again. Thus, $h = \lim_s h_s$ exists and is an isomorphism from $Tree_{\mathcal{A}}(a_0)$ to $Tree_{\mathcal{B}}(b_0)$.

Claim 2. The isomorphism h is a computable function.

Proof of Claim 2. Let $x \in tree_{\mathcal{A}}(a_0)$. To determine $h(x)$ we run through the stages of the construction until we define h on x. By the assumption that both branching functions and branch isomorphism functions are computable, each stage of the construction is easily seen to be computable. Therefore, we can effectively determine the image of x under h.

The construction of a computable isomorphism h from $exTree_{\mathcal{A}}(c_1)$ to $exTree_{\mathcal{B}}(d_1)$ is almost identical to the one presented above. The only difference is that at stage 1, after mapping c_1 to d_1 in stage 0, we must then identify via the branching func-

tions if c_1 and d_1 have non-cyclic pre-images. If they don't, then $exTree_{\mathcal{A}}(c_1)$ and $exTree_{\mathcal{B}}(d_1)$ are trivially computably isomorphic. Otherwise, we find the non-cyclic pre-images of both c_1 and d_1 (which is, of course, computable), then define h_1 as a map from the non-cyclic pre-image of c_1 to that of d_1. □

It is worth noting that the construction in Theorem 2.4.8 can be done without the assumption that $\beta_{\mathcal{B}}$ and $iso_{\mathcal{B}}$ are computable.

We conclude this section by presenting an example of a computable (2,1):1 structure with a computable branching function, but a non-computable branch isomorphism function. This supports our intuition that the branch isomorphism function actually provides us with new information about our graphs that cannot be obtained solely from the branching function.

Proposition 2.4.9. *There exists a computable* (2,1):1 *structure* $\mathcal{A} = (A, f)$ *such that* $\beta_{\mathcal{A}}$ *is computable but* $iso_{\mathcal{A}}$ *is not computable.*

Proof. We wish to construct a computable (2,1):1 structure $\mathcal{A}$ such that $\beta_{\mathcal{A}}$ is computable, but no computable function φ_e computes the branch isomorphism function $iso_{\mathcal{A}}$. We will accomplish this by building $\mathcal{A}$ using a standard priority argument to ensure that for all $e \in \omega$, the following requirement P_e is satisfied:

$$P_e : \varphi_e \neq iso_{\mathcal{A}}$$

We start with an effective enumeration of all partial computable functions $\{\varphi_e\}_{e\in\omega}$. Our desired structure $\mathcal{A} = (A, f)$ will again be a single 1-cycle, which we will construct in stages as follows.

Stage 0: Define $A_0 = \{0, 2\}$, $f_0(0) = 0$ and $f_0(2) = 0$.

Stage s+1: Suppose we have A_s and f_s from the previous stage. Let M_s denote the lowest level of the exclusive tree of 0 at the end of stage s, i.e., M_s is the unique number such that $extree_{\mathcal{A}_s}(0|M_s) \neq \emptyset$, and for all $n > M_s$, $extree_{\mathcal{A}_s}(0|n) = \emptyset$.

First, assign φ_e to level M_s, where e is the least number such that φ_e has not been assigned to a level of the exclusive tree at a previous stage. Let L_j denote the level of $extree_{\mathcal{A}_s}(0)$ that φ_j has been assigned to, so $L_e = M_s$. It is important to note that we will only ever assign a partial computable function to a level of the exclusive tree that contains only elements with two pre-images.

Then, find the least $i \leq e$ such that:

(1) $\varphi_{i,s}(x) \downarrow = 1$ for some $x \in L_i$, and

(2) P_i has not yet received attention.

If no such i exists, extend A_s to A_{s+1}, and f_s to f_{s+1}, by attaching two unused even numbers as pre-images to every number in $extree_{\mathcal{A}_s}(0|M_s)$. Set $M_{s+1} = M_s + 1$ and go on to the next stage.

If such an i exists, we say that P_i *requires attention*. If $x \notin extree_{\mathcal{A}_s}(0|M_s)$, let x_1 and x_2 be the distinct pre-images of x, with $x_1 < x_2$. Attach two unused odd numbers to every element that is in both $tree_{\mathcal{A}_s}(x_1)$ and $extree_{\mathcal{A}_s}(0|M_s)$, and attach two unused even numbers to every element that is in both $tree_{\mathcal{A}_s}(x_2)$ and $extree_{\mathcal{A}_s}(0|M_s)$. If $x \in extree_{\mathcal{A}_s}(0|M_s)$, then simply attach one unused odd number and one unused even number to x. In either case, repeat the procedure for every

number in the same level of the exclusive tree of 0 as x. Now, the $(M_s+1)^{th}$ level of the exclusive tree of 0 is complete.

Then, to every odd number in $extree_{\mathcal{A}_s}(0|M_s+1)$, attach exactly one unused even number. To every even number in $extree_{\mathcal{A}_s}(0|M_s+1)$, attach exactly two unused even numbers. This completes the $(M_s+2)^{th}$ level of the exclusive tree of 0. Let

$$A_{s+1} = A_s \cup extree_{\mathcal{A}_s}(0|M_s+1) \cup extree_{\mathcal{A}_s}(0|M_s+2),$$

extend f_s to f_{s+1} as described above, and set $M_{s+1} = M_s + 2$. At this point, P_i has *received attention* and we move on to the next stage.

This ends the construction. Let $A = \bigcup_s A_s$ and $f = \bigcup_s f_s$. We must now prove the following two claims.

Claim 1: The structure $\mathcal{A} = (A, f)$ is a computable (2,1):1 structure with $\beta_{\mathcal{A}}$ computable.

Proof of Claim 1: By construction, $A = \omega$. To compute $f(a)$, we simply run through the construction until we reach the stage s where a appears, then determine $f_s(a)$. Due to the construction, once f_s is defined on an element, we never redefine it at a later stage. Thus, $f_s(a) = f(a)$, and $\mathcal{A}$ is computable.

To see that $\mathcal{A}$ is a (2,1):1 structure, observe that every even number has exactly two pre-images, and every odd number has exactly one pre-image. This also proves that $\beta_{\mathcal{A}}$ is a computable function.

Claim 2: The branch isomorphism function is not computable.

Proof of Claim 2: We prove by induction that each requirement P_e is satisfied.

At the beginning of stage 1, φ_0 is assigned to level $L_0 = M_0 = 1$ of the exclusive tree of 0, so φ_0 is assigned to the single number 2. If $\varphi_{0,s}(2) \downarrow = 1$ for some stage s, then P_0 would require attention at stage s. However, due to the construction, the isomorphism on the branches of 2 would be ruined at stage s, and thus $iso_{\mathcal{A}_t}(2) = 0$ for all stages $t > s$, and $iso_{\mathcal{A}}(2) = 0 \neq 1 = \varphi_0(2)$. Hence, P_0 is satisfied. Otherwise, $\varphi_{0,s}(2) \neq 1$ for any stage s and thus $\varphi_0(2) \neq 1$. But the only requirement that can ruin the isomorphism on the branches of 2 is P_0. Any requirement receiving attention only ruins the isomorphism on the branches of the elements in its assigned level, due to the symmetry of the construction. Thus, for all stages s, $iso_{\mathcal{A}_s}(2) = 1$, which means that $iso_{\mathcal{A}}(2) = 1 \neq \varphi_{0,s}(2)$. Again, P_0 is satisfied.

Now suppose that for all $i < e$, P_i is satisfied. At stage $e + 1$, φ_e is assigned to some level L_e. If there do not exist a stage s and a number x in level L_e such that $\varphi_{e,s}(x) \downarrow = 1$, then P_e is satisfied since $iso_{\mathcal{A}}(x) = 1$ for all $x \in extree_{\mathcal{A}}(0|L_e)$ by the same symmetry argument as before. Otherwise, let t be the first stage at which for all $i < e$, P_i does not require attention and $\varphi_{e,t}(x) \downarrow = 1$ for some x in level L_e. Then at stage t, P_e would require attention, and the construction would ensure that $iso_{\mathcal{A}_r}(x) = 0$ for all stages $r > t$. So, $\varphi_e(x) \neq iso_{\mathcal{A}}(x)$ and again, P_e would be satisfied. Therefore, all requirements are satisfied and $iso_{\mathcal{A}}$ is not computable.

□

Chapter 3

Computable Categoricity of (2,1):1 Structures

In Chapter 2, we introduced (2,1):1 structures and discussed some of their most important properties as they relate to isomorphism complexity. Now, in this chapter, we seek to characterize those (2,1):1 structures that are computably categorical.

As we have established in Corollary 2.4.5, all (2,1):1 structures without $\mathbb{Z}$-chains and with a computable branching function in every computable copy are Δ^0_2-categorical. We also saw from Theorem 2.4.8 that computability of the branch isomorphism function greatly increased our ability to compute isomorphisms of Trees of an element. So naturally, we will begin to look for computably categorical (2,1):1 structures by first examining those structures with these specific properties. In particular, we will only examine (2,1):1 structures without $\mathbb{Z}$-chains.

3.1 N^+-Embeddability

For all (2,1):1 structures with a computable branching function and branch isomorphism function, we have an intuitive necessary condition for computable categoricity.

Lemma 3.1.1. *Let $\mathcal{A} = (A, f)$ be a computable (2,1):1 structure with $\beta_{\mathcal{A}}$ and $iso_{\mathcal{A}}$ computable. If $\mathcal{A}$ is computably categorical, then for every computable copy $\mathcal{B}$ of $\mathcal{A}$, $\beta_{\mathcal{B}}$ and $iso_{\mathcal{B}}$ are computable.*

Proof. Suppose that $\mathcal{A} = (A, f)$ is a computable (2,1):1 structure with $\beta_{\mathcal{A}}$ and $iso_{\mathcal{A}}$ computable, and suppose further that $\mathcal{A}$ is computably categorical. Let $\mathcal{B} = (B, g)$ be a computable copy of $\mathcal{A}$. Since $\mathcal{A}$ is computably categorical, there exists a computable isomorphism $h : \mathcal{A} \to \mathcal{B}$. Thus, given an element $x \in B$ we can compute $\beta_{\mathcal{B}}(x)$ by finding the pre-image of x under h, then using $\beta_{\mathcal{A}}$ to compute if the pre-image in A has one or two immediate predecessors. In other words, $\beta_{\mathcal{B}}(x) = \beta_{\mathcal{A}}(h^{-1}(x))$, which is a composition of computable functions. So $\beta_{\mathcal{B}}$ is computable.

Given $x \in B$, to compute $iso_{\mathcal{B}}(x)$, we first need to decide if x is in the domain of $iso_{\mathcal{B}}$, i.e., if $\beta_{\mathcal{B}}(x) = 2$. We have already shown this process to be computable. Then, if x is in the domain of $iso_{\mathcal{B}}$, we simply compute $iso_{\mathcal{B}}(x)$ in the same way we computed $\beta_{\mathcal{B}}(x)$, since $iso_{\mathcal{B}}(x) = iso_{\mathcal{A}}(h^{-1}(x))$. Therefore, $iso_{\mathcal{B}}$ is computable as well. □

The next theorem gives us our first type of computably categorical (2,1):1 structure. It essentially states that if $\mathcal{A}$ has only finitely many 1-cycles, finitely many 2-cycles, etc., then $\mathcal{A}$ is computably categorical. In particular, $\mathcal{A}$ is computably

categorical if it has only finitely many K-cycles.

Theorem 3.1.2. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure without $\mathbb{Z}$-chains and with $\beta_{\mathcal{A}}$ and $iso_{\mathcal{A}}$ computable. If, for each $k \in \omega$, $\mathcal{A}$ has only finitely many k-cycles, then $\mathcal{A}$ is computably categorical.*

Proof. Suppose that $\mathcal{A}$ is a (2,1):1 structure as described above, and $\mathcal{B}$ is a computable structure isomorphic to $\mathcal{A}$. For each $k \in \omega$, $\mathcal{A}$ has only finitely many k-cycles, and thus A has only finitely many cyclic elements $c_1, ..., c_n$ in those k-cycles, which we can computably identify. So we can non-uniformly and isomorphically map each of these cyclic elements c_i in A to a corresponding cyclic element d_i in B via a computable function. Then, by Theorem 2.4.8, we can construct a computable isomorphism $h_{i,k}$ from $exTree_{\mathcal{A}}(c_i)$ to $exTree_{\mathcal{B}}(d_i)$ for $1 \leq i \leq n$. Let $h_k = \bigcup_i h_{i,k}$. Then h_k is a computable isomorphism from the k-cycles in $\mathcal{A}$ to those in $\mathcal{B}$.

Repeat the procedure above for each $k \in \omega$, and let $h = \bigcup_k h_k$. Since $\mathcal{A}$ has no $\mathbb{Z}$-chains, every element in A is in some k-cycle of $\mathcal{A}$. So, $h : A \to B$ is a computable isomorphism from $\mathcal{A}$ to $\mathcal{B}$. Thus, $\mathcal{A}$ is computably categorical. □

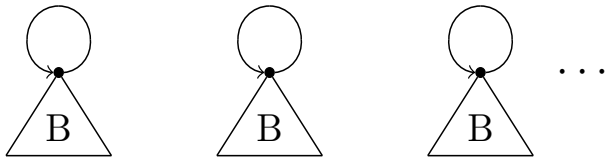
Corollary 3.1.3. *If $\mathcal{A}$ is a computable* (2,1):1 *structure with no $\mathbb{Z}$-chains, finitely many k-cycles for each $k \in \omega$, $\beta_{\mathcal{A}}$ and $iso_{\mathcal{A}}$ computable, and $\mathcal{B}$ is a computable structure such that $\mathcal{A} \cong \mathcal{B}$, then $\beta_{\mathcal{B}}$ and $iso_{\mathcal{B}}$ are also computable.*

Proof. By Theorem 3.1.2, $\mathcal{A}$ is computably categorical. Thus, by Lemma 3.1.1, $\beta_{\mathcal{B}}$ and $iso_{\mathcal{B}}$ are computable. □

The last condition in Theorem 3.1.2 is not necessary for a computable (2,1):1 structure of this type to be computably categorical. For example, consider the struc-

ture $\mathcal{A}$ in Figure 3.1, and assume that it is computable. It is composed of infinitely many 1-cycles, and attached to each one is an infinite full directed binary tree B. Its branching function is trivially computable, as every element has two pre-images. Also, its branch isomorphism function is certainly computable; $iso_{\mathcal{A}}(x) = 0$ if x is a cyclic element and $iso_{\mathcal{A}}(x) = 1$ otherwise. Furthermore, as every element has exactly two pre-images, $\mathcal{A}$ is simply a 2:1 structure. Thus, by Theorem 1.3.2, since $\mathcal{A}$ has finitely many $\mathbb{Z}$-chains, it is computably categorical.

Figure 3.1: A computably categorical structure $\mathcal{A}$ with infinitely many 1-cycles.



However, if we drop the condition in Theorem 3.1.2 requiring only finitely many K-cycles of each size, then we can no longer guarantee computable categoricity, as the next theorem shows.

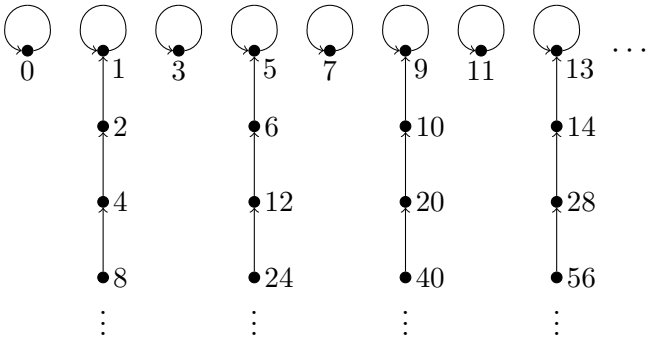
Theorem 3.1.4. *There exists a computable* (2,1):1 *structure* $\mathcal{A} = (A, f)$*, without* $\mathbb{Z}$*-chains and with* $\beta_{\mathcal{A}}$ *and* $iso_{\mathcal{A}}$ *computable, such that* $\mathcal{A}$ *is not computably categorical.*

Proof. We shall first present a computable (2,1):1 structure $\mathcal{A} = (A, f)$ with the desired properties, and then construct a computable isomorphic structure $\mathcal{B}$ that is not computably isomorphic to $\mathcal{A}$.

Let $\mathcal{A} = (A, f)$ be the (2,1):1 structure where $A = \omega$ and $f : A \to A$ is defined as follows:

$$f(x) = \begin{cases} x & \text{if } x = 0 \text{ or } x \text{ is odd,} \\ x - 1 & \text{if } x \equiv 2 \pmod 4, \\ \frac{x}{2} & \text{otherwise.} \end{cases}$$

Figure 3.2: The directed graph of $\mathcal{A}$.



This structure $\mathcal{A}$ is easily seen to be a computable (2,1):1 structure. We can also see that $\beta_{\mathcal{A}}(x) = 2$ if $x \equiv 1 \pmod 4$, and $\beta_{\mathcal{A}}(x) = 1$ otherwise. So the function $\beta_{\mathcal{A}}$ is clearly computable as well. The branch isomorphism function is trivially computable, since $iso_{\mathcal{A}}(x) = 0$ for all $x \in \Lambda_{\mathcal{A}}$. Finally, $\mathcal{A}$ is composed entirely of 1-cycles, and thus contains no $\mathbb{Z}$-chains. Thus, $\mathcal{A}$ has all of the desired properties.

Let $H = \{e : \varphi_e(e) \downarrow\}$ denote the halting set. We build $\mathcal{B} = (B, g)$, a computable isomorphic copy of $\mathcal{A}$, in stages as follows.

Stage 0: Let $B_0 = \{0\}$ and let $g_0(0) = 0$.

Stage s+1: Suppose we are given B_s and g_s from stage s. Find the least $e \leq s$ such that:

(1) $2e \in I_{\mathcal{B}_s}$

(2) $g_s(2e) = 2e$, and

(3) $\varphi_{e,s}(e)\downarrow$

If no such e exists, extend B_s to B_{s+1} and g_s to g_{s+1} by defining $g_{s+1}(2(s+1)) = 2(s+1)$. This adds a 1-cycle to $\mathcal{B}_s$. Also, extend any existing degenerate trees that are attached to a 1-cycle by adding an unused odd number to the end of each one. Go on to the next stage.

If such an e exists, extend B_s to B_{s+1} and g_s to g_{s+1} in the following manner. Attach a degenerate tree of height s, composed entirely of unused odd numbers, to the 1-cycle containing $2e$. Then, extend all existing degenerate trees attached to a 1-cycle by adding an unused odd number to the end of each one. Add a new 1-cycle by defining $g_{s+1}(2(s+1)) = 2(s+1)$. Then go on to the next stage.

Let $B = \bigcup_s B_s$, and $g = \bigcup_s g_s$. It is easy to see that $\mathcal{B}$ is a computable (2,1):1 structure. Also, observe that $x \in \Lambda_{\mathcal{B}}$ if and only if $\frac{x}{2} \in H$. Since H is not computable, both H and $\overline{H}$ are infinite, which means that $\mathcal{B}$ has infinitely many 1-cycles with degenerate trees attached and infinitely many 1-cycles with empty trees attached, as does $\mathcal{A}$. Thus, $\mathcal{A} \cong \mathcal{B}$.

However, $\mathcal{B}$ cannot be computably isomorphic to $\mathcal{A}$. This is because $\Lambda_{\mathcal{A}}$ is computable but $\Lambda_{\mathcal{B}}$ is not, as the computability of $\Lambda_{\mathcal{B}}$ would imply the computability of H. So $\beta_{\mathcal{A}}$ is computable while $\beta_{\mathcal{B}}$ is not. Therefore, by Lemma 3.1.1, $\mathcal{A}$ is not computably categorical. □

Although the (2,1):1 structures in Figures 3.1 and 3.2 both have infinitely many 1-cycles, no $\mathbb{Z}$-chains, and computable branching and branch isomorphism functions, one is computably categorical while the other is not. The key to this discrepancy

is graph embeddability. In Figure 3.1, none of the connected components properly embed into any other connected component, whereas in Figure 3.2, there are *infinitely many* connected components that properly embed into infinitely many other components.

The idea of characterizing computable categoricity via embeddability of components has been used for a number of structures, including linear orders with distinguished functions in [5], and undirected graphs with finite components in [11]. For example, the following result is due to Csima, Khoussainov, and Liu.

Theorem 3.1.5 ([11])**.** *Let $\mathcal{G} = (V, E)$ be a computable strongly locally finite graph with finite connected components $C_0, C_1, ...$, and let $size_{\mathcal{G}} : V \to \omega$ be defined as $size_{\mathcal{G}}(v) = |C(v)|$, where $|X|$ is the cardinality of X and $C(v)$ is the connected component containing v. If $size_{\mathcal{G}}$ is computable, and there are infinitely many i such that the set $\{j \mid C_i$ properly embeds into $C_j\}$ is infinite, then $\mathcal{G}$ is not computably categorical.*

We would like to adapt this theorem to (2,1):1 structures. Unfortunately, since our connected components are not necessarily finite, the graph embedding property in Theorem 3.1.5 does not translate directly to our structures. Instead, we use the following definition.

Definition 3.1.6. Let $\mathcal{A}$ be a (2,1):1 structure with K-cycles C_i and C_j.

(a) C_i is (*properly*) *N-embeddable* into C_j if (C_i, N), the K-cycle C_i truncated at level N, is (properly) embeddable into (C_j, N).

(b) C_i is (*properly*) N^+-*embeddable* into C_j if for all $n \geq N$, (C_i, n) is (properly) embeddable into (C_j, n).

Note that C_i is (properly) embeddable into C_j if and only if C_i is (properly) N^+-embeddable into C_j for some N.

We now give the analog to Theorem 3.1.5 for (2,1):1 structures.

Theorem 3.1.7. *Let $\mathcal{A} = (A, f)$ be a computable* (2,1):1 *structure with $\beta_{\mathcal{A}}$ and $iso_{\mathcal{A}}$ computable and without $\mathbb{Z}$-chains. Let $C_0, C_1, C_2, ...$ be the K-cycles of $\mathcal{A}$. If there are infinitely many i such that for some N, the set $\{j \mid C_i$* is properly N^+-embeddable into $C_j\}$ *is infinite, then $\mathcal{A}$ is not computably categorical.*

Proof. Given an arbitrary (2,1):1 structure $\mathcal{A}$ with the desired properties, we wish to construct a computable copy $\mathcal{B} = (B, g)$ that is not computably isomorphic $\mathcal{A}$. We accomplish this using a standard priority argument similar to the one in the proof of Theorem 3.1.4, ensuring that at the end of the construction, $\mathcal{A} \cong \mathcal{B}$, yet no computable function computes an isomorphism from $\mathcal{A}$ to $\mathcal{B}$.

The main difference between the proof of Theorem 3.1.4/3.1.5 and the one we give here is that now none of the components is necessarily finite. So, as we enumerate and reveal the K-cycles of $\mathcal{A}$ and build isomorphic copies in $\mathcal{B}$, we may never know what the full K-cycles look like, or which components properly embed into other components. Thus, at each stage, we will only consider *truncations* of each K-cycle. If at any point it *appears* that a K-cycle C_i in $\mathcal{A}$ properly embeds into a future K-cycle C_j in $\mathcal{A}$, and some computable function φ_e appears to be a partial isomorphism from C_i to D_i in $\mathcal{B}$, then we will properly extend D_i to ruin the potential computable

isomorphism. However, to ensure that $\mathcal{B}$ is ultimately isomorphic to $\mathcal{A}$, we will only do this finitely many times for each component, and only when we are "free" to do so.

Let $\varphi_0, \varphi_1, \varphi_2, ...$ be an algorithmic enumeration of all computable functions from ω to ω. To construct a computable graph $\mathcal{B} = (B, g)$ that is isomorphic but not computably isomorphic to $\mathcal{A}$, we must satisfy the following requirements:

$$R_e : \varphi_e \text{ is not an isomorphism from } \mathcal{A} \text{ to } \mathcal{B}.$$

We will attempt to satisfy these requirements according to the following priority order:

$$R_0 > R_1 > R_2 > ...,$$

where R_α has a higher priority than R_β if $\alpha < \beta$.

We now give the formal construction.

Stage 0: Let A_0 be the elements of the first K-cycle in $\mathcal{A}$ and let $(C_0, 0)$ be the directed graph associated with $(A_0, 0)$ (see Definition 2.1.7(c)). Set $\mathcal{A}_0 = (C_0, 0)$. Then construct $(D_0, 0)$, an isomorphic copy of $(C_0, 0)$, and set $\mathcal{B}_0 = (D_0, 0)$. Define h_0 to be an isomorphism from $\mathcal{A}_0$ to $\mathcal{B}_0$. Declare C_0 and D_0, the first K-cycles, free for all φ_e.

Stage $s+1$: Suppose that from stage s we have $\mathcal{A}_s = (C_0, s) \sqcup (C_1, s) \sqcup ... \sqcup (C_s, s)$, $\mathcal{B}_s = (D_0, s) \sqcup (D_1, s) \sqcup ... \sqcup (D_s, s)$, and $h_s : \mathcal{A}_s \cong \mathcal{B}_s$. Use $\beta_{\mathcal{A}}$ to reveal a new truncated K-cycle (C_{s+1}, s) in $\mathcal{A}$. Then find the least $e \leq s$ such that for some $i \leq s$:

(1) R_e has not received attention,

(2) $\varphi_{e,s+1}$ is a partial isomorphism from (C_i, s) to (D_i, s),

(3) (C_i, s) properly embeds into (C_{s+1}, s), and

(4) C_i and D_i are "free" for φ_e.

If no such e exists, use $\beta_{\mathcal{A}}$ to extend $\mathcal{A}_s$ to $\mathcal{A}_{s+1}$ by extending (C_i, s) to reveal $(C_i, s+1)$ for $i \leq s+1$. Also, extend $\mathcal{B}_s$ to $\mathcal{B}_{s+1}$ by constructing a truncated K-cycle $(D_{s+1}, s+1) \cong (C_{s+1}, s+1)$, and extending (D_i, s) to $(D_i, s+1)$ for $i \leq s$ such that $(D_i, s+1) \cong (C_i, s+1)$ for all $i \leq s$. Then, extend h_s to an isomorphism $h_{s+1} : \mathcal{A}_{s+1} \to \mathcal{B}_{s+1}$ such that $h_{s+1}((C_i, s+1)) = (D_i, s+1)$ for all $i \leq s+1$. Declare C_{s+1} and D_{s+1} free for all φ_e and go on to stage $s+2$.

Otherwise, R_e *requires attention*, and we do the following. First, properly extend (D_i, s) to create (D_{s+1}, s) such that $(D_{s+1}, s) \cong (C_{s+1}, s)$. Next, build a new copy of (D_i, s) isomorphic to (C_i, s). Then, extend $\mathcal{A}_s$ to $\mathcal{A}_{s+1}$ by using $\beta_{\mathcal{A}}$ to extend (C_i, s) and reveal $(C_i, s+1)$ for $i \leq s+1$, and do the same to $\mathcal{B}_s$ so that $(D_i, s+1) \cong (C_i, s+1)$ for all $i \leq s+1$. Redefine h_s to be an isomorphism $h_{s+1} : \mathcal{A}_{s+1} \to \mathcal{B}_{s+1}$ such that $h_{s+1}((C_i, s+1)) = (D_i, s+1)$ for all $i \leq s+1$. Lastly, declare C_i and D_{s+1} not free for all φ_j such that $j > e$, and declare that φ_e has *received attention*. Move on to the next stage.

This ends the construction. Let $\mathcal{B} = \cup_s \mathcal{B}_s$ and $h = \lim_s h_s$. We must now verify that $\mathcal{B}$ is a computable structure that is indeed isomorphic to $\mathcal{A}$, and that all requirements R_e are satisfied.

Claim 1. The structure $\mathcal{B}$ is computable. Furthermore, the function $h = \lim_s h_s$ exists and is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$.

Proof of Claim 1. Every stage of the construction of $\mathcal{B}$ is easily seen to be com-

putable. Under the assumption that $\mathcal{A}$ and $\beta_{\mathcal{A}}$ are computable, we can effectively determine $\mathcal{A}_s$ at every stage s, and thus effectively construct truncated K-cycles in $\mathcal{B}_s$ isomorphic to those in $\mathcal{A}_s$. Furthermore, once we define g_s on an element in B_s, we never redefine it, as we only extend the components of $\mathcal{B}_s$. Thus, $\mathcal{B}$ is computable.

Let $x \in A$. Since we add a new truncated K-cycle and extend all existing truncated K-cycles in $\mathcal{A}$ at the end of every stage, there must be a stage in the construction at which x appears in some connected component C_i. Let s be the first such stage. Then by the end of stage s, $h_s(x)$ is defined. If no computable function φ_e ever appears to be a partial isomorphism on C_i, then h is never redefined on x, and $h(x) = \lim_s h_s(x)$ exists. If, at some stage $t > s$, R_e requires attention and φ_e does appear to be a partial isomorphism on C_i, then we will only redefine $h(x)$ at most finitely many times after stage t. This is because C_i would not be free for any requirement of lower priority than e, and each requirement of equal or higher priority only requires attention at most once. So again, $h(x) = \lim_s h_s(x)$ exists.

Now let $y \in B$, and suppose that y first appears in the construction at stage s in some connected component D_i. By the end of stage s, $h_s^{-1}(y)$ is defined. Again, if D_i is never used for satisfying a requirement R_e, then $\lim_s h_s^{-1}(y)$ exists. Otherwise, if D_i is used at some stage t for some R_e requiring attention, it can only be used finitely many times after stage t, as D_i is then declared not free for all but finitely many requirements. Thus, $\lim_s h_s^{-1}(y)$ exists once again.

So we have shown that h is defined for all $x \in A$ and h^{-1} is defined for all $y \in B$. Finally, because h_s is built to be an isomorphism from $\mathcal{A}_s$ to $\mathcal{B}_s$ for all stages s, we are guaranteed that h is in fact an isomorphism from $\mathcal{A}$ to $\mathcal{B}$.

Claim 2. Each requirement R_e is satisfied.

Proof of Claim 2. The proof is by induction. Suppose that φ_0 is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$. Let C_i be a K-cycle in $\mathcal{A}$ (appearing at stage i) that is properly N^+-embeddable into infinitely many other K-cycles for some N, and let s be the stage at which $\varphi_{0,s}$ appears to be a partial isomorphism on (C_i, s). At the beginning of some stage $t > max\{i, s, N\}$, we have a truncated K-cycle $(C_t, t-1)$ such that $(C_i, t-1)$ properly embeds into it. Moreover, R_0 is the requirement of highest priority, and thus all K-cycles are free for R_0 at all stages of the construction, including C_i and its isomorphic copy D_i. So at stage t, R_0 would require attention, and thus by the end of the stage, the construction would ensure that φ_0 maps one K-cycle to a non-isomorphic one. Hence, φ_0 is not an isomorphism and R_0 is satisfied.

Suppose that R_i is satisfied for all $i < e$ by some stage s, and that φ_e is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$. Since there are only finitely many K-cycles for which R_e is not free, there exists some K-cycle C_j in $\mathcal{A}$, with C_j and D_j both free for R_e, such that C_j is properly N^+-embeddable into infinitely many other K-cycles for some N. Now, by the same argument as in the base case, there is a stage $t > max\{j, s, N\}$ where R_e would be the least requirement in need of attention, and thus the construction would ruin the potential isomorphism by properly extending $(D_j, t-1)$ into $(D_t, t-1)$. Therefore, R_e is satisfied.

□

Note that Theorem 3.1.7 holds independently of the assumption that the branch isomorphism function of $\mathcal{A}$ is computable. The construction in the proof can be done

effectively if $iso_{\mathcal{A}}$ is assumed to be computable. However, the construction can also be done effectively even if $iso_{\mathcal{A}}$ is not computable; indeed, it only requires that the branching function of $\mathcal{A}$ be computable in order to reveal the truncated K-cycles. Thus, we are free to include or remove this condition from the hypotheses.

3.2 Relative Computable Categoricity

In the previous section, we presented a broad class of computably categorical (2,1):1 structures, as well as a broad class of non-computably categorical (2,1):1 structures. However, there are still many types of (2,1):1 structures that have not been classified. As we mentioned in the introduction, it can be very difficult to give a full characterization of computable categoricity for many classes of structures, especially types of computable graphs. This is because computable categoricity does not have a nice syntactic definition in general.

For this reason, we now turn our attention to relative computable categoricity. By Theorem 1.2.7, we know that relative computable categoricity can be captured syntactically by existential formulas. This is especially helpful for classifying computably categorical structures, as the two notions of categoricity tend to coincide. Unfortunately, computable categoricity does not always imply relative computable categoricity, as Goncharov first observed.

Theorem 3.2.1 ([21])**.** *There exists a computable structure that is computably categorical but not relatively computably categorical.*

To prove this theorem, Goncharov used a sophisticated priority argument to con-

struct an infinite directed graph with finite components that exhibits the desired properties. Downey, Kach, Lempp, and Turetsky later strengthened this theorem in [14], also by building an infinite directed graph with finite components. These results (and others) demonstrate that even for graphs with finite components, computable categoricity can be very tricky to describe.

This revelation suggests that our structures may be similarly ill-behaved. Indeed, the main goal of this section is to establish a version of Theorem 3.2.1 for (2,1):1 structures, supporting our intuition that computable categoricity for such graphs (even without $\mathbb{Z}$-chains) will be very difficult to capture.

Theorem 3.2.2. *There exists a computable* (2,1):1 *structure* $\mathcal{A}$ *that is computably categorical, but not relatively computably categorical.*

Proof. The proof of Theorem 3.2.2 is lengthy and technical. In order to make the proof more readable, we have broken down the important ideas into more manageable, underlined segments.

Overview of the Proof

Our goal is to construct a computable (2,1):1 structure $\mathcal{A} = (A, f)$ that is computably categorical but not relatively so. The structure that we construct will consist entirely of K-cycles. In order to ensure that $\mathcal{A}$ is computably categorical, we must guarantee that if $\mathcal{B}$ is some computable (2,1):1 structure isomorphic to $\mathcal{A}$, then there is a computable isomorphism from $\mathcal{A}$ to $\mathcal{B}$. In order to prevent $\mathcal{A}$ from being relatively computably categorical, we must prevent $\mathcal{A}$ from having a formally c.e. Scott family, as these two properties are equivalent by Theorem 1.2.7.

Before we formally state the requirements that we need to satisfy, we make two combinatorial observations. First, although there are uncountably many (2,1):1 structures, there are only countably many *computable* (2,1):1 structures. This is because every computable (2,1):1 structure is defined via a computable function, of which there are only countably many. The same is true of computable graphs in general. Thus, it makes sense to refer to $\mathcal{B}_i$, the i^{th} (partial) computable graph, where i is a natural number.

Second, there are only countably many possible c.e. Scott families for $\mathcal{A}$. This is due to the fact that the formulas in a c.e. family of existential formulas must be computably enumerable via some partial computable function, of which there are only countably many. Furthermore, each c.e. Scott family can only take in a fixed finite tuple of elements from A as parameters, and clearly there are only countably many such tuples. So, there are only countably many pairs $(\Phi, \vec{c})$ of c.e. families Φ for $\mathcal{A}$ and fixed finite tuples $\vec{c}$ from A. Thus, it makes sense to refer to $(\Phi_j, \vec{c}_j)$, the j^{th} c.e. family and finite tuple pair for $\mathcal{A}$, or simply the j^{th} pair for $\mathcal{A}$. This enumeration will allow us to *diagonalize* against all possible formally c.e. Scott families for $\mathcal{A}$.

Now, in order to prove the theorem, we will construct, in stages, a computable (2,1):1 structure $\mathcal{A} = (A, f)$ that satisfies the following requirements:

$$R_i : \text{If } \mathcal{A} \cong \mathcal{B}_i, \text{ then there is a computable isomorphism } h_i : \mathcal{A} \to \mathcal{B}_i.$$

$$S_j : (\Phi_j, \vec{c}_j) \text{ is not a formally c.e. Scott family for } \mathcal{A}.$$

We will attempt to satisfy the above requirements in the following order of priority:

$$R_0 > S_0 > R_1 > S_1 > ...$$

It should be clear that if R_i is satisfied for all i, then $\mathcal{A}$ is computably categorical, and if S_j is satisfied for all j, then $\mathcal{A}$ is not relatively computably categorical.

The strategy for satisfying the R_i requirements differs greatly from the strategy for satisfying the S_j requirements. Moreover, these strategies will often be in direct conflict with each other, as we will see later. So, we shall first discuss the details of each of these strategies *in isolation*, that is, without regard to other strategies. Then we will address the potential conflicts between strategies and how to resolve them.

Strategy for Satisfying R_i (in Isolation)

Let s be the current stage in the construction, and let $\mathcal{A}_s$ be the finite approximation to $\mathcal{A}$ at stage s. During this stage, R_i will run $\varphi_{i,s}(x)$ on all inputs $x \leq s$. In this way, R_i can computably reveal $\mathcal{B}_{i,s}$, the computable directed graph associated with $\varphi_{i,s}$. If $\mathcal{B}_{i,s}$ does not appear to be a partial (2,1):1 structure isomorphic to $\mathcal{A}_s$, then R_i will do nothing, as computable categoricity is only concerned with computable graphs that are isomorphic to $\mathcal{A}$. Otherwise, R_i will attempt to non-uniformly define a partial isomorphism $h_{i,s}$ from $\mathcal{A}_s$ to $\mathcal{B}_{i,s}$.

Strategy for Satisfying S_j (in Isolation)

Before describing the strategy itself, we first define some special types of K-cycles for convenience. A **Type 0 K-cycle** consists of k cyclic elements, all of which have empty exclusive Trees except for one, whose exclusive Tree is degenerate. A **Type 1 K-cycle** consists of k cyclic elements, all of which have empty exclusive Trees except for one, whose exclusive Tree has a split hair in the first level and only hairs in every level after the first. A **Type 2 K-cycle** consists of k cyclic elements, all of which have empty exclusive Trees except for one, whose exclusive Tree has only split hairs

in the first two levels and only hairs in every level after the second. For all three types, we denote the cyclic element with the non-empty exclusive Tree as the **root vertex**.

Let s be the current stage in the construction, and let $(\Phi_j, \vec{c}_j)_s$ be the stage s approximation of the j^{th} pair for $\mathcal{A}$. Assuming S_j has not taken any action yet, S_j will do the following:

1) Let $K = j + 1$, and construct a Type 0 K-cycle and a Type 1 K-cycle, both truncated at level $s+3$. In doing so, only use elements for the vertices that are strictly larger than those found in the fixed finite tuple $\vec{c}_j$. Let v_0 and v_1 be the root vertices for the Type 0 and Type 1 K-cycles, respectively.

2) For every formula $\varphi(x, \vec{c}_j) \coloneqq (\exists \vec{y})[\theta(x, \vec{y}, \vec{c}_j)]$ in $(\Phi_j, \vec{c}_j)_s$, search through $\mathcal{A}_s$ for a tuple $\vec{a}_j$ such that $\vec{a}_j < s$ (when $\vec{a}_j$ is coded as a natural number) and $\mathcal{A}_s \models \theta(v_1, \vec{a}_j, \vec{c}_j)$.

   If no such formula φ and tuple $\vec{a}_j$ are found, extend the two K-cycles to the $(s+4)^{th}$ level by attaching a hair to every element in the $(s+3)^{th}$ level of $exTree_{\mathcal{A}_s}(v_0)$ and $exTree_{\mathcal{A}_s}(v_1)$. When doing so, be sure to only use elements that are strictly larger than those found in $\vec{c}_j$. Then repeat step 2 at stage $s+1$. If such a formula φ and tuple $\vec{a}_j$ are found, go to step 3.

3) Extend the Type 0 K-cycle containing v_0 to a Type 1 K-cycle by attaching a hair to the element in the first level of $exTree_{\mathcal{A}_s}(v_0)$. Similarly, extend the Type 1 K-cycle containing v_1 to a Type 2 K-cycle by attaching a hair to both elements in the second level of $exTree_{\mathcal{A}_s}(v_1)$. Then, extend the Trees of all elements in

$exTree_{\mathcal{A}_s}(v_0)$ and $exTree_{\mathcal{A}_s}(v_1)$ to the $(s+4)^{th}$ level of the exclusive Trees of v_0 and v_1 by attaching hairs. Again, for all of the new hairs that are attached, be sure to only use elements that are strictly larger than those found in $\vec{c}_j$.

4) At every subsequent stage t, continue to extend the Type 1 and Type 2 K-cycles created at step 3 by attaching one hair to every element in the $(t+3)^{th}$ level of $exTree_{\mathcal{A}_t}(v_0)$ and $exTree_{\mathcal{A}_s}(v_1)$. Once again, only use elements larger than those found in $\vec{c}_j$.

Why the Strategies Work (in Isolation)

It is straightforward to see why the strategy to satisfy R_i should work (in isolation). If $\mathcal{B}_i$ is not total computable, or if $\mathcal{B}_i \not\cong \mathcal{A}$, then $\mathcal{B}_i$ cannot possibly be a counterexample to the computable categoricity of $\mathcal{A}$. If $\mathcal{B}_i$ is a computable graph that is ultimately isomorphic to $\mathcal{A}$, then there should always be a stage at which we can define/extend a partial isomorphism h_i from $\mathcal{A}$ to $\mathcal{B}_i$, for all connected components that appear in $\mathcal{A}$. So ultimately, $h_i : \mathcal{A} \to \mathcal{B}_i$ will be an isomorphism.

To see why the strategy to satisfy S_j works (in isolation), observe that there are two possible outcomes for each S_j-strategy: either it waits forever at step 2 for a formula φ and a tuple $\vec{a}_j$, or it reaches step 3.

If the S_j-strategy never passes step 2, then it never found an existential formula $\varphi(x, \vec{c}_j)$ in $(\Phi_j, \vec{c}_j)$ such that v_1 satisfies it. So, not every tuple in $\mathcal{A}$ satisfies some formula in $(\Phi_j, \vec{c}_j)$, since the 1-tuple (v_1) does not satisfy any formula in $(\Phi_j, \vec{c}_j)$. Therefore, the j^{th} pair $(\Phi_j, \vec{c}_j)$ cannot be a formally c.e. Scott family for $\mathcal{A}$, as the first condition from Definition 1.2.6 would be violated.

If the S_j-strategy reaches the end of step 3 at some stage s, then the strategy found a formula $\varphi(x, \vec{c}_j)$ in $(\Phi_j, \vec{c}_j)$ such that $\mathcal{A}_s \models \varphi(v_1, \vec{c}_j)$. Thus, we have the following:

$$\mathcal{A}_s \models \varphi(v_1, \vec{c}_j) \implies \mathcal{A}_{s+1} \models \varphi(v_1, \vec{c}_j) \implies \mathcal{A} \models \varphi(v_1, \vec{c}_j) \tag{1}$$

The first implication in (1) is due to the fact that $\varphi(v_1, \vec{c}_j)$ is an existential sentence and that the K-cycle containing v_1 only contains elements strictly larger than those found in $\vec{c}_j$. So, $\varphi(v_1, \vec{c}_j)$ can only assert things about the existence of non-specific elements in the K-cycle containing v_1, and the way in which those non-specific elements are attached to each other and to v_1. Thus, if $\varphi(v_1, \vec{c}_j)$ is true in $\mathcal{A}_s$ when v_1 is part of Type 1 K-cycle, then it must also be true in $\mathcal{A}_{s+1}$ when the component containing v_1 is extended to a Type 2 K-cycle.

The second implication in (1) is due to the fact that once we modify the component containing v_1 to be a Type 2 K-cycle at stage s, the only further changes we will make to that component in future stages is to continue extending the exclusive Tree of v_1 (in order to ensure that it will ultimately be a full K-cycle). Again, since $\varphi(v_1, \vec{c}_j)$ is an existential sentence and the component containing v_1 never uses any elements from $\vec{c}_j$, $\varphi(v_1, \vec{c}_j)$ remains true as we repeatedly extend the component at step 4 of the strategy. Thus, for all stages $t > s$, $\mathcal{A}_t \models \varphi(v_1, \vec{c}_j)$, and therefore $\mathcal{A} \models \varphi(v_1, \vec{c}_j)$.

However, we also have the following:

$$\mathcal{A}_s \models \varphi(v_1, \vec{c}_j) \implies \mathcal{A}_{s+1} \models \varphi(v_0, \vec{c}_j) \implies \mathcal{A} \models \varphi(v_0, \vec{c}_j) \tag{2}$$

The first implication in (2) stems from the fact that the K-cycle containing v_1 at stage s is isomorphic to the K-cycle containing v_0 at stage $s+1$, since both K-cycles are of Type 1, and that neither of these K-cycles ever uses any elements from $\vec{c}_j$. Thus, by a similar argument as before, if $\varphi(v_1, \vec{c}_j)$ is true at stage s, then $\varphi(v_0, \vec{c}_j)$ must be true at stage $s+1$. The second implication in (2) holds for the same reason that the second implication in (1) is true.

From (1) and (2), we have that $\mathcal{A} \models \varphi(v_0, \vec{c}_j)$ and $\mathcal{A} \models \varphi(v_1, \vec{c}_j)$. Thus, v_0 and v_1 both satisfy the same formula $\varphi(x, \vec{c}_j)$ in $(\Phi_j, \vec{c}_j)$. However, v_0 and v_1 are clearly not automorphic in $\mathcal{A}$, as v_0 is on a Type 1 K-cycle while v_1 is on a Type 2 K-cycle. Therefore, $(\Phi_j, \vec{c}_j)$ cannot be a formally c.e. Scott family for $\mathcal{A}$, since the second condition from Definition 1.2.6 would not hold for $(\Phi_j, \vec{c}_j)$.

In each of the two possible outcomes for the S_j-strategy, the requirement S_j is satisfied. Furthermore, regardless of the outcome, the strategy makes sure to continue extending the two K-cycles that it builds so that the structure created is ultimately a (2,1):1 structure, and it never redefines the function f on any element in A once it is defined. Therefore, if all R_i and S_j-strategies are ultimately successful, we will have constructed a computable (2,1):1 structure that is computably categorical, but not relatively so.

The Conflicts Between R_i-Strategies and S_j-Strategies

Unfortunately, the R_i-strategy and the S_j-strategy are almost directly opposed to each other. The S_j-strategy seeks to build and extend K-cycles for the purpose of diagonalizing against potential c.e. Scott families, while the R_i-strategy would ideally like to prevent existing K-cycles from being altered, as this may ruin a partial

computable isomorphism between $\mathcal{A}$ and B_i.

For example, suppose at the beginning of some stage s, the R_0-strategy is able to build a partial isomorphism $h_{0,s} : \mathcal{A}_s \to \mathcal{B}_{0,s}$. If later on during the stage, a lower-priority strategy, say S_0, has the opportunity to diagonalize against $(\Phi_0, \vec{c}_0)$, it will properly extend the two components it has so-far built, call them $C_{0,s}$ and $C_{1,s}$, into Type 1 and Type 2 K-cycles, respectively, as described above in step 3 of the S_j-strategy. Thus, $C_{0,s+1}$ and $C_{1,s+1}$ would no longer be isomorphic to $D_{0,s} = h_{0,s}(C_{0,s})$ and $D_{1,s} = h_{0,s}(C_{1,s})$, respectively.

Now, if $D_{0,s}$ and $D_{1,s}$ never grow in $\mathcal{B}_0$ to become Type 1 and Type 2 K-cycles, then R_0 will still be satisfied, as $\mathcal{A}$ cannot possibly be isomorphic to $\mathcal{B}_0$ anymore. However, if $D_{0,s}$ eventually grows to become a Type 2 K-cycle and $D_{1,s}$ remains a Type 1 K-cycle, then C_0 and C_1 will be isomorphic to D_0 and D_1, but $h_{0,s}$ will no longer be a computable isomorphism from $\mathcal{A}$ to $\mathcal{B}_0$. This is because $h_{0,s}$ will still map C_0 (now a Type 1 K-cycle) to D_0 (now a Type 2 K-cycle), and C_1 (now a Type 2 K-cycle) to D_1 (now a Type 1 K-cycle). So $h_{0,s}$ would need to be redefined on C_0 and D_0 to still be an isomorphism. Furthermore, because there are infinitely many lower-priority S_j requirements, there are sufficiently many opportunities for the S_j-strategies to defeat not only the computable isomorphism $h_{0,s}$, but *all possible computable isomorphisms* from $\mathcal{A}$ to $\mathcal{B}_0$. Thus, if left unchecked, the S_j-strategies could potentially prevent R_0 from ever being satisfied.

On the other hand, if a lower-priority S_j-strategy has the opportunity to extend its components and diagonalize against $(\Phi_j, \vec{c}_j)$, but is stalled indefinitely by R_0 hoping to maintain its computable isomorphism h_0, then $(\Phi_j, \vec{c}_j)$ may very well end up being

a formally c.e. Scott family for $\mathcal{A}$. This would prevent S_j from ever being satisfied.

Slowing Down the S_j-Strategy and Guessing R_i Outcomes

Before we give the actual proof of Theorem 3.2.2, we must address the two conflicts described in the previous segment. We do so here.

The first conflict mentioned above can be avoided if we carefully slow down the steps in the S_j-strategy. In particular, an S_j-strategy can avoid diagonalizing against a partial computable isomorphism h_i built by a higher-priority R_i if we slow down step 3 of the strategy.

If at any stage s, the S_j strategy reaches step 3, it will only extend the Type 1 K-cycle containing v_1 into a Type 2 K-cycle, and will leave the Type 0 K-cycle containing v_0 as a Type 0 K-cycle. Then, for all higher-priority R_i-strategies, the S_j-strategy will consider those that have successfully built a partial isomorphism $h_{i,s}$ from $\mathcal{A}_s$ to $\mathcal{B}_{i,s}$. For all such B_i, the S_j-strategy will wait until a Type 2 K-cycle, which looks like the one containing v_1, appears in B_i. If and when this happens, the S_j-strategy will finally complete its diagonalization effort and extend the Type 0 K-cycle containing v_0 into a Type 1 K-cycle.

By slowing down the construction in this way, the issue of diagonalizing against a potential computable isomorphism cannot occur. Suppose at some stage s, an R_i-strategy has successfully built a computable isomorphism $h_{i,s} : \mathcal{A}_s \to \mathcal{B}_{i,s}$. Now, if a lower-priority S_j-strategy sees an opportunity to advance to step 3, it will only extend the Type 1 K-cycle containing v_1, call it $C_{1,s}$, into a Type 2 K-cycle, and leave the Type 0 K-cycle containing v_0, call it $C_{0,s}$, as is.

After this point, if $\mathcal{B}_i$ is ultimately isomorphic to $\mathcal{A}$, a unique Type 2 K-cycle (with

exactly K cyclic elements) must appear in $\mathcal{B}_i$. Moreover, this unique Type 2 K-cycle must either be an extension of $D_{0,s} = h_{0,s}(C_{0,s})$ or $D_{1,s} = h_{0,s}(C_{1,s})$. Whichever it is, the R_i-strategy will know exactly how to define/extend the partial isomorphism h_i accordingly. Later, if and when the S_j-strategy is ready to complete step 3 and extend C_0 into a Type 1 K-cycle, a unique Type 1 K-cycle must now appear in $\mathcal{B}_i$, if it has not already. When it does, the R_i-strategy will once again know exactly how to define and extend the partial isomorphism h_i. Therefore, since at every point during the S_j-strategy, R_i can computably determine how to define h_i, the computable isomorphism will remain intact on C_0 and C_1.

It may appear that the slowing-down technique does not actually solve our problem, as we may still need to redefine h_i after S_j reaches Step 3. However, we can sidestep this issue if we think of prior failed attempts to define h_i as "false paths", and only the final, correct version of h_i as the "true path". We will come back to this point later in the proof when we define the *priority tree*.

Here is the slowed-down version of step 3 of the S_j-strategy at stage s. Steps 1, 2, and 4 remain the same. At all steps, whenever adding new elements is required, we always use elements strictly larger than those found in $\vec{c}_j$.

3a) Extend the Type 1 K-cycle containing v_1 to a Type 2 K-cycle by attaching a hair to both elements in the second level of $exTree_{\mathcal{A}_s}(v_1)$. Then, extend the Trees of all elements in $exTree_{\mathcal{A}_s}(v_1)$ to the $(s+4)^{th}$ level of the exclusive Trees of v_1 by attaching hairs.

3b) For all $i \leq j$ such that $\mathcal{A} \cong \mathcal{B}_i$, wait for a Type 2 K-cycle (with exactly K cyclic

elements), truncated at some arbitrary level, to appear in $\mathcal{B}_i$. While waiting, continuously extend the K-cycles containing v_0 and v_1 by adding hairs to the ends.

3c) Extend the Type 0 K-cycle containing v_0 to a Type 1 K-cycle by attaching a hair to the element in the first level of $exTree_{\mathcal{A}_s}(v_0)$. Then, extend the Trees of all elements in $exTree_{\mathcal{A}_s}(v_0)$ by attaching hairs as necessary.

Although this revised S_j-strategy solves the first conflict mentioned in the previous underlined section, it still does not address the second conflict. That is, for a given i, we cannot computably determine if a Type 2 K-cycle will ever appear in $\mathcal{B}_i$, much less if $\mathcal{A}$ will eventually be isomorphic to $\mathcal{B}_i$. In fact, for an arbitrary i, it is likely that the corresponding directed graph $\mathcal{B}_i$ will not even end up being a (2,1):1 structure. Thus, even with the revised construction, it is possible that the S_j-strategy will now become stuck waiting at step 3b, once again unable to complete its diagonalization against $(\Phi_j, \vec{c}_j)$.

The solution to this problem is to have the S_j-strategy "guess" the outcomes of the higher-priority R_i-strategies. Specifically, if at some stage s, an S_j-strategy has completed step 3a, it will check for each $i \leq j$ whether the R_i-strategy *believes* that $\mathcal{A} \cong \mathcal{B}_i$. That is, during stage s, the R_i-strategies will have an outcome stating that at the moment, $\mathcal{A}_s \cong \mathcal{B}_{i,s}$, and the S_j-strategy will check for those R_i-strategies with this outcome.

For all $i \leq j$ such that the R_i-strategy believes that $\mathcal{A} \cong \mathcal{B}_i$, if the S_j-strategy sees a Type 2 K-cycle in every $\mathcal{B}_{i,s}$ that looks like the one it created, it will proceed to

step 3c, as we will then be in the scenario described above. However, if there is some $i \leq j$ such that the R_i-strategy believes that $\mathcal{A} \cong \mathcal{B}_i$ but $\mathcal{B}_{i,s}$ does not have a Type 2 K-cycle like the one created by S_j, the S_j-strategy will continue waiting at step 3b for a stage t when the Type 2 K-cycle it is waiting for appears in $\mathcal{B}_{i,t}$. (To complete the description of the strategy, we note that each S_j-strategy only considers information given from higher-priority R_i-strategies, and will proceed to step 3c regardless of the state of lower-priority R_i-strategies.)

With this modified guessing technique, an S_j-strategy will never wait endlessly at step 3b, *provided that it makes the right guesses.* After the S_j-strategy creates its Type 2 K-cycle at stage s, if the S_j-strategy correctly guesses that, for some $i \leq j$, $\mathcal{A} \cong \mathcal{B}_i$, it will only have to wait until some finite stage t when a Type 2 K-cycle like the one it created appears in $\mathcal{B}_{i,t}$. There are only finitely many $i \leq j$ for which the S_j-strategy will guess that $\mathcal{A} \cong \mathcal{B}_i$ at stage s, and each $\mathcal{B}_i$ will only have to wait finitely many stages for the Type 2 K-cycle to appear, assuming all of the S_j-strategy's guesses were correct. Thus, the S_j-strategy will only wait finitely many stages before it can proceed to step 3c.

Of course, we cannot *computably* know which guesses are correct. However, all that matters is that there *is* a correct guess for the outcome of each R_i-strategy.

The Priority Tree Construction, and the Full R_i and S_j-Strategy

We are now ready to give the full proof of Theorem 3.2.2. In this segment, we present the construction of the desired structure $\mathcal{A}$, based on the refined strategies described in the previous segment.

Since the actions and outcomes of each strategy depend on the actions and out-

comes of other strategies, it is best to represent the strategies on a tree, similar to the trees used to analyze moves in a game. We refer to this tree as the *priority tree*.

Before we construct the priority tree, we establish the set of possible outcomes for our strategies, and other preliminaries. Each R_i-strategy will have an infinite set of outcomes, $\{0, 1, 2, ..., k, ..., \infty\}$. At the end of a stage s, an outcome of ∞ signifies that the R_i-strategy believes that $\mathcal{A} \cong \mathcal{B}_i$, whereas an outcome of m (for each $m \in \omega$) signifies that the R_i-strategy has had outcome ∞ at m stages before stage s. Each S_j-strategy will have a set of two outcomes, $\{0, 1\}$. At the end of a stage s, an outcome of 0 indicates that the S_j-strategy is still waiting at step 2 of its procedure, and an outcome of 1 indicates that the S_j-strategy has reached step 3 of its procedure. Let $\Omega = \{0, 1, 2, ..., m, ..., \infty\}$ be the set of all possible outcomes of a strategy. We put an ordering on Ω, denoted by $<_\Omega$, such that $\infty <_\Omega \cdots <_\Omega m <_\Omega \cdots <_\Omega 1 <_\Omega 0$.

Next, let $\Omega^{<\omega}$ denote the set of all finite sequences of elements from Ω, and let Ω^ω denote the set of all infinite sequences of elements from Ω. For all $\alpha \in \Omega^{<\omega}$, let $|\alpha|$ denote the length of the sequence α. Then, for each $n \in \mathbb{N}$ such that $n < |\alpha|$, define $\alpha(n)$ to be the n^{th} term in the finite sequence α. Similarly, if $a \in \Omega^\omega$ and $n \in \mathbb{N}$, define $a(n)$ to be the n^{th} term in the infinite sequence a.

Now, define the priority tree T as follows:

$$T = \{\alpha \in \Omega^{<\omega} : (\forall e)[2e + 1 < |\alpha| \implies \alpha(2e + 1) \in \{0, 1\}]\}.$$

In words, the priority tree T is the set of all finite sequences of outcomes from Ω such that the odd-numbered terms of the sequence are either 0 or 1. This definition is motivated by the fact that we wish to assign, in alternating order, each R_i and S_j requirement to one level of the tree, where level n of T is just the set of all $\alpha \in T$ such

that $|\alpha| = n$. Each requirement R_i is assigned to level $2i$ of T, and each requirement S_j is assigned to level $2j + 1$ of T, to capture the priority order of the requirements. The possible outcomes of each strategy are labeled on the nodes one level below where the requirement was assigned; thus the outcomes of R_i-strategies are found on odd levels, while the outcomes of S_j-strategies are found on even levels. Since the root node at level 0 corresponds to the empty sequence, and the first number of a sequence α is given by $\alpha(0)$, the odd-numbered terms of a sequence correspond to even levels of T, which correspond to outcomes of S_j-strategies. Therefore, our definition of T requires that $\alpha(2e + 1) \in \{0, 1\}$.

Furthermore, let $[T]$ be the set of all *paths* through T. That is,

$$[T] = \{a \in \Omega^\omega : (\forall n)[a \upharpoonright n \in T]\},$$

where $a \upharpoonright n$ denotes the restriction of a to its first n terms. Thus, $[T]$ is the set of all infinite sequences of outcomes from Ω such that the odd-numbered terms of the sequence are either 0 or 1.

Let $\alpha, \beta \in T$. We write $\alpha \subseteq \beta$ if β extends α, and $\alpha \subset \beta$ if β properly extends α. We say that α and β are *incomparable* if neither $\alpha \subseteq \beta$ nor $\beta \subseteq \alpha$. Also, let $\alpha^\frown\beta$ denote the concatenation of the strings α and β. Then we say that α is *to the left of* β, written as $\alpha <_L \beta$, if

$$(\exists p, q \in \Omega)(\exists \gamma \in T)[p <_\Omega q \wedge \gamma^\frown p \subseteq \alpha \wedge \gamma^\frown q \subseteq \beta].$$

Furthermore, we write $\alpha \leq \beta$ if $\alpha <_L \beta$ or $\alpha \subseteq \beta$, and we write $\alpha < \beta$ if $\alpha \leq \beta$ and $\alpha \neq \beta$. (It is straightforward to check that $\leq$ is a linear ordering on T.) Finally, for $a, b \in [T]$, we say that a is *to the left of* b, also written as $a <_L b$, if

$$(\exists n)[a \upharpoonright n <_L b \upharpoonright n].$$

As we have seen before, each $\alpha \in T$ is associated with a node at level $|\alpha|$ on the priority tree, which is in turn associated with a strategy to satisfy the requirement assigned to that level. Let $\rho \in T$ be used to denote R_i-strategies (nodes on even levels of T), and let $\sigma \in T$ be used to denote S_j-strategies (nodes on odd levels of T). At any given stage s of the construction, each strategy $\alpha \in T$ with $|\alpha| \leq s$ will have access to all outcomes of prior strategies (i.e., $\beta \subseteq \alpha$) as well as outcomes of future strategies of length at most s. In this way, the R_i-strategies and S_j-strategies no longer act in isolation of each other.

We now give the full strategies for satisfying the R_i and S_j requirements, stated in the terminology of the priority tree.

The Full Strategy for Satisfying R_i

Let s be the current stage of the construction, and let $\mathcal{A}_s$ and $\mathcal{B}_{i,s}$ be the stage s approximation of $\mathcal{A}$ and $\mathcal{B}_i$, respectively, with $h_{i,s}$ a partial isomorphism from $\mathcal{A}_s$ to $\mathcal{B}_{i,s}$. Let $\rho \in T$ with $|\rho| \leq s$ be an R_i-strategy, that is, a node on the $(2i)^{th}$ level of the priority tree. Let m be the number of stages less than s at which ρ had outcome ∞.

The R_i-strategy ρ will consider three types of root vertices in $\mathcal{A}_s$:

1) Root vertices created by S_j-strategies $\sigma \subset \rho$ such that $\sigma^\frown 0 \subseteq \rho$.

2) Root vertices created by S_j-strategies $\sigma \subset \rho$ such that $\sigma^\frown 1 \subseteq \rho$ and σ has reached step 4 of its strategy.

3) Root vertices created by S_j-strategies $\sigma \not\subset \rho$ such that $|\sigma| \leq s$ and σ is incomparable with $\rho^\frown n$ for all $n \in \omega$.

For each root vertex v in $\mathcal{A}_s$ that ρ is considering, we check if $h_{i,s}$ is defined on v. If $h_{i,s}(v)$ is not yet defined, we search $\mathcal{B}_{i,s}$ for a root vertex u with a truncated K-cycle that appears identical to the one containing v. If we find one, we define $h_{i,s}(v) \coloneqq u$, then extend $h_{i,s}$ to an isomorphism of the truncated K-cycles containing v and u; otherwise, we do nothing. If $h_{i,s}(v)$ is already defined, and the truncated K-cycle containing v still appears identical to the component of $h_{i,s}(v)$ in $\mathcal{B}_{i,s}$, we extend $h_{i,s}$ to an isomorphism of the truncated K-cycles containing v and $h_{i,s}(v)$, if it is not already; otherwise, we do nothing.

After this action, if for every vertex v that ρ is considering, $h_{i,s}$ is defined on v and is an isomorphism of the truncated K-cycles containing v and $h_{i,s}(v)$, then we extend $h_{i,s}$ to $h_{i,s+1}$ as prescribed above, and declare that ρ has outcome ∞ at the end of stage s. Otherwise, ρ has outcome m at the end of stage s.

The Full Strategy for Satisfying S_j

Let s be the current stage in the construction, and let $\mathcal{A}_s$, $\mathcal{B}_{i,s}$, and $(\Phi_j, \vec{c}_j)_s$ be the stage s approximation of $\mathcal{A}$, $\mathcal{B}_i$, and the j^{th} pair for $\mathcal{A}$, respectively. Let $\sigma \in T$ with $|\sigma| \leq s$ be an S_j-strategy, that is, a node on the $(2j+1)^{th}$ level of the priority tree. Assuming σ has not taken any action yet, σ will perform the following steps, always using numbers larger than those found in $\vec{c}_j$ whenever adding new vertices is necessary.

1) Let $K = j + 1$, and construct a Type 0 K-cycle and a Type 1 K-cycle, both truncated at level $s + 3$. Let v_0 and v_1 be the root vertices for the Type 0 and Type 1 K-cycles, respectively.

2) For every formula $\varphi(x, \vec{c}_j) \coloneqq (\exists \vec{y})[\theta(x, \vec{y}, \vec{c}_j)]$ in $(\Phi_j, \vec{c}_j)_s$, search through $\mathcal{A}_s$ for a tuple $\vec{a}_j$ such that $\vec{a}_j < s$ (when $\vec{a}_j$ is coded as a natural number) and $\mathcal{A}_s \models \theta(v_1, \vec{a}_j, \vec{c}_j)$.

If no such formula φ and tuple $\vec{a}_j$ are found, extend the two K-cycles to the $(s+4)^{th}$ level by attaching a hair to every element in the $(s+3)^{th}$ level of $exTree_{\mathcal{A}_s}(v_0)$ and $exTree_{\mathcal{A}_s}(v_1)$. Declare that σ has outcome 0. Then repeat step 2 at stage $s+1$. If such a formula φ and tuple $\vec{a}_j$ are found, declare that σ has outcome 1, then go to step 3a.

3a) Extend the Type 1 K-cycle containing v_1 to a Type 2 K-cycle by attaching a hair to both elements in the second level of $exTree_{\mathcal{A}_s}(v_1)$. Then, extend the Trees of all elements in $exTree_{\mathcal{A}_s}(v_1)$ to the $(s+4)^{th}$ level of the exclusive Trees of v_1 by attaching hairs.

3b) For all $i \leq j$ such that ρ guesses that $\mathcal{A} \cong \mathcal{B}_i$, wait for a stage when a Type 2 K-cycle (with exactly K cyclic elements), truncated at some arbitrary level, appears in every $\mathcal{B}_i$. While waiting, continuously extend the K-cycles containing v_0 and v_1 by adding hairs to the ends.

3c) Extend the Type 0 K-cycle containing v_0 to a Type 1 K-cycle by attaching a hair to the element in the first level of $exTree_{\mathcal{A}_s}(v_0)$. Then, extend the Trees of all elements in $exTree_{\mathcal{A}_s}(v_0)$ by attaching hairs as necessary.

4) At every subsequent stage t, continue to extend the Type 1 and Type 2 K-cycles created during step 3 by attaching one hair to every element in the $(t+3)^{th}$

level of $exTree_{\mathcal{A}_t}(v_0)$ and $exTree_{\mathcal{A}_t}(v_1)$.

As we mentioned before, we cannot computably know which guesses made by a σ-strategy at step 3b are correct. More generally, we cannot computably know which outcomes a given strategy will have, and thus which nodes will be visited on the priority tree during the construction. However, we can *approximate* which route through T the construction will take, and ultimately use this approximation to show that there is an infinite path in $[T]$, called the *true path*, on which all requirements are satisfied.

Let s be the current stage, and let $n \leq s$. Define $\delta_s \in T$, the stage s approximation to the true path, as follows. For $n = 2e$,

$$\delta_s(n) = \begin{cases} \infty & \text{if the } R_e\text{-strategy has outcome } \infty \text{ at the end of stage } s, \\ m & \text{if the } R_e\text{-strategy has outcome } m \text{ at the end of stage } s. \end{cases}$$

For $n = 2e + 1$,

$$\delta_s(n) = \begin{cases} 0 & \text{if the } S_e\text{-strategy has outcome 0 at the end of stage } s, \\ 1 & \text{if the } S_e\text{-strategy has outcome 1 at the end of stage } s. \end{cases}$$

At a stage s in the construction, for all nodes $\alpha \in T$ with $|\alpha| \leq s$ that are visited by δ_s, we let each α carry out its strategy in order of priority. As the construction progresses, $\delta_s(n)$ may obviously change for certain values of n; in fact, it may change infinitely often. However, it is important to note that if δ_s moves to the left of a node

$\alpha \in T$ that has already been visited, α cannot be visited again at a later stage t by δ_t.

This can be seen by considering the ordering we defined on T, the steps of the individual R_i and S_j-strategies, and the definition of δ_s. If $n = 2e+1$ and $\delta_s(n) = 1$, then the S_e-strategy reached step 3a at stage s and cannot revert back to step 2 at a later stage t. So $\delta_t(n)$ cannot be 0 at any stage $t > s$. If $n = 2e$ and $\delta_s(n) = m$ for some $m \in \omega$, then the R_e-strategy believed that $\mathcal{A} \cong \mathcal{B}_i$ at least m times by stage s, and therefore cannot later believe that $\mathcal{A} \cong \mathcal{B}_i$ less than m times. Thus, $\delta_t(n) \leq_\Omega m$ for all stages $t > s$.

We now define the *true path* p *through* $[T]$ in the following manner. For $n = 2e$,

$$p(n) = \begin{cases} \infty & \text{if } (\forall s)(\exists t)[t > s \wedge \delta_t(n) = \infty], \\ m & \text{if } (\exists s)(\forall t)[t > s \implies \delta_t(n) = m]. \end{cases}$$

For $n = 2e + 1$,

$$p(n) = \begin{cases} 0 & \text{if } (\forall s)[\delta_s(n) = 0], \\ 1 & \text{if } (\exists s)[\delta_s(n) = 1]. \end{cases}$$

Equivalently, we can define $p(n) \coloneqq \liminf_s \delta_s(n)$, under the $\leq_\Omega$ ordering. We may also think of p as the leftmost path through $[T]$ that is visited infinitely often by the sequence $\{\delta_s\}_{s\in\omega}$.

This ends the priority tree construction. Let $\mathcal{A}$ be the structure built along the true path p. We must now verify that p, as defined above, exists, that the structure $\mathcal{A}$ created by p is a computable (2,1):1 structure, that each R_i is satisfied, and that

each S_j is satisfied.

Verification

In this final segment of the proof, we verify the correctness of the presented construction in four claims.

Claim 1. The true path $p(n)$ exists.

Proof of Claim 1. Let $n = 2e$. If the R_e-strategy has outcome ∞ infinitely often, then $p(n) = \infty$. Otherwise, it has outcome ∞ at only finitely many stages. Let m be the number of stages at which the R_e-strategy has outcome ∞. Then at any stage after the R_e-strategy has outcome m, it cannot have any outcome other than m. Thus, $p(n) = m$.

Let $n = 2e+1$. If the S_e-strategy never has outcome 1, then $p(n) = 0$. Otherwise, the S_e-strategy has outcome 1 at some stage s, and it can never have outcome 0 again at any stage after s. Thus, $p(n) = 1$.

Therefore, p is defined for all $n \in \mathbb{N}$.

Claim 2. The structure $\mathcal{A}$ built by the true path is a computable (2,1):1 structure.

Proof of Claim 2. The computability of $\mathcal{A}$ follows directly from the construction. In particular, once we establish a directed edge between two vertices, we never redefine the edge relation between those vertices; we only extend the K-cycles by adding new directed edges.

It is also immediate from the construction that $\mathcal{A}$ is a (2,1):1 structure. Each S_j-strategy introduces two new truncated K-cycles, and continues to extend them at all future stages of the construction regardless of the outcome of S_j at any stage. Thus, $\mathcal{A}$ is composed of infinitely many infinite K-cycles.

Claim 3. Each requirement R_i is satisfied along the true path.

Proof of Claim 3. Suppose that $\mathcal{A} \cong \mathcal{B}_i$ for some index i. Let ρ be the R_i-strategy on the true path. We wish to show that ρ builds a computable isomorphism $h_i : \mathcal{A} \to \mathcal{B}_i$. Note that by assumption, for every K-cycle in $\mathcal{A}$, $\mathcal{B}_i$ contains exactly one K-cycle isomorphic to it.

Let $\sigma \subset \rho$ be an S_j-strategy along the true path such that $\sigma^\frown 0 \subseteq \rho$. Then at some stage s, σ creates a truncated Type 0 K-cycle and Type 1 K-cycle, after which ρ will begin considering them. Because $\sigma^\frown 0$ is on the true path, σ has final outcome 0 and will never extend its K-cycles into Type 1 and Type 2 K-cycles; it will only extend the existing K-cycles by adding levels to them. Thus, once truncated K-cycles isomorphic to the ones created by σ appear in $\mathcal{B}_i$ (which must happen by assumption), ρ will define h_i on these truncated components and continue extending the partial isomorphism at future stages as the K-cycles grow. Furthermore, h_i will never have to be redefined on these components. Thus, h_i is ultimately a computable isomorphism on these K-cycles.

Let $\sigma \subset \rho$ be an S_j-strategy along the true path such that $\sigma^\frown 1 \subseteq \rho$. Since $\sigma^\frown 1$ is on the true path, σ will eventually reach step 4 of its strategy (see the proof of Claim 4), at which point it has built truncated Type 1 and Type 2 K-cycles. Remember that ρ only considers the root vertices built by strategies σ with $\sigma^\frown 1 \subseteq \rho$ if σ reaches step 4 of its strategy. Thus, ρ only considers the components built by σ *after* they have been extended into Type 1 and Type 2 K-cycles. After ρ begins considering them, it waits for isomorphic truncated Type 1 and Type 2 K-cycles to appear in $\mathcal{B}_i$, which must happen by assumption. When they do, ρ will define h_i on these truncated

components and continue extending h_i as in the last case, never needing to redefine the partial isomorphism. Thus, h_i is ultimately a computable isomorphism on these K-cycles.

Let σ be an S_j-strategy incomparable with ρ. Since ρ is on the true path and σ is not, once ρ begins considering the components created by σ, the construction will never visit σ again. Thus, after ρ considers them, the truncated K-cycles created by σ can never extend into different types of K-cycles; they can only be extended by levels. Therefore, by the same argument as above, ρ defines a computable isomorphism h_i on these components as well.

Let $\sigma \supset \rho$ be an S_j-strategy such that $\sigma \supseteq \rho^\frown\infty$. If $\sigma^\frown 0$ is on the true path, then σ ultimately builds a Type 0 K-cycle and a Type 1 K-cycle, which ρ would begin considering after they are built. By the same argument as in the first case, ρ will ultimately define a computable isomorphism on these K-cycles.

Now suppose $\sigma^\frown 1$ is on the true path. Let C_0 be its Type 0 K-cycle and let C_1 be its Type 1 K-cycle. If ρ is able to find truncated copies of C_0 and C_1 in $\mathcal{B}_i$ before σ extends C_1 into a Type 2 K-cycle, then ρ will define a partial isomorphism h_i on the truncated versions of C_0 and C_1. Let D_0 and D_1 in $\mathcal{B}_i$ be the images under h_i of C_0 and C_1, respectively. Once σ extends C_1 into a Type 2 K-cycle, by construction, ρ will never have outcome ∞ again unless D_1 later grows into a Type 2 K-cycle. Since ρ has final outcome ∞ in this scenario, D_1 must eventually become a Type 2 K-cycle isomorphic to the one created by σ. When it does, ρ can extend its isomorphism from C_1 to D_1, and will never have to redefine h_i on that K-cycle.

Furthermore, since σ is on the true path, it will eventually reach step 3c of its

strategy, and extend C_0 into a Type 1 K-cycle. After this step, ρ will not have outcome ∞ again unless D_0 later grows into a Type 1 K-cycle. Thus, D_0 will eventually grow into a Type 1 K-cycle isomorphic to the one created by σ. Once again, ρ will then be able to extend h_i to be an isomorphism on the newly-created Type 1 K-cycles. Therefore, h_i will be a computable isomorphism on both of the K-cycles created by σ.

If σ extends C_1 into a Type 2 K-cycle before ρ can define an isomorphism on C_0 and C_1, then ρ will never have outcome ∞ again unless a Type 2 K-cycle D_1 isomorphic to C_1 appears in $\mathcal{B}_i$. Once D_1 appears, ρ can map C_1 to D_1. Since $\mathcal{A} \cong \mathcal{B}_i$, and C_1 is the only K-cycle of its type and size in $\mathcal{A}$, D_1 is guaranteed to be the unique and correct image of C_1. By the same argument as in the previous paragraph, σ will then extend C_0 into a Type 1 K-cycle at some later stage, and some copy of C_0, call it D_0, is guaranteed to appear in $\mathcal{B}_i$. When it does, ρ can map C_0 to D_0. Moreover, D_0 is the correct and unique image of C_0 in $\mathcal{B}_i$. Therefore, h_i is once again a computable isomorphism on both of the K-cycles created by σ.

Lastly, let $\sigma \supset \rho$ be an S_j-strategy such that $\sigma \supseteq \rho^\frown m$ for some $m \in \omega$, and let C_0 and D_0 be the K-cycles built by σ. By construction, the only way that ρ will consider C_0 and D_0 is if at some stage after their creation, ρ has outcome ∞. But once ρ has outcome ∞, σ will never be visited again in the construction, and thus C_0 and D_0 can never change types; they can only be extended by levels. So once ρ finds truncated K-cycles in $\mathcal{B}_i$ that appear isomorphic to C_0 and D_0, ρ can define h_i on C_0 and D_0, and will never have to redefine it further except to extend it as the K-cycles grow. Therefore, h_i will be a computable isomorphism on these K-cycles.

By construction, every component in $\mathcal{A}$ is eventually considered by ρ. Moreover, by all of the above cases, ρ will define h_i on every component it considers, and it will be correct and computable for every component on which h_i is defined. Therefore, h_i is a computable isomorphism from $\mathcal{A}$ to $\mathcal{B}_i$, and R_i is satisfied.

Claim 4. Each requirement S_j is satisfied along the true path.

Proof of Claim 4. Fix some $j \geq 0$ and let σ be the S_j-strategy on the true path. If $\sigma^\frown 0 \subset p$, then σ has final outcome 0 and never passed step 2 of its strategy. So, by the argument stated earlier, $(\Phi_j, \vec{c}_j)$ is not a Scott family for $\mathcal{A}$, and thus S_j is satisfied.

If $\sigma^\frown 1 \subset p$, then σ has final outcome 1, and thus extends a Type 1 K-cycle into a Type 2 K-cycle at step 3a of its procedure. Since σ is on the true path, it will only need to wait finitely many stages before extending its Type 0 K-cycle into a Type 1 K-cycle (as explained earlier). Thus, σ will ultimately reach step 4 of its procedure and successfully diagonalize against $(\Phi_j, \vec{c}_j)$. Therefore, once again, $(\Phi_j, \vec{c}_j)$ is not a Scott family for $\mathcal{A}$, and S_j is satisfied.

□

3.3 An Application to the Collatz Conjecture

One interesting example of a particular (2,1):1 structure is given to us by an open problem in number theory, due to Lothar Collatz in the 1930's. Suppose that, given a natural number, we divide it by 2 if it is even, and multiply it by 3 and add 1 if it is odd. This gives us a new natural number, on which we can repeat the same

procedure. If we start with an arbitrary natural number and iterate this procedure enough times, what will be the outcome? The *Collatz conjecture* states that, given *any* natural number n, this iterative process on n will always eventually lead to the number 1.

The Collatz conjecture, also known as the $3n + 1$ *conjecture, the Thwaites conjecture, the Syracuse problem, Kakutani's problem, Ulam's problem*, and *Hasse's algorithm*, has received a great deal of attention since its formulation, due to its deceptively simple statement. See the annotated bibliographies by Lagarias ([27], [28]), as well as [29], for more details on the history of the problem and the research done on it. Despite considerable efforts by many mathematicians, the Collatz conjecture has been neither proven nor disproven. Conway [9] showed in 1972 that problems of this type can be formally undecidable, and in 2007 Kurtz and Simon [26] proved that a natural generalization of the Collatz problem is, in fact, undecidable. However, nothing to this effect has been proven for the original Collatz problem.

The goal of this section is not to directly address the truth or undecidability of the conjecture, but rather to establish a connection between our theory and the conjecture, as well as to highlight some related interesting open problems. We start by giving a formal definition of the *Collatz function* to capture the iterative process informally described above.

Definition 3.3.1. The *Collatz structure* is the structure $\mathcal{C} = (C, f)$ where $C = \mathbb{N} - \{0\}$ and $f : C \to C$ is defined as:

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even,} \\ \\ \frac{3x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

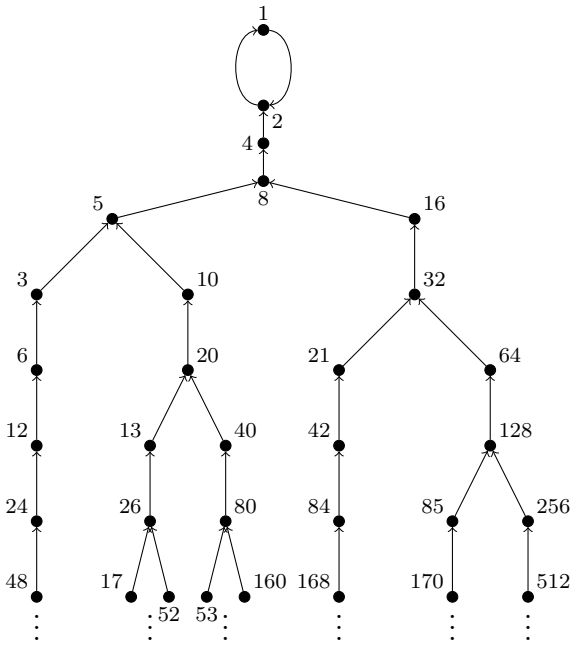
Notice that in our formal definition, we take the additional step of immediately dividing $3x+1$ by 2 when x is odd. This is, of course, equivalent to our original description, since $3x + 1$ is even whenever x is odd, and thus the next iteration of the rule would divide $3x + 1$ by 2.

We may now formally state the Collatz conjecture.

Conjecture 3.3.2 (Collatz 1937)**.** *For all $x \in C$, there exists an $n \in \mathbb{N}$ such that $f^n(x) = 1$.*

A common approach used to study the Collatz problem is to look at the graph of inverse iterates of 1 under the function f. This is often referred to as the *Collatz graph*, and is pictured in the figure below. The Collatz conjecture then states that this graph contains all natural numbers.

Figure 3.3: The Collatz graph.



The graph above very closely resembles a 2-cycle in a (2,1):1 structure. Indeed, it is easy to see that the Collatz structure $\mathcal{C}$ is in fact a (2,1):1 structure. This is due to the fact that a number $x \in C$ has exactly one pre-image under f (namely, $2x$) if $x \equiv 0 \pmod 3$ or $x \equiv 1 \pmod 3$, and exactly two pre-images under f (namely, the *left branch* $\frac{2x-1}{3}$ and the *right branch* $2x$) if $x \equiv 2 \pmod 3$. Thus, the Collatz conjecture states that the 2-cycle containing 1 is the only connected component in the structure.

Not only is the Collatz structure $\mathcal{C}$ a (2,1):1 structure, but it is also a computable (2,1):1 structure with a computable branching function. The computability of $\mathcal{C}$ follows immediately from Definition 3.3.1, while computability of the branching function $\beta_{\mathcal{C}}$ follows from the aforementioned fact that $\beta_{\mathcal{C}}(x) = 2$ if and only if $x \equiv 2 \pmod 3$. Perhaps unsurprisingly, the branch isomorphism function of $\mathcal{C}$ is also computable,

but demonstrating this requires significantly more effort. To do so, we first need the following theorem.

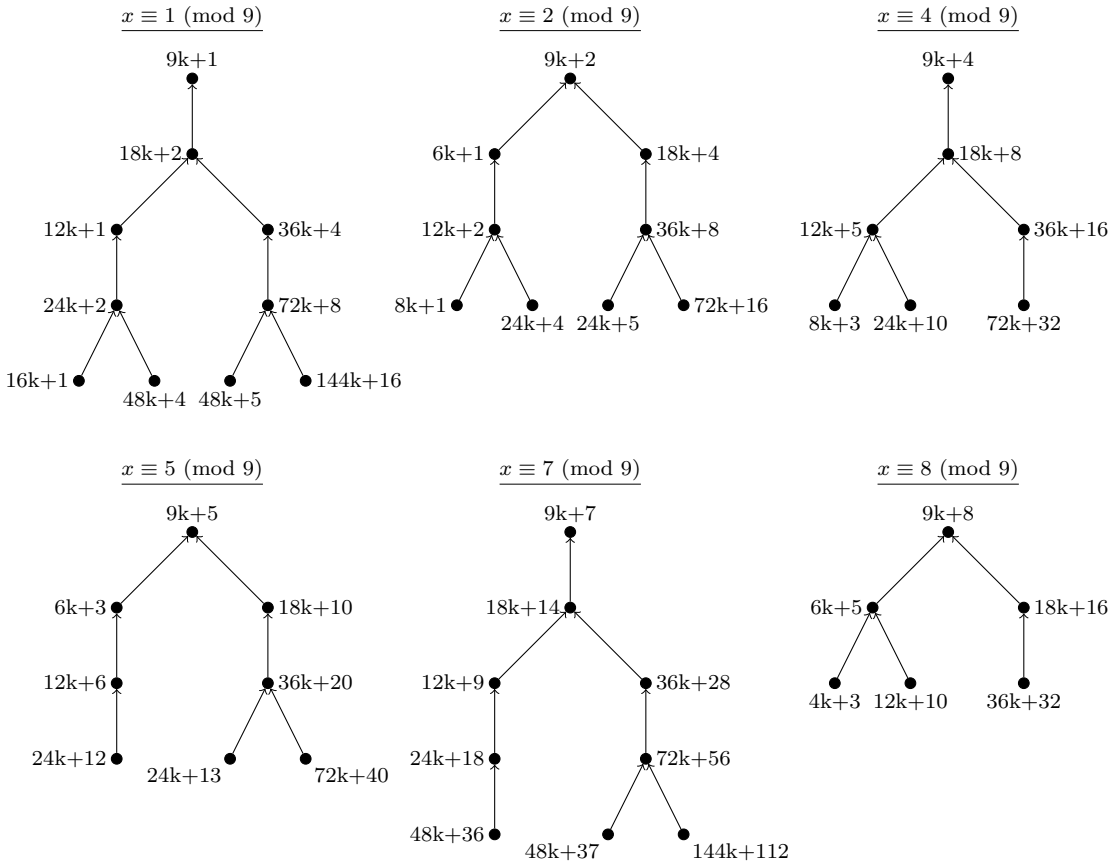
Theorem 3.3.3. *Let $\mathcal{C} = (C, f)$ be the Collatz structure, and let $x_1, x_2 \in C$, with $x_1 \neq x_2$. For all $n \geq 1$, if $x_1 \not\equiv x_2 \pmod{3^n}$, and at least one of x_1 and x_2 is not a multiple of 3, then $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$.*

Proof. Suppose that exactly one of x_1 and x_2 is not a multiple of 3, and assume without loss of generality that $x_1 \equiv 0 \pmod{3}$ and $x_2 \not\equiv 0 \pmod{3}$. Then $Tree_{\mathcal{C}}(x_1)$ is a degenerate tree, since every predecessor of x_1 under f would also be congruent to 0 (mod 3). However, because $x_2 \not\equiv 0 \pmod{3}$, $Tree_{\mathcal{C}}(x_2)$ cannot be degenerate, as it must contain a predecessor that is congruent to 2 (mod 3), which would have two pre-images under f. So $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$.

Thus, for the remainder of the proof we shall assume that neither x_1 nor x_2 is a multiple of 3. We proceed by induction on n. For the sake of convenience during the inductive step, our base case will include the cases of both $n = 1$ and $n = 2$.

For $n = 1$, if $x_1 \not\equiv x_2 \pmod{3}$, then one of x_1 and x_2 is a hair and the other is a split hair, and so their Trees are automatically not isomorphic. To show the same result for $n = 2$, consider the following truncated Trees for elements that are congruent to 1, 2, 4, 5, 7, and 8 modulo 9.

Figure 3.4: Truncated Trees for residue classes modulo 9 (where $k > 0$).



By inspection of the graphs above, it is evident that if $x_1 \not\equiv x_2 \pmod 9$, then $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$, proving the case for $n = 2$.

Now, assume that for some $m \geq 2$, if $x_1 \not\equiv x_2 \pmod{3^m}$ then $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$. We wish to show that if $x_1 \not\equiv x_2 \pmod{3^{m+1}}$ then $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$. So let's suppose that $x_1 \not\equiv x_2 \pmod{3^{m+1}}$. We may also assume that $x_1 \equiv x_2 \pmod{3^m}$, since if they were not, we would immediately have $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$ by the inductive hypothesis. In particular, we can assume that $x_1 \equiv x_2 \pmod 9$, and proceed by investigating each of the six possible residue classes modulo 9.

First, suppose that $x_1 \equiv x_2 \equiv 2 \pmod 9$. Then we can write: $x_1 = 3^{m+1}k_1 + a_1$

and $x_2 = 3^{m+1}k_2 + a_2$, where $a_1 \equiv a_2 \equiv 2 \pmod 9$, $a_1, a_2 < 3^{m+1}$, and $a_1 \neq a_2$. Now, the two pre-images of x_1 are $2 \cdot 3^m k_1 + \frac{2a_1-1}{3}$ and $2(3^{m+1}k_1 + a_1)$, while the two pre-images of x_2 are $2 \cdot 3^m k_2 + \frac{2a_2-1}{3}$ and $2(3^{m+1}k_2 + a_2)$. Observe that $\frac{2a_1-1}{3} \not\equiv \frac{2a_2-1}{3} \pmod{3^m}$. (Otherwise, $2a_1 - 1 \equiv 2a_2 - 1 \pmod{3^{m+1}}$, and so $a_1 \equiv a_2 \pmod{3^{m+1}}$. However, since both a_1 and a_2 are less than 3^{m+1}, this would imply that $a_1 = a_2$, contradicting our assumption.) So, $2 \cdot 3^m k_1 + \frac{2a_1-1}{3} \not\equiv 2 \cdot 3^m k_2 + \frac{2a_2-1}{3} \pmod{3^m}$. Also, since $x_1 \equiv x_2 \equiv 2 \pmod 9$, we know that neither of the left branches of x_1 and x_2 is a multiple of 3. Thus, by inductive hypothesis, the left branches of x_1 and x_2 are not isomorphic to each other.

However, since we assumed that $x_1 \equiv x_2 \pmod{3^m}$, we have that $2(3^{m+1}k_1 + a_1) \equiv 2(3^{m+1}k_2 + a_2) \pmod{3^m}$, i.e., the right branches of x_1 and x_2 are congruent modulo 3^m. So any isomorphism between $Tree_{\mathcal{C}}(x_1)$ and $Tree_{\mathcal{C}}(x_2)$ *must* map the left branch of x_1 to the left branch of x_2, but also *cannot* map the left branch of x_1 to the left branch of x_2. Thus, there is no isomorphism from $Tree_{\mathcal{C}}(x_1)$ to $Tree_{\mathcal{C}}(x_2)$, and therefore $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$. This proves the theorem for $x_1 \equiv x_2 \equiv 2 \pmod 9$.

Next, suppose $x_1 \equiv x_2 \equiv 5 \pmod 9$. Then again, we can write $x_1 = 3^{m+1}k_1 + a_1$ and $x_2 = 3^{m+1}k_2 + a_2$, where $a_1 \equiv a_2 \equiv 5 \pmod 9$, $a_1, a_2 < 3^{m+1}$, and $a_1 \neq a_2$. Since $x_1 \equiv x_2 \equiv 5 \pmod 9$, the left branches of x_1 and x_2, which are $2 \cdot 3^m k_1 + \frac{2a_1-1}{3}$ and $2 \cdot 3^m k_2 + \frac{2a_2-1}{3}$ respectively, are both congruent to 0 (mod 3), while the right branches, which are $2(3^{m+1}k_1 + a_1)$ and $2(3^{m+1}k_2 + a_2)$ respectively, are both congruent to 1 (mod 3). Thus, any isomorphism from $Tree_{\mathcal{C}}(x_1)$ to $Tree_{\mathcal{C}}(x_2)$ must map $2(3^{m+1}k_1 + a_1)$ to $2(3^{m+1}k_2 + a_2)$, and consequently, must also map their unique pre-images, $4(3^{m+1}k_1 + a_1)$ and $4(3^{m+1}k_2 + a_2)$, to each other.

However, $4(3^{m+1}k_1 + a_1) \not\equiv 4(3^{m+1}k_2 + a_2) \pmod{3^{m+1}}$, otherwise $a_1 \equiv a_2 \pmod{3^{m+1}}$ and then $a_1 = a_2$, contradicting our assumption. Also, both $4(3^{m+1}k_1 + a_1)$ and $4(3^{m+1}k_2 + a_2)$ are congruent to 2 (mod 9). Thus, by the previous case, the two numbers do not have isomorphic Trees. Therefore, x_1 and x_2 also do not have isomorphic Trees, proving the theorem for the case where $x_1 \equiv x_2 \equiv 5 \pmod{9}$.

Now, suppose $x_1 \equiv x_2 \equiv 8 \pmod{9}$. Then once again, we can write $x_1 = 3^{m+1}k_1 + a_1$ and $x_2 = 3^{m+1}k_2 + a_2$, where $a_1 \equiv a_2 \equiv 8 \pmod{9}$, $a_1, a_2 < 3^{m+1}$, and $a_1 \neq a_2$. Since $x_1 \equiv x_2 \equiv 8 \pmod{9}$, the left branches of x_1 and x_2 are split hairs while the right branches are hairs. Hence, any isomorphism from $Tree_{\mathcal{C}}(x_1)$ to $Tree_{\mathcal{C}}(x_2)$ must map the right branch of x_1 to that of x_2, and consequently must map the unique pre-images of the right branches, $4(3^{m+1}k_1 + a_1)$ and $4(3^{m+1}k_2 + a_2)$, to each other. However, both of these numbers are congruent to 5 (mod 9), and yet they are not congruent to each other (mod 3^{m+1}), by the same argument as before. Thus, by the previous case, the two numbers do not have isomorphic Trees, and therefore, neither do x_1 and x_2. So the theorem is proved for $x_1 \equiv x_2 \equiv 8 \pmod{9}$.

Finally, suppose that $x_1 \equiv x_2 \equiv 1$, 4, or 7 (mod 9), but $x_1 \not\equiv x_2 \pmod{3^{m+1}}$. Then both x_1 and x_2 are hairs and thus their only pre-image is $2x_1$ and $2x_2$, respectfully. Of course, any isomorphism from $Tree_{\mathcal{C}}(x_1)$ to $Tree_{\mathcal{C}}(x_2)$ must map $2x_1$ to $2x_2$. However, $2x_1 \equiv 2x_2 \equiv 2$, 5, or 8 (mod 9) and $2x_1 \not\equiv 2x_2 \pmod{3^{m+1}}$, otherwise $x_1 \equiv x_2 \pmod{3^{m+1}}$, a contradiction. Thus, due to the previous three cases, $Tree_{\mathcal{C}}(2x_1) \ncong Tree_{\mathcal{C}}(2x_2)$, and therefore $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$.

Hence, the theorem is proved for all natural numbers x_1 and x_2. □

The following corollary is an easy consequence of Theorem 3.3.3. It says that the only elements in $\mathcal{C}$ with isomorphic trees are the multiples of 3.

Corollary 3.3.4. *Let $\mathcal{C} = (C, f)$ be the Collatz structure, and let $x_1, x_2 \in C$, with $x_1 \neq x_2$. Then $Tree_{\mathcal{C}}(x_1) \cong Tree_{\mathcal{C}}(x_2)$ if and only if $x_1 \equiv x_2 \equiv 0 \pmod 3$.*

Proof. ($\longleftarrow$) If $x_1 \equiv x_2 \equiv 0 \pmod 3$, then the Trees stemming from x_1 and x_2 are necessarily degenerate trees, as every predecessor of x_1 and x_2 under f would also be congruent to 0 (mod 3). Thus, $Tree_{\mathcal{C}}(x_1) \cong Tree_{\mathcal{C}}(x_2)$.

($\longrightarrow$) Suppose that at least one of x_1 and x_2 is not congruent to 0 (mod 3). Choose $n \in \mathbb{N}$ such that both x_1 and x_2 are less than 3^n. Then, since $x_1 \neq x_2$, $x_1 \not\equiv x_2 \pmod{3^n}$. Therefore, by Theorem 3.3.3, $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$. □

We can now show that the branch isomorphism function of the Collatz structure is computable, in a trivial way.

Corollary 3.3.5. *Let $\mathcal{C} = (C, f)$ be the Collatz structure. Then $iso_{\mathcal{C}}$ is computable. In fact, $iso_{\mathcal{C}}(x) = 0$ for all $x \in \Lambda_{\mathcal{C}}$.*

Proof. Let $x \in \Lambda_{\mathcal{C}}$, with distinct pre-images x_1 and x_2. Since $x \equiv 2 \pmod 3$, at least one of x_1 and x_2 is not congruent to 0 (mod 3). Thus, by Corollary 3.3.4, $Tree_{\mathcal{C}}(x_1) \ncong Tree_{\mathcal{C}}(x_2)$, and therefore, $iso_{\mathcal{C}}(x) = 0$. □

Having established the computability of the branching and branch isomorphism functions of the Collatz structure, we can draw a connection between the computable categoricity of $\mathcal{C}$ and the truth of the Collatz conjecture, as the following corollary demonstrates.

Corollary 3.3.6. *Let $\mathcal{C} = (C, f)$ be the Collatz structure. If the Collatz conjecture holds, then $\mathcal{C}$ is computably categorical. Equivalently, if $\mathcal{C}$ is not computably categorical, then the Collatz conjecture is false.*

Proof. By Corollary 3.3.5, we have that both $\beta_{\mathcal{C}}$ and $iso_{\mathcal{C}}$ are computable. Furthermore, if the Collatz conjecture is true, then $\mathcal{C}$ consists of only one K-cycle, namely the 2-cycle that contains the number 1, and no $\mathbb{Z}$-chains. Therefore, by Theorem 3.1.2, $\mathcal{C}$ is computably categorical. □

Given the categoricity results from Section 3.1, a natural question to ask about the Collatz structure is the following: given the existence of two K-cycles in $\mathcal{C}$, is it possible for one K-cycle to properly embed (or properly N^+-embed) into the other? Due to Corollary 3.3.4, we know that if x_1 and x_2 are distinct elements in C and $Tree_{\mathcal{C}}(x_1)$ is not degenerate, then there is no embedding from $Tree_{\mathcal{C}}(x_1)$ into $Tree_{\mathcal{C}}(x_2)$ that maps x_1 to x_2. It is not hard to see that any nontrivial K-cycle in $\mathcal{C}$ must contain at least one cyclic element whose exclusive Tree is not degenerate. Thus, we can conclude that it is not possible for any K-cycle in the Collatz structure to properly embed into any other K-cycle. So although the full Collatz graph may not be computably categorical, we cannot use Theorem 3.1.7 to prove this.

Also, the converse of Corollary 3.3.6 is not necessarily true. It is possible for the Collatz conjecture to fail, but for the structure itself to still be computably categorical. For example, if there are only finitely many non-trivial cycles in the Collatz structure, Theorem 3.1.2 would still guarantee that $\mathcal{C}$ is computably categorical, even though the conjecture would be false.

Unfortunately, very few results on the possible counterexamples to the Collatz conjecture are known. Due to an ongoing computation by Oliveira e Silva [42], the conjecture has been verified for all natural numbers less than $n = 19 \cdot 2^{58}$, giving a lower bound for the set of counterexamples to the conjecture. Also, Lagarias [29] proved that there are no nontrivial K-cycles for $\mathrm{K} < 275,000$. However, it is not known whether there can be infinitely many K-cycles for a given size K, whether there can be infinitely many $\mathbb{Z}$-chains, or whether $\mathbb{Z}$-chains can exist in the Collatz structure at all. These open problems have given rise to weaker versions of the Collatz conjecture, which are still being investigated.

Bibliography

[1] C.J. Ash and J.F. Knight, *Computable Structures and the Hyperarithmetical Hierarchy*, volume 144 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, 2000.

[2] C.J. Ash, J.F. Knight, M.S. Manasse, and T.A. Slaman, Generic copies of countable structures, *Ann. Pure Appl. Logic* 42 (1989) 195-205.

[3] W.W. Boone, The word problem, *Annals of Mathematics* Second Series, Vol. 70, No. 2 (1959), pp. 207-265.

[4] W. Calvert, D. Cenzer, V. Harizanov, and A. Morozov, Effective categoricity of equivalence structures, *Annals of Pure and Applied Logic* 141 (2006), pp. 61-78.

[5] D. Cenzer, B.F. Csima, and B. Khoussainov, Linear orders with distinguished function symbol, *Archive for Mathematical Logic, Special Issue: University of Florida Special Year in Logic*, 48 (2009), no. 1, 63-76.

[6] D. Cenzer, V. Harizanov, and J.B. Remmel, Computability-theoretic properties of injection structures, *Algebra and Logic* 53 (2014), pp. 39-69.

[7] D. Cenzer, V. Harizanov, and J.B. Remmel, Two-to-one structures, *Journal of Logic and Computation* 23 (2013), pp. 1195-1223.

[8] J. Chisholm, Effective model theory vs. recursive model theory, *J. Symbolic Logic* 55 (1990) 1168-1191.

[9] J.H. Conway, Unpredictable iterations, *Proc. 1972 Number Th. Con.*, University of Colorado, Boulder, Colorado (1972), pp. 49-52.

[10] S.B. Cooper, *Computability Theory*, Chapman Hall/CRC Mathematics Series, Chapman and Hall/CRC, 2004.

[11] B. Csima, B. Khoussainov, and J. Liu, Computable categoricity of graphs with finite components, *Lecture Notes in Computer Science* 5028 (2008), pp. 139-148.

[12] N.J. Cutland, *Computability*, Cambridge University Press, 1980.

[13] R.G. Downey, A.M. Kach, S. Lempp, A.E.M. Lewis-Pye, A. Montalbán, and D.D. Turetsky, The complexity of computable categoricity, *Advances in Mathematics*, Vol. 268 (2015) 423-466.

[14] R.G. Downey, A.M. Kach, S. Lempp, and D.D. Turetsky, Computable categoricity versus relative computable categoricity, *Fundamentae Mathematica* Vol. 221 (2013), 129-159.

[15] V. Dzgoev and S. Goncharov, Autostability of models, *Algebra and Logic* 19 (1980), pp. 28-37.

[16] E.B. Fokina, V. Harizanov, and A. Melnikov, Computable model theory, In Rod Downey, editor, *Turing's Legacy: Developments from Turing's Ideas in Logic*, volume 42 of *Lecture Notes in Logic*, pp. 124-194, Cambridge University Press, Cambridge, May 2014.

[17] R.M. Friedberg, Three theorems on recursive enumeration: I. Decomposition, II. Maximal Set, III. Enumeration without duplication, *J. Symbolic Logic* 23 (1958), pp. 309-316.

[18] R.M. Friedberg, Two recursively enumerable sets of incomparable degrees of unsolvability, *Proc. Natl Acad, Sci. USA* 43 (1957), pp. 236-238

[19] A. Fröhlich and J.C. Shepherdson, Effective procedures in field theory, *Philos. Trans. Roy. Soc. London. Ser. A.* vol. 248 (1956), pp. 407-432.

[20] S.S. Goncharov, Autostability and computable families of constructivizations, *Algebra and Logic*, vol. 14 (1975), (English translation) pp. 392-408.

[21] S.S. Goncharov, The number of nonautoequivalent constructivizations, *Algebra i Logika* 16(3):257-282, 377, 1977.

[22] S.S. Goncharov, The problem of the number of nonautoequivalent constructivizations (Russian), *Algebra i Logika* 19 (1980), pp. 621-639.

[23] S. Goncharov, S. Lempp, and R. Solomon, The computable dimension of ordered abelian groups, *Advances in Mathematics* 175 (2003), pp. 102-143

[24] V.S. Harizanov, Pure computable model theory, In Yu. L. Ershov, S.S. Goncharov, A. Nerode, and J.B. Remmel, editors, *Handbook of recursive mathematics, Vol. 1 – Recursive Model Theory*, volume 138 of *Stud. Logic Found. Math*, pp. 3-114, North-Holland, Amsterdam, 1998.

[25] D.R. Hirschfeldt, B. Khoussainov, R.A. Shore, and A.M. Slinko, Degree spectra and computable dimensions in algebraic structures, *Ann. Pure Appl. Logic* vol. 115 (2002), pp. 71-113.

[26] S.A. Kurtz and J. Simon, The undecidability of the generalized Collatz problem, *Theory and Applications of Models of Computation: Proceedings of the 4th International Conference* (*TAMC 2007*) *held in Shanghai*, May 22-25, 2007, Ed. J.-Y. Cai, S.B. Cooper, and H. Zhu, Berlin: Springer, pp. 542–553.

[27] J.C. Lagarias, The $3x + 1$ problem: an annotated bibliography (1963-1999), *eprint: arxiv:math.NT/0309224* Sept. 13, 2003, v11.

[28] J.C. Lagarias, The $3x + 1$ problem: an annotated bibliography II (2000-2009), *eprint: arxiv:math.NT/0608208* Feb. 12, 2012, v6.

[29] J.C. Lagarias, The $3x+1$ problem and its generalizations, *Amer. Math. Monthly* 92 (1985), pp. 3-23.

[30] P. LaRoche, Recursively presented boolean algebras, *Notices of the American Mathematical Society* 24 (1977), A552-A553.

[31] S. Lempp, C. McCoy, R. Miller, and R. Solomon, Computable categoricity of trees of finite height, *Journal of Symbolic Logic* 70 (2005), pp. 151-215.

[32] J. Liu, *A journey from finite to automatic structures and beyond*, Ph.D. Thesis, University of Auckland, Auckland, New Zealand.

[33] A.I. Mal'cev, On recursive abelian groups, *Dokl. Akad. Nauk SSSR* vol. 146 (1962), pp. 1009-1012.

[34] L. Marshall, *Computability-theoretic properties of partial injections, trees, and nested equivalences*, Ph.D. Thesis, The George Washington University, Washington, D.C.

[35] Y. Matiyasevich, *Hilbert's Tenth Problem*, MIT Press, 1993.

[36] R. Miller, The computable dimension of trees of infinite height, *Journal of Symbolic Logic* 70 (2005), pp. 111-141.

[37] A.A. Muchnik, On the unsolvability of the problem of reducibility in the theory of algorithms, *Dokl. Akad. Nauk SSSR, N.S.* 108 (1956), pp. 194-197 (Russian).

[38] P.S. Novikov, On the algorithmic unsolvability of the word problem in group theory, *Trudy Mat. Inst. Steklov* 44 (1955), pp. 1-143.

[39] M.O. Rabin, Computable algebra, general theory and theory of computable fields, *Transactions of the American Mathematical Society*, vol. 95 (1960), pp. 341-360.

[40] J.B. Remmel, Recursive isomorphism types of recursive boolean algebras, *Journal of Symbolic Logic* 46 (1981), pp. 572-594.

[41] J.B. Remmel, Recursively categorical linear orderings, *Proceedings of the American Mathematical Society* 83 (1981), pp. 387-391.

[42] Tomas Oliveira e Silva, Computational verification of the $3x + 1$ conjecture, *http://www.ieeta.pt/ tos/3x+1.html* (2008).

[43] R.I. Soare, *Recursively Enumerable Sets and Degrees. A Study of Computable Functions and Computably Generated Sets*, Spring-Verlag, Berlin, 1987.

[44] R.I. Soare, *Turing Computability* (*Theory and Applications*), Springer, 2016.

[45] R. Steiner, *Reducibility, degree spectra, and lowness in algebraic structures*, Ph.D. Thesis, CUNY Graduate Center, New York, NY.

[46] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.* 42 (1936), pp. 230–265.

[47] B.L. van der Waerden, Eine Bemerkung über die Unzerlegbarkeit von Polynomen, *Mathematische Annalen*, vol. 102 (1930), pp. 738-739.

[48] H.J. Walker, Computable isomorphisms for certain classes of infinite graphs, to appear.