A LOGICAL INSIGHT INTO THE THEORY OF COMPUTATION

MARC DE FONTNOUVELLE

Abstract. Descriptive complexity is a relatively new field in computer science that illuminates connections between mathematical logic and computational complexity theory. In this paper we will motivate and prove the Immerman-Vardi theorem, which gives a logical characterization of the computational complexity class P. We will also sketch a proof of Fagin's theorem, which provides a logical characterization of the class NP. Taken together, these theorems show how open questions in the theory of computation such as whether P equals NP can be re-formulated as questions in mathematical logic.

Contents

1. Introduction

The theory of computation studies the power of *algorithms*, which are machines that take an input, manipulate it according to a fixed set of instructions, and return the resulting output. The development of this field was motivated by questions such as the *Entscheidungsproblem* [Hil28], posed by David Hilbert in 1928:

Question 1.1. *Does there exist an algorithm which, given any mathematical statement and a set of axioms, determines whether the statement is provable from the axioms?*

If such an algorithm existed, it would effectively automate the work of mathematicians: to determine the truth of any mathematical conjecture, we could simply input it into this algorithm. Hilbert believed it likely that such an algorithm indeed existed; in 1936, however, Alan Turing [Tur36] and Alonzo Church [Chu36]

proved the impossibility of such an algorithm.

Later in the 20th century, the development of physical computers made the abstract concept of an algorithm more readily implementable in reality. This led mathematicians to consider questions about the limitations of *feasible* computation, such as whether there exist problems that can not be solved efficiently by computers – that is, within a given amount of time, or using only a given amount of memory.

Observe that in his *Entscheidungsproblem*, Hilbert did not impose any restriction on the amount of time or memory used by the hypothetical truth-determining algorithm. Mathematicians thus considered the following variation of the *Entscheidungsproblem*:

Question 1.2. *Does there exist an efficient algorithm which, given any mathematical statement and a set of axioms, determines whether an efficient (i.e., relatively short) proof of the statement can be deduced from the axioms?*

The existence of such an algorithm would perhaps have even more profound implications than the existence of Hilbert's truth-determining algorithm, for it would allow much of the work of mathematics to be *feasibly* automated by a physical computer. However, Question 1.2 has proven much more difficult to resolve than the *Entscheidungsproblem*, and remains an open question to this day. It is in fact closely related to the P vs. NP question, which stands as the most prominent open question in computer science.[1] One formulation of the P vs. NP question is as follows:

Question 1.3 (P vs. NP)**.** *If there exists an algorithm that efficiently verifies solutions to a problem, does there also exist an algorithm that efficiently solves the same problem?*

The P vs. NP question is an instance of a range of related questions in the theory of computation which were formulated in the latter half of the 20th century. In spite of significant developments in the field of computer science, virtually all of these questions remain open.

In this paper, we examine the emerging field of *descriptive complexity*, which uses mathematical logic to analyze the theory of computation. We will find a remarkably deep connection between logic and computability, and will show that many central concepts in the theory of computation have natural analogues in logic. It will follow that many of these fundamental open questions about computability, including the P vs. NP question, are equivalent to very natural open questions about mathematical logic.

In some cases, the methods of descriptive complexity have been used to resolve long-standing open questions in computer science.[2] In other cases, descriptive complexity

[1]Observe that the problem of determining the existence of an efficient proof is in NP, beacuse a non-deterministic algorithm can simply "guess" the proof. Therefore $P = NP$ implies a positive answer to Question 1.2.

[2]For instance, Neil Immerman and Róbert Szelepcsényi used descriptive complexity to prove that non-deterministic space complexity classes such as NL are closed under complementation [Imm88].

at least offers us new ways to understand and approach these questions. It shows that questions such as whether P is equal to NP are not merely questions about the power of computers, but also more general questions about the expressibility of mathematical properties. As such, descriptive complexity helps us understand why such seemingly simple questions about computation have proven so difficult for mathematicians to answer.

Before examining this connection between logic and computability, we will introduce the relevant concepts in each field. We will define *descriptive* and *computational* complexity classes and introduce several techniques that will prove useful as we relate these two types of classes.

2. Logical Complexity

If mathematics is an abstraction of the physical world, mathematical logic is an abstraction of mathematics itself. More specifically, logic studies mathematical structures as models of an underlying logical system. Let us formally define the notions of a logical *vocabulary* and a *model* thereof:

Definition 2.1 (Vocabulary)**.** A *vocabulary*

$$\tau = \langle R_1^{a_1}, ..., R_r^{a_r}, c_1, ..., c_s, f_1^{b_1}, ..., f_f^{b_f} \rangle$$

is a tuple of relation symbols, constant symbols, and function symbols. Here $R_i^{a_i}$ denotes a relation symbol with arity a_i, while $f_i^{b_i}$ denotes a function with arity b_i.

Definition 2.2 (Model)**.** A *model* M of a vocabulary τ

$$M = \langle A, I \rangle$$

is an interpretation of the symbols of τ over the set A (called the "universe"), given by the interpretation function I. In particular, I maps each relation symbol $R_i^{a_i} \in \tau$ to a relation of arity a_i over A, i.e., $I(R_i^{a_i}) \subseteq A^{a_i}$. Likewise, I maps each function symbol $f_i^{b_i} \in \tau$ to a function of arity b_i over A, i.e., $I(f_i^{b_i})$ is a total function from A^{b_i} to A, and maps each constant symbol c_i to an element of A. We denote by $MOD(\tau)$ the set of all possible models of a vocabulary τ.

Example 2.3. The vocabulary of undirected graphs consists of a single binary relation symbol:

$$\tau_g = \langle E^2 \rangle$$

Any undirected graph can be understood as a model of this vocabulary: the universe A is the set of vertices of the graph, while the relation $I(E^2)$ describes the edges between these vertices. For example, the complete graph on four vertices is given by $K_4 = \langle A, I \rangle$, where $A = \{0, 1, 2, 3\}$ and $I(E^2) = \{(x, y) \mid x, y \in [0, 3]; x \neq y\}$.

We assume familiarity with the concept of first-order logic, and denote by $FO(\tau)$ the set of first-order formulae in the vocabulary τ. To say that a model M *satisfies* a formula ϕ, we will write $M \vDash \phi$. For instance, if we define:

$$\phi_{complete} \equiv (\forall x)(\forall y)(x \neq y \Rightarrow E(x,y))$$
$$\phi_{2out} \equiv (\forall x)(\exists yz)(y \neq z \wedge E(x,y) \wedge E(x,z))$$

Then we have:

$$K_4 \vDash \phi_{complete} \wedge \phi_{2out}.$$

We will not go through a formal definition of satisfaction, which can be found in Section 1.1 of [Imm99].

2.1. **Descriptive Complexity Class.**

An algorithm can be understood as a function which assigns an output to every possible input that it may receive. From a logical point of view, we can consider both the input and the output of an algorithm as models of a logical vocabulary. The algorithm is therefore given by a map between these models, which we call a *query*.

Definition 2.4 (Query)**.** A query is a map $Q : MOD(\sigma) \to MOD(\tau)$ from models of one vocabulary to models of another vocabulary. A *Boolean query* is a map $Q : MOD(\sigma) \to \{0,1\}$.

A *first-order query* is a query which can be defined using only first-order logic. That is, the universe of $Q(M)$ is a first-order definable subset of tuples of the universe of M; each relation R_i over $Q(M)$ is a first-order definable subset of this universe; and so forth. A more precise (and lengthy) definition of a first-order query can be found in Section 1.4 of [Imm99].

In this paper, we will deal primarily with Boolean queries. A Boolean query can be understood as representing an algorithm that returns a truth value for each input (1 for "true" and 0 for "false"). Any first-order formula $\phi \in FO(\tau)$ defines a first-order Boolean query Q on $MOD(\tau)$, where $Q(M) = 1$ if and only if $M \vDash \phi$.

For instance, let COMPLETE be the query on graphs that is true if and only if the input graph is complete. We see that

$$COMPLETE \equiv \phi_{complete}$$

and therefore, COMPLETE is a first-order Boolean query. We will sometimes informally refer to a Boolean query $I : MOD(\sigma) \to \{0,1\}$ as a *problem* whose solutions are precisely the models $M \in MOD(\sigma)$ for which $I(M) = 1$.

A *descriptive complexity class* is the set of all queries which can be expressed in a certain logical language. For example:

Definition 2.5. $Q(FO)$ is the set of all first-order queries, and FO is the set of all first-order Boolean queries.

Descriptive complexity classes provide a way to classify the complexity of problems according to the power of the logical language needed to express their solutions. For instance, the fact that COMPLETE $\in FO$ suggests that the problem of determining whether a given graph is complete is not a very difficult one, because completeness is an "easy" property to express.

Intuitively, we might suspect that any problem in FO can be solved by a relatively simple and efficient algorithm – and as we will soon see, this intuition is correct (cf. Lemma 4.8). More generally, the power of the logical language needed to express the solutions to a problem is closely related to the complexity of the algorithm needed to solve it. In order to examine this connection, let us review the foundations of computational complexity, and introduce the notion of a computational complexity class.

3. Computational Complexity

Computational complexity offers a different way to classify the difficulty of a mathematical problem: rather than considering the strength of the logical language needed to express its solutions, we consider the amount of time and/or space an algorithm would need to solve the problem. In this paper, we will use the Turing machine as our formal definition of an algorithm. We assume familiarity with the definition of a Turing machine, and we write $L(T)$ to denote the language accepted by a Turing machine T.

We also assume familiarity with the concept of a *non-deterministic* Turing machine. This is a theoretical machine which may be imagined as a computer whose processors have the ability to create copies of themselves. At every point in the computation, every processor may create n copies of itself, each of which undertakes a different computation. The "parent" processor then accepts if and only if one of its "offspring" reaches an accepting state. A non-deterministic Turing machine is of course more powerful than its deterministic counterpart, because it has the ability to run many processors simultaneously. However, these processors communicate with one another in a very limited way, as each processor can compute only the logical disjunction of its offspring.

If the processors of this machine could communicate with one another in a more intricate way – for example, if some processors could accept if and only if *all* of their offspring accepted – the resulting Turing machine would be significantly more powerful and efficient than an ordinary non-deterministic Turing machine. Such a machine, which we will call an *alternating* Turing machine, will prove useful as we characterize various computational complexity classes. Let us formally define the concept of alternation as follows:

Definition 3.1 (Alternating Turing Machine)**.** An alternating Turing machine T is a non-deterministic Turing machine whose states are divided into two groups: *existential* and *universal*. Let c denote an arbitrary configuration of T, i.e., a

description of its current state, work-tape contents, and head position. The machine T accepts in configuration c if and only if one of the following holds:

(1) c is in a final accepting state;
(2) c is in an existential state, and there exists a next configuration c' that leads T to accept;
(3) c is in a universal state, and there is at least one next configuration, and *all* next configurations accept.

Note that this definition of acceptance is inductive. Observe also that a non-deterministic Turing machine can be understood as an alternating Turing machine all of whose states are existential.

3.1. **Computational Complexity Class.**

Time complexity classes classify problems according to the number of steps an optimal algorithm must take in order to solve them. We will define the time complexity classes P and NP in the standard way:

Definition 3.2 (Time Complexity Classes)**.** A set $S \subseteq MOD(\sigma)$ is in the class P if there exists a deterministic Turing machine T such that $L(T) = S$, and T is polynomially bounded. That is, there exists some $k \in \mathbb{N}$ such that T computes every input of length n in at most $n^k + k$ steps. The class NP is defined analogously for non-deterministic Turing machines T.

Space complexity classes, meanwhile, consider the amount of *memory* – or more formally, the size of the working tape of the Turing machine – needed to solve a given problem. In this paper, we will work with the logarithmic space complexity classes L, NL, and AL:

Definition 3.3 (Space Complexity Classes)**.** A set $S \subseteq MOD(\sigma)$ is in the class L if there exists a deterministic Turing machine T such that $L(T) = S$, and T uses at most $O(log(n))$ squares of its working tape on an input of length n. The class NL is defined analogously for non-deterministic Turing machines T, while AL is defined analogously for *alternating* Turing machines T.

Lemma 3.4. $L \subseteq P$.

Proof. Let A be an arbitrary problem in L, so that it is decided by a logarithmic-space Turing machine T. Observe that T has only $2^{O(logn)}$ possible configurations (i.e., descriptions of its current working tape contents, state, and head positions) on an input of length n. Observe also that T can not repeat a configuration during its computation of any input w, otherwise it would enter an endless loop and neither accept nor reject w. Therefore, T takes at most $2^{O(logn)} = n^{O(1)}$ steps in the computation of an input of length n, so T runs in polynomial time. It follows that $A \in P$, and therefore $L \subseteq P$.

□

The following characterization of P, which we will not prove in this paper, will be useful to us:

Theorem 3.5. $P = AL$

That is, a problem is decided in deterministic polynomial time if and only if it is decided in alternating logarithmic space. This theorem is an instance of the more general result that $ASPACE(f(n)) = DTIME(2^{O(f(n))})$, which was proved in 1981 by Chandra, Kozen, and Stockmayer [Cha88].

3.2. **Reductions and Completeness.**

We now introduce the concept of *reducibility*, which allows us to classify the difficulty of problems within a given complexity class:

Definition 3.6 (Reduction)**.** Let $A \subseteq MOD(\sigma)$ and $B \subseteq MOD(\tau)$ be Boolean queries. A *reduction* from A to B is a query $I : MOD(\sigma) \to MOD(\tau)$ such that

$$M \in A \Leftrightarrow I(M) \in B.$$

If I is a first-order query, then it is a *first-order reduction* from A to B. In this case, we say that A is *first-order reducible* to B, denoted by $A \leq_{fo} B$.

The intuitive meaning of $A \leq_{fo} B$ is that the complexity of problem A is less than or equal to that of B. For instance, if $B \in FO$ and $A \leq_{fo} B$, then A must also be in the class FO. This result relies on the fact that first-order queries are closed under composition, which we leave as an exercise to the reader.

The vast majority of naturally occuring complexity classes, both computational and descriptive, are closed under first-order reductions, which we define as follows:

Definition 3.7 (Closure)**.** Let C be a set of Boolean queries. C is *closed* under first-order reductions if for every Boolean query I such that $I \leq_{fo} J$ and $J \in C$, I is also in C.

The concept of reduction also allows us to speak of the hardest or most complex problem within a complexity class, which we define as follows:

Definition 3.8 (Completeness)**.** Let $A \subseteq MOD(\sigma)$ be a Boolean query in a complexity class C. A is *complete* for C ("C-complete") under first-order reductions if for all $B \in C$, $B \leq_{fo} A$.

As examples, we will provide problems that are complete for the classes L and NL, and outline the respective proofs. We will later use a very similar proof technique to identify a problem that is complete for the class P, which will be a key step in proving one of the central results of descriptive complexity.

Definition 3.9 (PATH)**.** Let $\tau_{gst} = \langle E^2, s, t\rangle$ be the vocabulary of graphs with designated points s and t. Let PATH $\in MOD(\tau_{gst})$ be the set of graphs G which contain a path from s to t.

Theorem 3.10. PATH *is NL-complete under first-order reductions.*

Proof. We first show that PATH $\in NL$ by describing a non-deterministic algorithm T that decides PATH in logarithmic space. On input $\langle G, s, t\rangle$, where s and t are vertices in the graph G, T non-deterministically guesses a path from s to t. Starting at vertex s, the algorithm non-deterministically chooses one of the vertices connected

to the current vertex, and accepts its input if it ever reaches vertex t. If a branch of computation runs for more than $|G|$ steps, then this branch rejects. At any point in the computation, the algorithm only needs to store the current vertex in its path and the length of the path, which is feasible in logarithmic space. Therefore, PATH $\in NL$.

To complete the proof that PATH is NL-complete, we must show that for any $A \subseteq MOD(\sigma)$ in NL, there exists a first-order query $I : MOD(\sigma) \to MOD(\tau_{gst})$ such that for all $S \in MOD(\sigma)$:

$$S \in A \Leftrightarrow I(A) \in PATH$$

Because $A \in NL$, there exists a non-deterministic Turing machine T that decides A in logarithmic space. We construct I in such a way that $I(A)$ is a graphic representation of the computation of T on a given input. In particular, the set of vertices of $I(A)$ is the set CF of all possible configurations of T – that is, all the possible combinations of state, working tape contents, and head positions.

For any two configurations c_1, c_2 of T, (c_1, c_2) is an edge in $I(A)$ if and only if c_2 is one of the next possible configurations of T starting from c_1. Let c_S be the starting configuration of T on input S, and let c_F be the accepting configuration of T.[3]

More formally, $I(A)$ is defined as a model $\langle A, I\rangle$ of $\tau_{gst} = \langle E^2, s, t\rangle$ as follows:

(i) $A = CF$
(ii) $I(E^2) = \{(c_1, c_2) \in CF^2 \mid c_2 \in \delta(c_1)\}$, where δ denotes the transition relation of T
(iii) $I(s) = c_S$
(iv) $I(t) = c_F$.

By this definition, S is accepted by T if and only if there exists a path from c_S to c_F in the graph $I(A)$. It follows that I is indeed a reduction from A to $PATH$.

To show that I is a *first-order* reduction, we must show that $I(E^2), I(s)$, and $I(t)$ can be defined using first-order logic. This is evidently true in the case of $I(s)$ and $I(t)$. The proof that $I(E^2)$ is expressible in first-order logic is tedious, though not overly difficult; we therefore omit it from this paper, and refer the interested reader to Section 3.3 of [Imm99]. □

Corollary 3.11. *$NL \subseteq P$.*

Proof (sketch). It is not overly difficult to construct a polynomial-time algorithm that decides PATH; we refer the reader to Theorem 7.14 in [Sip12] for an example of such an algorithm. Now, for any problem A $\in NL$, there exists a first-order

[3]A non-deterministic Turing machine T may be defined to have multiple accepting states. However, in this case it is easy to create a slightly modified Turing machine T' which has only one accepting state, and is otherwise equivalent.

reduction I from A to $PATH$. A first-order reduction can be computed in polynomial time (we will prove this a bit later, in Corollary 4.9), and PATH can be decided in polynomial time. Therefore A $\in P$, and so $NL \subseteq P$. □

Using an analogous technique, we can show that the following problem is complete for the class L:

Definition 3.12 (DPATH)**.** Let DPATH $\in MOD(\tau_{gst})$ be the set of graphs G which contain a *deterministic* path from s to t. That is, for every edge (x, y) along this path, (x, y) is the unique edge leaving the vertex x.

Theorem 3.13. DPATH *is L-complete under first-order reductions.*

Proof (sketch). Let $A \in MOD(\sigma)$ be arbitrary, so that A is decided by a deterministic Turing machine T in logarithmic space. Using precisely the same technique as in the previous theorem, we construct a first-order query I such that $I(A) \in MOD(\tau_{gst})$ is a graphic representation of the computation of T on a given input. Because T is deterministic, every vertex in $I(A)$ will be connected to at most one other vertex. Therefore, T accepts an input S if and only if $I(A)$ contains a deterministic path from c_S to c_F.

We leave it as an exercise to the reader to show that DPATH $\in L$. It follows that DPATH is L-complete.

□

4. A Descriptive Characterization of P

We have seen that the fields of logical and complexity theory provide two different ways to classify the difficulty of mathematical problems, using *descriptive* and *computational* complexity classes, respectively. A natural question to ask is to what extent these two types of classes coincide. Does the power of the logical language needed to describe a problem indicate the complexity of an algorithm needed to solve it? For instance, if a problem is describable in a relatively weak logical language, is it necessarily decided by a relatively efficient algorithm?

The field of descriptive complexity reveals that there is indeed a deep connection between these two types of complexity classes. In fact, virtually every naturally occurring computational complexity class (L, NL, P, NP, $coNP$, $PSPACE$, and so forth) is precisely equal to a natural descriptive complexity class. As a result, many of the fundamental open questions in the theory of computation, such as whether P is equal to NP or whether NP is closed under complementation, can be translated into natural questions about mathematical logic.

We now prove one of the most striking results in descriptive complexity, discovered independently by Neil Immerman and Moshe Vardi in the early 1980s [Imm86] [Var82]. This result states that a problem is in P if and only if its solutions can be described using a certain strengthened form of first-order logic in which we can define new relations by induction. We will call this new logical language *inductive* first-order logic, and denote the corresponding descriptive complexity class $FO(LFP)$ (short for the *least fixed point* operator, which we will soon define). The Immerman-Vardi theorem thus states that $P = FO(LFP)$. Let us begin by motivating and defining inductive-first order logic and the descriptive complexity class

$FO(LFP)$.

4.1. **The Least Fixed Point Operator.**

Consider the problem PATH introduced in Section 3.3. In terms of computational complexity, we have seen that PATH $\in P$, and also that PATH is complete for the class NL. However, it is less clear how to characterize the *descriptive* complexity of PATH.

Question 4.1. *Is PATH in FO?*

Certainly any finite instance of PATH is in FO – that is, if we let $PATH_n$ be the problem of whether a graph G contains a path from s to t of length at most n, then:

$$PATH_n \equiv (\exists x_1)...(\exists x_{n-2})(E(s, x_1) \wedge E(x_1, x_2) \wedge ... \wedge E(x_{n-2}, t)).$$

Therefore $PATH_n \in FO$. However, it turns out that PATH itself is *not* in FO. A proof of this fact using the methods of descriptive complexity can be found in Section 6.2 of [Imm99]. It follows that the complexity class P is not contained in the descriptive class FO. As we will soon see, however, PATH is expressible in a slightly strengthened form of first-order logic where we add the power to define new relations by induction.

An example of a relation that can not be expressed in first-order logic, but can be defined inductively, is the *transitive closure* of a relation. For example, recall the vocabulary of graphs $\tau_g = \langle E^2 \rangle$. The transitive closure of E^2, which we denote E^*, is the set of pairs of vertices connected by a path in G. This relation can be inductively defined by

$$E^*(x, y) \equiv (x = y) \vee \exists z(E(x, z) \wedge E^*(z, y)).$$

We now present a more formal inductive definition of E^*, which can be abstracted to the general case of defining a new relation by induction. Let R be a binary relation variable, and consider the formula

$$\psi(R, x, y) \equiv x = y \vee \exists z(E(x, z) \wedge R(z, y)).$$

For any model M of vocabulary τ_g, let P^{A^2} denote the set of binary relations on the universe A of M. Then the formula ψ induces a map $\psi^M : P^{A^2} \to P^{A^2}$, given by

$$\psi^M(R) = \{(a, b) \mid M \vDash \psi(R, a, b)\}.$$

Let us call a map of relations $f : P^{A^2} \to P^{A^2}$ *monotone* if for all R, $S \in P^{A^2}$:

$$R \subseteq S \Leftrightarrow f(R) \subseteq f(S).$$

The following lemma, which should appear somewhat intuitive, confirms that ψ^M is monotone:

Lemma 4.2. *Let M be a finite model, R^k a k-ary relation symbol, and $\phi(R^k, x_1, ..., x_k)$ a first-order formula. If R^k occurs only* positively *within ϕ – that is, only within an even number of negation symbols – then the induced function ϕ^M is monotone.*

Letting $(\psi^M)^r$ denote ψ^M iterated r times, we can therefore consider the chain of inclusions

$$\emptyset \subseteq (\psi^M)(\emptyset) \subseteq (\psi^M)^2(\emptyset) \subseteq (\psi^M)^3(\emptyset) \subseteq ...$$

Observe that for the graph M, the relations in this chain of inclusions are given by

$$\begin{aligned}
(\psi^M)(\emptyset) &= \{(a,b) \in A^2 \mid distance(a,b) \leq 0\} \\
(\psi^M)^2(\emptyset) &= \{(a,b) \in A^2 \mid distance(a,b) \leq 1\} \\
(\psi^M)^3(\emptyset) &= \{(a,b) \in A^2 \mid distance(a,b) \leq 2\} \\
&\vdots \\
(\psi^M)^r(\emptyset) &= \{(a,b) \in A^2 \mid distance(a,b) \leq r-1\}.
\end{aligned}$$

Setting $n = |A|$, which is the number of vertices in the graph M, the transitive closure E^* can therefore be expressed as

$$E^* = (\psi^M)^n(\emptyset).$$

Observe that E^* is the minimal relation with the property that $\psi^M(E^*) = E^*$. We therefore call E^* the *least fixed point* of ψ^M, as formalized in the following definition.

Definition 4.3 (Least Fixed Point)**.** Let R^k be a relation symbol of arity k, and let $\phi(R^k, x_1, ..., x_k)$ be a first-order formula. Given any finite model M with universe A, let ϕ^M be the induced map defined as above. If there exists a minimal relation T such that $\phi^M(T) = T$, then we call T the *least fixed point* of ϕ^M, denoted $(LFP_{R^k x_1 ... x_k}\phi)$.

The Knaster-Tarski theorem provides us with a condition for the existence of the least fixed point, as well as a description thereof:

Theorem 4.4 (Knaster-Tarski)**.** *If ϕ^M is monotone, then the least fixed point T of ϕ^M exists. T is equal to $(\phi^M)^r(\emptyset)$, where r is minimal such that $(\phi^M)^{r+1}(\emptyset) = (\phi^M)^r(\emptyset)$ Furthermore, setting $n = |A|$, we have $r \leq n^k$.*

Proof. Because ϕ^M is monotone, we can consider the chain of inclusions:

$$\emptyset \subseteq (\psi^M)(\emptyset) \subseteq (\psi^M)^2(\emptyset) \subseteq ...$$

If $(\psi^M)^{i+1}(\emptyset)$ strictly contains $(\psi^M)^i(\emptyset)$, then it must contain at least one new k-tuple from the universe A. Because A is finite, there are only n^k possible k-tuples that can be formed from A. Therefore, for some $r \leq n^k$, $(\phi^M)^{r+1}(\emptyset) = (\phi^M)^r(\emptyset)$. It follows that $(\phi^M)^r$ is a fixed point of ϕ^M.

To complete the proof, we must show that $(\phi^M)^r$ is minimal with the desired property, i.e., $(\phi^M)^r \subseteq S$ for any other fixed point S of ϕ^M. We will prove by induction that for all i, $(\phi^M)^i(\emptyset) \subseteq S$.

The base case is that $i = 0$, so:

$$(\phi^M)^i(\emptyset) = \emptyset \subseteq S$$

For the inductive step, suppose that $(\phi^M)^i(\emptyset) \subseteq S$. Because ϕ^M is monotone:

$$(\phi^M)^{i+1}(\emptyset) = \phi^M((\phi^M)^i(\emptyset)) \subseteq \phi^M(S) = S$$

Therefore $(\phi^M)^r \subseteq S$, so ϕ^M_r is indeed the least fixed point of S.

□

We are now ready to formally define the descriptive complexity class $FO(LFP)$.

Definition 4.5 ($FO(LFP)$)**.** Define *inductive first-order logic* by adding the least fixed point operator LFP to first-order logic. That is, if $\phi(R^k, x_1, ..., x_k)$ is a first-order formula in which R^k appears only positively, then $(LFP_{R^k x_1 ... x_k}\phi)$ may be used as a new k-ary relation symbol. Let $FO(LFP)$ be the set of inductive first-order Boolean queries.

Example 4.6. Defining $\psi(R, x, y) \equiv x = y \vee \exists z(E(x, z) \wedge R(z, y))$ as previously, we have

$$PATH \equiv (LFP_{Rxy}\psi)(s, t).$$

Therefore, PATH $\in FO(LFP)$.

4.2. **The Immerman-Vardi Theorem.**

Having defined the descriptive complexity class $FO(LFP)$, we now prove its equality to the computation complexity class P.

Theorem 4.7 (Immerman-Vardi)**.** $FO(LFP) = P$.

Proof. Let us first consider the more general case of how to prove that a descriptive complexity class D is equal to a computational complexity class C. A common strategy for such proofs is as follows:

(1) Show that for any query $I \in D$, there exists an algorithm in C that evaluates I.

(2) Produce a Boolean query J which is complete for C under first-order reductions.

(3) Show that $J \in D$.

(4) Prove that D is closed under first-order reductions.

The inclusion $D \subseteq C$ follows immediately from Step 1. To see that these steps also imply $C \subseteq D$, let $A \in C$ be arbitrary. Step 2 implies that $A \leq_{fo} J$, and because $J \in D$ (Step 3) and D is closed under first-order reductions (Step 4), this in turn implies that $A \in D$. Thus $C \subseteq D$, and so $C = D$.

Our approach to proving the Immerman-Vardi theorem will follow this strategy. In particular, we will:

(1) Show that given any query $I \in FO(LFP)$, there exists a polynomial-time algorithm T that evaluates I.

(2) Define the query APATH, and show that it is complete for P under first-order reductions.

(3) Show that APATH $\in FO(LFP)$.

(4) Show that $FO(LFP)$ is complete under first-order reductions.

Let us begin with the first step. We will require the following lemma, which is a basic result in descriptive complexity:

Lemma 4.8. *$FO \subseteq L$.*

Proof (sketch). Any query $I : MOD(\tau) \to \{0, 1\}$ is given by a first-order formula ϕ. We therefore wish to construct a log-space Turing machine T such that for any input $A \in MOD(\tau)$, T accepts A if and only if $A \vDash \phi$. The proof will proceed by induction on k, the number of quantifiers (either existential or universal) occuring in ϕ.

Consider the base case in which $k = 0$, so that ϕ is a quantifier-free formula. For each atomic formula α in ϕ – that is, for each relation and function – T can evaluate α by reading the corresponding portion of the input A. We will not go through the details of this process, and instead refer the reader to Section 3.1 of [Imm99]. Once T has evaluated each atomic formula, it can determine whether $A \vDash \phi$ by computing the resulting finite Boolean combination.

For the inductive step, suppose that all first-order formulae with $k-1$ quantifiers can be computed by a log-space Turing machine, and let $\kappa \equiv (\exists x_1)(Q_2 x_2)...(Q_k x_k)\phi(x)$, where $Q_2, ..., Q_k$ are quantifiers. Let T be the logspace Turing machine that computes the formula $\psi(c) = (Q_2 x_2)...(Q_k x_k)\phi(x)$, where c is a constant symbol substituted for x_1. To compute κ, we construct the log-space Turing machine T' that cycles through all possible values of x_1, substitutes them for c, and runs T. T'

accepts its input if and only if some value of x_1 leads T to accept. In the case that the first quantifier in κ is universal, T' accepts if and only if every value of x_1 leads T to accept. Thus all first order formulae with k quantifiers can be computed by a log-space Turing machine; this completes both the inductive step and the proof itself.

□

Corollary 4.9. *$FO \subset P$.*

Proof. Recall from Lemma 3.4 that $L \subseteq P$. To see that the containment $FO \subset P$ is strict, recall that PATH $\in P$, whereas PATH $\notin FO$.

□

We are now ready to perform the first step in the proof of the Immerman-Vardi theorem, which we restate as a lemma:

Lemma 4.10. *Given any query $I \in FO(LFP)$, there exists a polynomial-time algorithm T that evaluates I.*

Proof. Any query $I \in FO(LFP)$ is given a by an inductive first-order formula ψ. Because any first-order formula can be evaluated in polynomial time, it will suffice to show that any least fixed point formula $L = (LFP_{R^k x_1 \ldots x_k} \phi)$ can be evaluated in polynomial time.

Let A be the input structure on which this formula is being evaluated, and let $n = |A|$. By Theorem 4.4, $L = (\phi^A)^{n^k}$. Therefore, we can evaluate L on A by evaluating the first-order formula ϕ at most n^k times. Because ϕ can be evaluated in polynomial time (Corollary 4.9), so too can L.

□

We now move on to the second step of our proof of Theorem 4.7, which is to produce a problem that is complete for the complexity class P under first-order reductions.

Recall that in Section 3.3, we showed that the computation of a non-deterministic log-space Turing machine can be modeled by a finite graph. This allowed us to prove that PATH, the problem of reachability on a graph, is complete for the class NL. Similarly, we saw that the computation of a deterministic Turing machine can be modeled by a deterministic graph, and thus DPATH, the problem of reachability on a determinstic graph, is complete for L.

In a similar vein, we now introduce the notion of an *alternating* graph, which allows us to model the computation of an alternating Turing machine. It will follow that APATH, the problem of reachability on an alternating graph, is complete for the class AL, which is in turn equal to P (cf. Theorem 3.5).

Definition 4.11 (Alternating Graph)**.** An *alternating graph* $G = (V, E, A)$ is a directed graph in which every vertex is labeled either universal or existential. $A \subseteq V$ denotes the set of universal vertices in V. The vocabulary of alternating graphs with designated vertices s and t is $\tau_{ag} = \langle E^2, A^1, s, t \rangle$.

Reachability in an alternating graph is defined in an analogous way to acceptance in an alternating Turing machine (Definition 3.1). Let us write $R_G(x, y)$ to denote that y is reachable from x in the alternating graph $G = (V, E, A)$. We define R_G

as the smallest relation on V^2 such that:

(1) For all $x \in V$, $R_G(x, x)$.
(2) If $x \in V$ is existential and there exists an edge $(x, z) \in E$ such that $R_G(z, y)$, then $R_G(x, y)$.
(3) If $x \in V$ is universal, and there is at least one edge leaving x, and $R_G(z, y)$ for *all* edges $(x, z) \in E$, then $R_G(x, y)$.

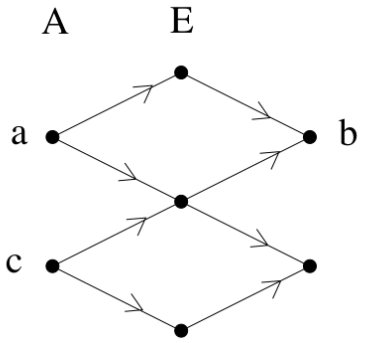

FIGURE 1. An alternating graph with universal vertices a and c.

Example 4.12. Let $G = MOD(\tau_{ag})$ be the alternating graph depicted in Figure 1, where a and c are universal vertices, while all other vertices are existential. Observe that b is reachable from a, because it is reachable from *every* node connected to a. On the other hand, b is *not* reachable from c. That is, $R_G(a, b)$ holds, but not $R_G(c, b)$.

Definition 4.13 (APATH)**.** APATH is the set of alternating graphs G in which t is reachable from s:

$$APATH = \{G \mid R_G(s, t)\}$$

.

Theorem 4.14. *APATH is complete for P under first-order reductions.*

Proof. (sketch) This proof is analogous to the proof of Theorem 3.10. Let $S \in MOD(\tau)$ be an arbitrary query decided by an alternating log-space Turing machine T. We wish to construct a first-order reduction $I_a : MOD(\sigma) \to MOD(\tau_{ag})$ such that for all $A \in MOD(\tau)$, T accepts A if and only if $I_a \in$ APATH.

Let I be identical to the first-order query defined in the proof of Theorem 3.10, with the additional stipulation that a vertex in $I_a(A)$ is universal if and only if the corresponding configuration in T is in a universal state. We refer the reader who wishes to verify that this property can be expressed in first-order logic to Section 3.4 of [Imm99].

In this way, $I_a(A)$ models the computation of T on the input A, so that T accepts A if and only if $I_a(A)$ contains a path from c_S to c_F. Therefore APATH is complete for the class $AL = P$ under first-order reductions.

□

The next step in our proof of Theorem 4.7 will be to show that APATH can be expressed in inductive first-order logic.

Lemma 4.15. *$APATH \in FO(LFP)$.*

Proof. Given a binary relation P, define the formula

$$\begin{aligned}\phi(P,x,y) \;\equiv\; x = y \vee ((\exists z)(E(x,z) \wedge P(z,y)) \\ \wedge\, (A(x) \rightarrow (\forall z)(E(x,z) \rightarrow P(z,y)))).\end{aligned}$$

We leave it as an exercise to check that:

$$R_a = (LFP_{Pxy}\phi)$$
$$APATH = (LFP_{Pxy}\phi)(s,t)$$

whence APATH $\in FO(LFP)$.

□

The only step in the proof of Theorem 4.7 which we will omit from this paper is showing that $FO(LFP)$ is closed under first-order reductions. This step is not overly difficult, but requires the use of the *dual* of a first-order query I, which is a map that functions similarly to an inverse of I. In fact, the dual allows us to see that virtually every naturally occuring complexity class (both descriptive and computation) is closed under first-order reductions. Intuitively speaking, this is because first-order queries are relatively weak, i.e., they do not profoundly change the structure of the input model. We refer the interested reader to Section 3.2 of [Imm99].

We have thus gone through all four steps in the proof of the Immerman-Vardi theorem. Step 1 (Lemma 4.10) immediately implies that $FO(LFP) \subseteq P$. The following three steps show that any problem $A \in P$ is first-order reducible to $APATH$, which is in turn in the class $FO(LFP)$. By the closure of $FO(LFP)$ under first-order reductions, we conclude that $A \in FO(LFP)$, and therefore $P \subseteq FO(LFP)$. This completes the proof that $P = FO(LFP)$.

□

5. A Descriptive Characterization of NP

The proof strategy outlined in Section 4.2 can be used to prove many more equalities between computational and descriptive complexity classes, and thus to illuminate the connection between mathematical logic and the theory of computation. For instance, Fagin's Theorem shows that a problem is in NP if and only if its solutions can be described using a restricted form of second-order logic called *existential*

second-order logic. This result was proved in 1973 by Ronald Fagin in his doctoral thesis, and marks the first major result in the field of descriptive complexity [Fag74].

We will not go through the proof of Fagin's theorem in this paper, but we will define existential second-order logic and the corresponding descriptive complexity class $SO\exists$. We will see how Fagin's theorem can be proved using a strategy similar to the one which we used to prove the Immerman-Vardi theorem, and how the two theorems together allow us to translate the P vs. NP problem into the language of mathematical logic.

Definition 5.1. (Second-Order Logic) Second-order logic consists of first-order logic together with the power to quantify over new relation variables. For instance, if ϕ is a first-order formula, the second-order formula $(\forall R^k)\phi$ may be used to mean that for any choice of k-ary relation R, ϕ holds true.

Definition 5.2. *Existential* second-order logic consists of first-order logic together with the power to quantify only existentially over new relation variables. That is, if ϕ is a first-order formula, $(\exists R^k)\phi$ may be used as a new formula, but *not* $(\forall R^k)\phi$. Similarly, *universal* second-order logic consists of first-order logic together with the power to quantify only universally over new relation variables.

Let SO be the set of second-order Boolean queries, and $SO\exists$ and $SO\forall$ the sets of existential and universal second-order Boolean queries, respectively. Observe that $SO\exists$ and $SO\forall$ are complementary to one another:

Lemma 5.3. *$SO\exists$ = co-$SO\forall$, and $SO\forall$ = co-$SO\exists$. That is, if a query I is in $SO\exists$, then $\bar{I} \in SO\forall$, while if I is in $SO\forall$, then $\bar{I} \in SO\exists$.*

An immediate corollary is that the open question of whether NP is closed under complementation is equivalent to the question of whether these two descriptive complexity classes are equal:

Corollary 5.4. *NP = co-NP if and only if $SO\exists = SO\forall$.*

A particularly important example of a problem in $SO\exists$ is the Boolean satisfiability problem SAT, which we now define.

Definition 5.5. Let us say that a Boolean formula is in *conjunctive normal form* (CNF) if it consists of a conjunction of clauses, each of which is a disjunction of literals. For instance, the formula

$$\phi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

is in conjunctive normal form. We can write the vocabulary of CNF Boolean formulae as $\tau_{cnf} = \langle P^2, N^2 \rangle$. In any Boolean formula which is a model $\phi = \langle A, I \rangle$ of τ_{cnf}, the universe A is a set of clauses and variables. The relation $I(P^2)(c, v)$ means that variable v occurs positively in clause c, while $I(N^2)(c, v)$ means that v occurs negatively in c.

Let SAT $\subset MOD(\tau_{cnf})$ be the set of CNF Boolean formulae which are *satisfiable*. That is, a formula $\phi \in MOD(\tau_{cnf})$ is in SAT if and only if there is some way to assign truth values to each of its variables such that the entire formula evaluates to

be true. For example, the above formula $\phi_0 \in$ SAT, because if we set x_1 true and x_2, x_3 false, ϕ_0 evaluates to be true.

Observe that SAT can be expressed as an existential second-order Boolean query:

$$\psi_{SAT} \equiv (\exists A)(\forall c)(\exists v)((P(c,v) \wedge A(v)) \vee (N(c,v) \wedge \neg A(v)))$$

The above formula states that there exists an assignment A of truth values to the variables of a given formula such that at least one literal in each clause evaluates as true. We thus see that SAT $\in SO\exists$. This fact offers a way to prove Fagin's theorem, which we now formally state.

Theorem 5.6. *(Fagin) NP = SO∃.*

Fagin's theorem can be proved using the following strategy, which parallels our approach to the proof of the Immerman-Vardi theorem:

(1) Prove that any existential second-order formula can be evaluated by a non-deterministic polynomial time algorithm.

(2) Prove that SAT is NP-complete under first-order reductions.

(3) Show that SAT is in $SO\exists$.

(4) Show that $SO\exists$ is closed under first-order reductions.

The proof of Step 1 is fairly straightforward: given an existential second-order formula $(\exists R_1^{a_1})...(\exists R_n^{a_n})\phi$, we construct an algorithm which non-deterministically guesses the relations $R_1, ..., R_n$ and accepts if and only if ϕ holds true for some choice of these relations. Step 4 is also a relatively intuitive result, and we have already proven Step 3.

This leaves Step 2 as by far the most difficult part of this approach to proving Fagin's Theorem. The fact that SAT is NP-complete under *polynomial-time* reductions was proven in the 1970s by Stephen Cook and Leonid Levin, and was a breakthrough result in complexity theory [Co71] [Lev73]. But because $FO \subset P$ (Corollary 4.9), the condition of completeness under *first-order* reductions is stronger than that of completeness under polynomial-time reductions. We refer the reader to Section 7.2 of [Imm99] for a rigorous proof of Fagin's theorem.

5.1. **The P vs. NP Question.**

The P vs. NP question (Question 1.3), first formulated in 1971 by Stephen Cook [Co71], asks whether every problem that can be decided by a non-deterministic polynomial-time algorithm can also be decided by a deterministic polynomial time algorithm – that is, whether $P = NP$.

This question is the most well-known instance of a range of related open questions in computational complexity theory. Letting $PSPACE$ and $EXPTIME$ be the sets of problems solvable in polynomial space and in exponential time, respectively, we have the chain of inclusions:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME.$$

It is striking that none of the above inclusions have been proven to be strict: even the fact that L is not equal to NP, for instance, remains an open conjecture. However, it has been proven that $P \neq EXPTIME$; this result follows immediately from the Time Hierarchy Theorem (cf. Section 9.1 of [Sip12]). This inequality implies that at least one of the last three inclusions presented above must be strict, but it remains unknown which one(s) are strict.

The methods of descriptive complexity have yet to resolve any of these open questions about the relations between computational complexity classes. However, descriptive complexity allows us to translate such questions into the language of mathematical logic. To see how the P vs. NP can be translated into a question about logic, we will require the following lemma:

Lemma 5.7. *If $P = NP$, then $SO\exists = SO$.*

Proof. Observe that the class P is closed under complementation: if a problem A is decided by a deterministic polynomial-time Turing machine T, we can simply switch the accepting and rejecting states in T to get a Turing machine T' which accepts $\bar{A}$ in polynomial time. Therefore, if $P = NP$, then NP is also closed under complementation, i.e. $NP =$ co-NP. By Corollary 5.4, this entails $SO\exists = SO\forall$. We leave it as an exercise to the reader to see that $SO = SO\exists \cup SO\forall$. We thus have $SO\exists = SO$.

□

The P vs. NP question is therefore equivalent to a very natural question about mathematical logic:

Question 5.8. *Is $FO(LFP)$ equal to SO? That is, can every second-order expressible property over finite structures also be expressed using inductive first-order logic?*

This reformulation of course does not resolve the problem of whether P is equal to NP – but it helps shed light on why this problem has been so difficult to resolve, and why it has emerged as the most prominent open question in the theory of computation. It shows that questions such as whether P equals NP are not merely questions about the power of computation: they are also questions about the structure of mathematical knowledge as a whole.

6. Acknowledgements

Thank you to my mentor Olga Medrano Martin del Campo for guiding and supporting this project since its inception, as well as for creating a sense of camaraderie and community within our small group of students. Thank you also to Isabella Scott for sparking my interest in mathematical logic, and to Professor Maryanthe

Malliaris for meeting with me to discuss potential topics for a research project in logic. Finally, thank you to Peter May for putting together this excellent program, which has allowed me to greatly develop and deepen my mathematical interests while still an undergraduate student.

References

[Chu36] Church, Alonzo. "An unsolvable problem of elementary number theory." American Journal of Mathematics, 58 (1936), pp 345–363.

[Cha88] Chandra, Ashok K.; Kozen, Dexter C.; Stockmeyer, Larry J. "Alternation." Journal of the ACM. 28 (1988): pp 114–133.

[Co71] Cook, Stephen. "The complexity of theorem proving procedures." Proceedings of the Third Annual ACM Symposium on Theory of Computing (1971), pp. 151–158.

[Fag74] Fagin, Ronald. "Generalized first-order spectra and polynomial-time recognizable sets." Complexity of Computation, ed. R. Karp, SIAM-AMS Proceedings, Vol. 7 (1974), pp 43-73.

[Hil28] Hilbert, David; Ackermann, Wilhelm. *Grundzüge der Theoretischen Logik*. Berlin: Springer, 1928.

[Imm86] Immerman, Neil. "Relational queries computable in polynomial time." Information and Control, 68 (1–3) (1986), pp 86–104.

[Imm88] Immerman, Neil. "Nondeterministic space is closed under complementation." SIAM Journal on Computing, 17 (1988), pp 935–938.

[Imm99] Immerman, Neil. *Descriptive Complexity*. New York: Springer-Verlag, 1999.

[Lev73] Levin, Leonid. "Универсальные задачи перебора." Problems of Information Transmission. 9 (3) (1973): pp 115–116.

[Sip12] Sipser, Michael. *An Introduction to the Theory of Computation*. Boston: Cengage Learning, 3 ed., 2012.

[Tur36] Turing, Alan. "On Computable Numbers, with an Application to the Entscheidungsproblem." Proceedings of the London Mathematical Society, Series 2, 42 (1936–7), pp 230–265.

[Var82] Vardi, Moshe. "The Complexity of Relational Query Languages." STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing (1982) pp. 137–146.