Blum axioms and nondeterministic computation of functions

Tiago Royer[1], **Jerusa Marchi**[1]

[1] Universidade Federal de Santa Catarina — Departamento de Informática e Estatística

royertiago@gmail.com, jerusa.marchi@ufsc.br

Abstract. *In his doctoral thesis, Manuel Blum proposed two axioms for complexity measures that allows us to talk about complexity in an axiomatic manner. His axioms does not even specify the machine model — it just requires it to satisfy some properties. Blum axioms, however, are defined in the context of function computation. This restriction is easy to implement with deterministic machines, since there is only one output for a given input, but how can a nondeterministic Turing machine compute a function? This paper surveys techniques to associate nondeterministic machines with functions and analyze how they interact with computational complexity.*

1. Introduction

In Theory of Computation, we usually use languages to mathematically model problems in the real world. Decision problems ("yes/no") are mapped to languages in a very natural way, by just putting every "yes" instance in the language, and leaving the rest out. Search problems usually are rewritten as a decision problem, and then this problem is converted to a language. For instance, the task of finding a satisfying assignment for a Boolean formula is reinterpreted as the task of deciding whether such an assignment exists, and this task is then converted to a language — in this example we have SAT, the Boolean satisfiability problem. This does the trick when it comes to proving that something is hard; for instance, if we show that the decision problem is NP-hard or undecidable, then intuitively the corresponding search problem must be at least as hard. Therefore, concepts like "decidable", "NP-complete", "polynomial-time decidable" arise naturally in the context of decision problems.

However, even from the theoretical standpoint, it is useful to extend these concepts to functions. For instance, the concept of polynomial-time computable functions are required to define Karp reductions [Arora and Barak 2009, p. 42]. For single-tape deterministic Turing machines, this definition is easy to extend: A function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be *polynomial-time computable* if there is a Turing machine M and a polynomial p such that, when given the string x as input, M halts with $f(x)$ in its tape within $p(|x|)$ steps.

Most extensions to this basic model, like the use of several tracks, or several tapes, or multidimensional tapes, can be easily incorporated in the definition of polynomial-time computable. Except nondeterminism. We hit a wall right in the start: how does a nondeterministic machine computes a function in the first place?

This paper presents several attempts to establish how a nondeterministic Turing machine could compute a function, and to extend the concept of "polynomial-time computable" under each definition. Section 2 sets reasonableness criteria to both the definition

of function computation and complexity of the computation. Section 3, which contains the several definitions, presents and criticizes Hopcroft and Ullman's and Goldreich's definitions (sections 3.1 and 3.2), proposes one possible definition that meet the quality standards (section 3.3), and shows other two definitions, by Krentel (section 3.4) and Valiant (section 3.5), that, altough were not proposed in the context of general nondeterministic computation of functions, also meet the quality standards and sidesteps the problems with the proposed definition (presented in section 3.3.1). The development of this project is mentioned in section 5, after the concluding remarks (section 4).

2. Our approach: Gödel numberings and Blum axioms

In this paper, we will try to associate nondeterministic computation with partial recursive functions, in some well-behaved manner, and preserving the apparent[1] exponential speed-up present in nondeterministic deciders. The concept of Gödel numberings (section 2.1) captures the notion of "well-behaved". The Blum axioms (section 2.2) capture the notions of computational complexity. We thus will demand the definitions to satisfy the requirements of Gödel numberings and Blum axioms.

We are restricting ourselves to using single-valued functions, but there are alternative approaches. Complexity classes like NPVM (see, for example, the paper of Selman [Selman 1994, p. 359]) are defined using *multivalued functions*, which are allowed to return several values for a single input. Another approach is to use *function problems* associated to problems in NP, where the machine is required to return any certificate for the given instance, or to reject the input if it is not in the language [Papadimitriou 1994, p. 229]. We will not consider these approaches here.

2.1. Reasonableness criterion: acceptable Gödel numberings

One of the most important theoretical results concerning Turing machines is the existence of undecidable problems. Namely, the *halting problem* (the task of deciding whether a given Turing machine will halt on a given input) cannot be solved by Turing machines [Arora and Barak 2009, p. 23]. The formalization (and proof) of this fact requires the definition of some sort of *encoding*; since Turing machines can only reason about strings, we need somehow to encode Turing machines into strings, to be able to pose the halting problem as a language question.

Each Turing machine can be associated to the corresponding partial recursive function it computes. There are several ways to encode Turing machines as strings, but what is most important about them is that they allow us to manipulate these partial recursive functions *indirectly* — partial recursive functions are (potentially) infinite objects, so we cannot write them down on a Turing machine tape, but we *can* write the encoding of a Turing machine that compute these functions.

Therefore, these encodings provide a way to associate a string (which is a finite, manipulable object) with a partial recursive function (which is an infinite, mathematical, "untouchable" object). Encodings are *enumerations* of all partial recursive functions.

[1] It is apparent in the sense that, if proven would show $\mathsf{P} \neq \mathsf{NP}$, and if disproved (showing a polynomial slowdown is the best we can do) would show $\mathsf{P} = \mathsf{NP}$. Although most researchers expect the former to be the case [Gasarch 2012, p. 54], as Papadimitriou noted [Papadimitriou 1994, p. 412], in the absence of a proof that $\mathsf{P} \neq \mathsf{NP}$ we should not be too emphatic in stating the simulation *require* (as opposed to *seemingly require*) exponential slowdown.

One important feature about the standard encodings of deterministic Turing machines is the Universal Turing Machine Theorem: the existence of a partial recursive function $U : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that, if w encodes the machine M, then $U(m,x)$ is the result of running the machine M on x. A machines that compute U is called *universal Turing machine*.

The concept of *acceptable Gödel numbering* ([Rogers 1987, p. 41], [Blum 1967, p. 324]) encompasses the existence of universal machines and a little more. We will use it as our reasonableness criteria to our definitions.

Definition 1. Let $\mathcal{P}$ be the set of all partial recursive functions. An *acceptable Gödel numbering* is a function $\phi : \{0,1\}^* \to \mathcal{P}$, that associates each string (or program[2]) $w \in \{0,1\}^*$ to a function $\phi_w \in \mathcal{P}$, that satisfies

1. ϕ is surjective; that is, every partial recursive function $f \in \mathcal{P}$ has a program $w \in \{0,1\}^*$ such that $\phi_w = f$;
2. There is a partial recursive function $U : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that, for every w and x, $U(w,x)$ is defined if and only if $\phi_w(x)$ is defined, and, in this case,
$$U(w,x) = \phi_w(x);$$
3. There is a total recursive function $\sigma : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that, for every y, $\phi_w(x,y)$ is defined if and only if $\phi_{\sigma(w,x)}(y)$ is defined, and, in this case,
$$\phi_w(x,y) = \phi_{\sigma(w,x)}(y).$$
This concept captures the intuitive notion of "well-behaved numbering".

Condition 1 guarantees that the image of ϕ is all of $\mathcal{P}$, so that the numbering neither "forgets" a partial recursive function nor generate a function that is not partial recursive. Condition 2 is the universal Turing machine theorem.

Condition 3 is the "little more" we mentioned earlier. It is known as the S_{mn} theorem [Rogers 1987, p. 24]. Essentially, given a partial recursive function of two variables, we can obtain a partial recursive function of one variable by fixing the first argument. The function σ provides a systematic way of doing this: given a description w of a partial recursive function of two variables and the value x to be fixed as the first variable, $\sigma(w,x)$ is a machine that computes this new partial recursive function.

Example 2. Any encoding of deterministic Turing machines as a binary string yields an acceptable numbering of the recursive functions.

Example 3. Any programming language can be understood as an acceptable Gödel numbering. For example, if we restrict a `C` program to perform input and output only using the standard input and standard output (that is, forbid interactions with the user, file reading, GUIs, access to system clock, etc.), the resulting program will map a binary input to a binary output, characterizing a partial recursive function. Thus, we can see a `C` compiler as an implementation of an acceptable Gödel numbering.[3]

[2] In texts like Rogers' [Rogers 1987], the partial recursive functions have the naturals as domain and codomain (that is, they are of the form $f : \mathbb{N} \to \mathbb{N}$), and Gödel numberings associates natural numbers with recursive functions. In this paper we will work with binary strings instead of numbers, which will simplify the definition of complexity classes and allows us to think the string w as a *program* for ϕ_w (see example 3).

[3] Note we are ignoring here issues like compilation and run-time errors. These can be dealt with as in the case of Turing machines: any invalid program will signify some fixed partial recursive function (say, the function that is defined nowhere); and any invalid step in computation makes the function to be not defined on that input.

One important theorem that can be proven using acceptable Gödel numberings alone is the recursion theorem [Rogers 1987, p. 181]. It states that, if ϕ is any acceptable Gödel numbering and f is any total recursive function, then there is some program w_0 such that $\phi_{w_0} = \phi_{f(w_0)}$. That is, if f is any systematic transformation on programs, there is a program w (a fixed point for f) whose meaning under ϕ is unchanged. The recursion theorem can be used, for example, to show the existence of quines (programs whose output are their own source codes) in any Turing-complete programming language: choose f to be the function that, given a program w, returns another program $f(w)$ that prints the string w when run.[4] Then, by the recursion theorem, there is some program w_0 that is equivalent to its transformed version $f(w_0)$; thus, w_0 already writes the string w_0, its own source code. Therefore, any programming language has quines [Kozen 2006, p. 227].

2.2. Efficiency criterion: Blum axioms

The *complexity* of a computation is how much of a resource that is invested in that computation [Hopcroft and Ullman 1979, p. 285]. For each model of computation and each resource under that model, we can establish a *complexity measure* concerning that resource. This section is devoted to formalizing this notion. As we did with the machine encodings, we will impose some restrictions on what can be a complexity measure to be able to manipulate it (at least indirectly). In our case, we will use Blum axioms [Blum 1967, p. 324].

Definition 4. Given an acceptable Gödel numbering ϕ, a *complexity measure* for ϕ is a function $\Phi : \Sigma^* \times \Sigma^* \to \mathbb{N}$ of two variables that satisfies [Blum 1967, p. 324]:

1. For every w and x, $\phi_w(x)$ exists if and only if $\Phi(w, x)$ exists; and
2. For every string w, x and every natural number k, the predicate "$\Phi(w, x) = k$?" is decidable.

$\Phi(w, x)$ is the complexity of computing $\phi_w(x)$ using the program w. The first axiom says that it only makes sense to talk about the complexity of a computation that ends. The second axiom gives minimum tools to manipulate Φ indirectly, in the same manner we require ϕ to be an *acceptable* Gödel numbering.

Example 5. The standard measures of time and space can be constructed over the acceptable numbering of example 2. They are, respectively, the number of moves and tape cells scanned before halting. Leave the complexity undefined if the machine does not halt — this satisfies the first axiom. The predicate of the second axiom is simple for time complexity; for space complexity, we must keep the whole history of computation to make sure the machine does not loop in a limited amount of space (because the complexity is not defined in this case).

Definition 6. Given a complexity measure Φ for an acceptable Gödel numbering ϕ and a total recursive function $f : \mathbb{N} \to \mathbb{N}$, define the *complexity class* $\mathcal{C}_\Phi(f)$ by ([Kozen 2006, p. 232])

$$\mathcal{C}_\Phi(f) = \{\phi_w \mid \Phi(w, x) \leq f(|x|) \text{ almost everywhere}^5\}.$$

[4] For Turing machines, for instance, we can encode the bits of w in the transition table of $f(w)$.

[5] *Almost everywhere* means that the inequality $\Phi_i(x) \leq f(|x|)$ holds for all but a finite number of different x.

This formalizes the notion of complexity class. There are some important theorems concerning complexity classes under the Blum axioms; we mention only the Union Theorem [Kozen 2006, p. 234]. It states that, if $\{f_i\}$ is any recursive list of increasing functions such that $f_i(n) \leq f_{i+1}(n)$ for all x (that is, each f_i is greater than the previous), then there is some function g such that

$$\mathcal{C}_\Phi(g) = \bigcup_{i \in \mathbb{N}} \mathcal{C}_\Phi(f_i).$$

That is, the class $\mathcal{C}_\Phi(g)$ contains *exactly* all functions present in the classes $\mathcal{C}_\Phi(f_i)$. Choosing Φ as the time complexity and $f_i(n) = n^i$ (the polynomial functions), the union in the right is exactly the class P, the problems solvable in polynomial time. By the Union Theorem, P is $\mathcal{C}_\Phi(g)$ for some recursive function g, so, even though P does not have an easily specifiable bounding function, such function exists nevertheless.

3. Nondeterministic computation of functions

This section surveys several approaches for defining nondeterministic computation of functions.

Sections 2.1 and 2.2 introduced the concept of acceptable Gödel numbering, the Blum axioms, and its complexity classes. As we want to regard these concepts as "quality requirements" for the definitions, we will analyze each of them to see whether they define acceptable Gödel numberings, the analogous time complexity satisfies Blum axioms, and that the characteristic function[6] of the Boolean satisfiability problem can be solved in "polynomial time" according to that complexity measure. (This last requirement expresses that the definition preserves the exponential speed-up that nondeterminism gives to deciders.)

3.1. Hopcroft-Ullman's definition

Definition 7 (Hopcroft and Ullman's definition[7])**.** If w is an encoding for the Turing machine M, we say that $\phi_w(x) = y$ if and only if, when processing x, there is some branch of M that halts with y in the tape, and there is no branch that halts with some $z \neq x$ in the tape.

The problem with their definition is that $\phi_w(x)$ is allowed to be defined, even if some branch of computation does not halt. This allows us to solve the complement of the halting problem.

Proposition 8. Define the partial function $f : \Sigma^* \times \Sigma^* \to \{0, 1\}$ by

$$f(w, x) = \begin{cases} 1, & \text{if the machine } w \text{ does not halt on } x. \\ \text{undefined}, & \text{if } w \text{ halts on } x. \end{cases}$$

This function can be computed by a nondeterministic Turing machine under Hopcroft and Ullman's definition.

[6]The characteristic function of a set A is the function 1_A defined to be 1 for $x \in A$ and 0 for $x \notin A$.

[7]Hopcroft and Ullman's original definition [Hopcroft and Ullman 1979, p. 313] was defined in the context of computation of integer functions. We are rephrasing here in terms of strings, but keeping the relation they imposed on the execution branches.

Proof. On input (w, x), create two branches of computation. On the first, write 1 on the tape and halt. On the second, simulate the universal Turing machine U in the input, and if U halts, write 0 on the tape and halt too.

If w halts on x, there will be two halting branches of computation, each writing a different value in the tape, so, by definition, the function is not defined on this input. If w never halts, then only the first branch will halt (and with 1 written on the tape), so the function is defined on this input and its value is 1. □

Therefore, under Hopcroft and Ullman's definition, we can compute some non-computable functions, violating the requirement 1 of Gödel numberings.

We can try to fix this definition by forcing all branches to halt; but then, as all branches are required to return the same value, the machine will be (almost) deterministic. Therefore, if there is a machine M that computes the characteristic function of the satisfiability problem in nondeterministic polynomial time, we could simulate M on a given input, choosing (say) always the first option when confronted with nondeterminism. If this branch of computation returns 1, then every branch returns 1 and the input is satisfiable; if this branch returns 0, every branch returns 0 and the input is unsatisfiable. We thus could solve SAT in *deterministic* polynomial time. So, with this restriction, we lose the apparent exponential speed-up in computation time we have when using nondeterminism.

3.2. Goldreich's definition

Definition 9 (Goldreich's definition)**.** Let $\perp$ be some special symbol not in $\{0,1\}^*$. (This symbol will represents "don't know".) A nondeterministic machine M computes the function f if, when processing the input x, both the following conditions hold [Goldreich 2008, p. 168]:

- Every branch of M halts and outputs either $f(x)$ or $\perp$.
- At least one branch of M halts with $f(x)$ on the tape.

The extra symbol $\perp$ sidesteps the problems of a branch looping forever. This allows us to mechanically convert nondeterministic machines under Goldreich's definition to deterministic machines via simulation — the deterministic machine just need to simulate all branches until completion, to actually be sure every branch halts; if some branch do not halt, then the simulating machine will not halt either, but the function is not defined in this case, so this behavior is correct.

Thus, we only enumerate computable functions. To show the universal machine theorem and the S_{mn}, we can simply first convert the machine in question to a deterministic machine and use their theorems; thus, this definition yields an acceptable Gödel numbering. And, by counting the number of steps of the deepest branch, we have a Blum complexity measure.

So, Goldreich's definition defines an acceptable Gödel numbering and we can form a complexity measure that satisfies Blum axioms. But, again, we have trouble with the "exponential speed-up" requirement. For instance, a machine trying to solve the satisfiability problem would correctly return 1 for a satisfiable instance, but no branch can write a 0 alone because it cannot be sure that instance is unsatisfiable — so, the function would be undefined for unsatisfiable formulas.

That is, unless $\mathsf{NP} = \mathsf{coNP}$.

Proposition 10. If there is a nondeterministic Turing machine that computes in polynomial time, according to Goldreich's definition, the characteristic function of the satisfiability problem, then $\mathsf{NP} = \mathsf{coNP}$.[8]

Proof. Suppose M is the machine that computes SAT's characteristic function, under Goldreich's definition, in polynomial time.

The characteristic function of SAT is a total function, and its only outputs are 0 and 1. Therefore, in every computation of M, there is at least one branch that writes something different that $\perp$ on the tape, and whatever it writes is the correct answer. So, if we convert this to a nondeterministic decider and invert the output (branches that write 0 will accept the input and vice-versa; branches that write $\perp$ always reject), any satisfiable formula will be rejected, because no branch of M ever writes 0 on the tape for these formulas; and any unsatisfiable formula will be accepted, because at least one branch of M writes 0 for these formulas. Thus, we can decide $\overline{\mathsf{SAT}}$, the complement of SAT.

Since $\overline{\mathsf{SAT}}$ is coNP-complete, the existence of such a machine M would show that $\mathsf{coNP} \subseteq \mathsf{NP}$, and this implies that $\mathsf{coNP} = \mathsf{NP}$.[9] □

Therefore, even though Goldreich's definition yields an acceptable Gödel numbering and a Blum complexity measure, it also have trouble in transposing the exponential speed-up we have using nondeterminism.

3.3. Proposed definition

Analyzing the problems with the first two definitions, we know the nondeterministic machine must be allowed to return several values (one for each branch) and somehow pick only one to be the value of the function.

If M is any deterministic decider for the language L, we can create a machine that computes the characteristic function of L by running M and returning 1 if M accepted and 0 if it rejected.

If we apply this transformation to a nondeterministic machine that recognizes the Boolean satisfiability problem, then, when running this machine, we have three possible results.

- For tautological formulas, the set of possible answers is only $\{1\}$.
- For contradictory formulas, the set of answers is $\{0\}$.
- For satisfiable formulas that are not tautological, we have both values: $\{0, 1\}$.

We want the return value for all satisfiable formulas to be 1 and for unsatisfiable formulas to be 0. Note that this corresponds exactly to the maximum value of each set; so, our definition of nondeterministic computation of functions will preserve exactly this behavior.

Definition 11 (Proposed definition)**.** Let M be a nondeterministic Turing machine, and x an input. If every branch of M halts when processing x, the value of the function

[8] $\mathsf{NP} = \mathsf{coNP}$ implies that $\mathsf{NP} = \mathsf{PH}$; that is, the polynomial hierarchy collapses to the first level [Kozen 2006, p. 280].

[9] To see why $\mathsf{coNP} \subseteq \mathsf{NP}$ implies $\mathsf{coNP} = \mathsf{NP}$, pick a language $L \in \mathsf{NP}$. Its complement $\overline{L}$ is in coNP, by the definition of coNP. But by hypothesis, $\overline{L}$ is in NP, so, by the definition of coNP, its complement, $\overline{\overline{L}} = L$ is in coNP, thus showing $\mathsf{coNP} = \mathsf{NP}$.

computed by M on x is the lexicographically maximum between all strings written in the computation branches. If some branch does not halt, leave the function undefined in x.

The same reasoning of Goldreich's definition applies here; therefore, we have an acceptable Gödel numbering, and counting steps of the deepest branch (as in Goldreich's definition) yields a Blum complexity measure.

And, using exactly the algorithm mentioned above, we can compute the characteristic function of the satisfiability problem under linear time; thus, this definition meets all the requirements proposed in the beginning of section 3.

3.3.1. Extending NP-completeness

The last definition provides an extension of the class NP to function computation, so the next step is extend the concept of NP-completeness.

Call FNP the class of all functions computable in polynomial time, under our definition[10]. A language L is NP-complete if both $L \in \mathsf{NP}$ and every language in NP reduces to L; that is, for every language $L' \in \mathsf{NP}$, there is a polynomial-time computable function f such that, for every x,

$$x \in L' \text{ if and only if } f(x) \in L.$$

(We are using Karp reductions here [Arora and Barak 2009, p. 42].) If we rephrase in terms of the characteristic functions 1_L and $1_{L'}$, of L and L', respectively, we have $1_{L'}(x) = 1_L(f(x))$ for all x. We will generalize specifically this equation to define FNP-completeness.

Definition 12. A function f is FNP-complete if $f \in \mathsf{FNP}$ and, for every function g in FNP, there is a polynomial-time computable function h such that

$$g(x) = f(h(x))$$

for every $x \in \{0,1\}^*$.

We can construct a FNP-complete function based on the halting problem: define $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^*$ such that $f(w,n)$ is the lexicographically greatest value written by any branch of w after running for n steps. Such functions are FNP-complete because they simulate Turing machines directly. However, this definition is very rigid and allow for few functions to be FNP-complete; the requirement of directly returning the output of the function g above restricts the class of FNP-complete functions to functions that perform simulations.

3.4. Krentel's OptP class

The next two authors were not specifically concerned with nondeterministic computation of functions, but rather in generalizing the NP class for functions. Therefore, the corresponding notion of NP-completeness behave better than our proposed generalization.

Krentel's definition, in particular, are very similar to ours.

[10]Note that this definition is different from the one given by Papadimitriou [Papadimitriou 1994, p 229].

Definition 13. A function $f : \{0,1\}^* \to \mathbb{N}$ is in OptP if there is some nondeterministic Turing machine M such that [Krentel 1988, p. 493]:

- For every input, every branch of M halts within a polynomial number of steps and writes in its tape a number in binary; and
- Either, for all x, the largest number written by M on x is $f(x)$, or, for all x, the smallest number written by M on x is $f(x)$.

Therefore, Krentel's OptP class contains the optimization problems that can be "solved" in polynomial time by nondeterministic Turing machines. (Note we must always take the maximum value, or always take the minimum value.) The definition of OptP-completeness, however, is significantly different.

Definition 14. A function f is OptP-complete if $f \in \mathsf{OptP}$ and, for every function $g \in \mathsf{OptP}$, there are two functions $T_1 : \{0,1\}^* \to \{0,1\}^*$ and $T_2 : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^*$, both computable in deterministic polynomial time, such that, for all x,

$$g(x) = T_2\big(x, f(T_1(x))\big).$$

That is, besides the preprocessing function T_1, we are allowed to make a post-processing which have access both to the input x and to the "reduced output" $f(T_1(x))$. Under this definition, the traveling salesperson problem and $0-1$ integer linear programming are both OptP-complete [Krentel 1988, p 495].

3.5. Valiant's #P class

Definition 15 (Valiant's definition)**.** A nondeterministic machine M computes a function $f : \{0,1\}^* \to \mathbb{N}$ if, for every input x, M halts on every branch[11] and the number of accepting branches of computation is $f(x)$ [Valiant 1979, p. 191].

The class #P is the set of functions computed in polynomial time, under Valiant's definition. #P-completeness is defined through oracles; that is, a function f is #P-complete if $f \in \#\mathsf{P}$ and $\#\mathsf{P} \subseteq \mathsf{FP}^f$ (that is, every function of #P can be computed by a deterministic machine with access to an oracle that computes f) [Valiant 1979, p. 191].

Besides problems like counting the number of satisfying assignments for a given formula, Valiant also proves less trivial problems are #P-complete, like the task of computing the permanent of a matrix [Valiant 1979, p. 194]. Therefore, this definition is also flexible, like Krentel's.

Krentel note that both his and Valiant's definition arise from applying an associative operator to all values returned in the branches of computation — maximum/minimum in Krentel's case, and cardinality (counting) in Valiant's case. Therefore, using other associative operations yield alternative definitions of nondeterministic function computation [Krentel 1988, p. 493].

4. Concluding Remarks

As we have seen, it is possible to associate nondeterministic Turing machines with function computation, although the association is not so straightforward as with deterministic machines.

[11] Valiant did not include the requirement of halting in his definition, but we will include it to avoid having the same problems of Hopcroft and Ullman's definition.

Through the use of the notions of acceptable Gödel numberings and Blum axioms, we formalized the notion of "reasonable" definition for nondeterministic function computation, and by consequence we formalized what is a "generalization of NP to functions". Krentel's and Valiant's definitions, besides generalizing NP, in particular, also allow for a flexible generalization of NP-completeness — that is, they allow for interesting problems to be "functionally NP-complete", under each definition.

5. Project Development

Tiago Royer studied this problem of associating a nondeterministic machine with a function in his undergraduate thesis[12], under the supervision of Jerusa Marchi. After finding Hopcroft and Ullman's definition (section 3.1) and noting it yields a computation that is very close to be deterministic, he devised the definition of section 3.3 (using the concept of acceptable Gödel numberings and Blum axioms to be sure his definition would not bee too unreasonable). As noted in section 3.3.1, his generalization to NP and NP-completeness are very rigid.

Krentel's and Valiant's definitions (which were discovered later in the project), although not created specifically to associate nondeterministic machine with functions, provides a more elegant generalization of NP and NP-completeness; this paper summarize all these findings under the light of Gödel numberings and Blum axioms.

References

Arora, S. and Barak, B. (2009). *Computational Complexity - A Modern Approach*. Cambridge University Press.

Blum, M. (1967). A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336.

Gasarch, W. I. (2012). Guest column: the second P =? NP poll. *SIGACT News*, 43(2):53–77.

Goldreich, O. (2008). *Computational complexity - a conceptual perspective*. Cambridge University Press.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Kozen, D. (2006). *Theory of Computation*. Texts in Computer Science. Springer.

Krentel, M. W. (1988). The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490 – 509.

Papadimitriou, C. H. (1994). *Computational complexity*. Addison-Wesley.

Rogers, Jr., H. (1987). *Theory of recursive functions and effective computability (Reprint from 1967)*. MIT Press, Cambridge, MA, USA.

Selman, A. L. (1994). A taxonomy of complexity classes of functions. *Journal of Computer and System Sciences*, 48(2):357–381.

Valiant, L. G. (1979). The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201.

[12]The project was developed while he was an undergraduate student at Federal University of Santa Catarina (UFSC). Currently, he is pursuing his master's degree at the University of São Paulo (USP).