Teaching Nondeterminism

Dana Richards
George Mason University

Abstract

Nondeterminism is an important topic that is difficult to teach. We give a plan that starts with a new abstraction of nondeterminism. We present language models first and foremost. The use of pseudo-code augmented by a new construct makes it easier to follow.

1 Introduction

The subject of nondeterminism is largely ignored in the Computer Science curriculum. It has its place in algorithms and software engineering, but it is in a Theory of Computation course that a student is likely to encounter it. When it is met it is usually minimized as a necessary but unwelcome topic [1]. Below we discuss the ambiguity of "nondeterminism" but here we focus on its role in the Theory of Computation.

This paper proposes some simple methodologies for teaching nondeterminism. First we propose putting nondeterminism up front in a Theory course and later we focus on the abstraction behind nondeterminism. In such a course there is an ebb and flow between Automata Theory and Formal Languages. Instead we use the dichotomy between defining a language computationally with either *recognition* or *generative* models. A recognition model is a mechanistic/algorithmic specification that accepts a string as input and decides[1] membership in the language. A generative model is a mechanistic specification for generating/building all string in the language. Examples of generative models include grammatical models and regular expressions.

The key observation is that generative models are fundamentally understood as nondeterministic models. Deterministic variants are the exception. Students seem to have no difficulty grasping generative models; the nondeterministic aspects present no hurdle. A string x is in the language L if and only if x *can* be generated, that is, *there exists* a sequence of steps in the generative process that produces x. Examples are given below.

2 The Meaning of Nondeterminism

Armoni and Ben-ari [2] give a detailed and historical survey of nondeterminism. Unfortunately they give equal weight to disparate meanings. When teaching "heaps" in an algorithms course you would not consider teaching the other meaning of "heaps" as found in an operating systems course. Similarly, students would be ill-served by teaching confounding meanings in our case. It is true that Dijkstra's guarded commands were introduced with "nondeterminism" but in fact what is meant is that correctness does not depend on which of several enabled statements are executed; "flexible" better describes this universally quantified condition. It is also true the term arises in discussions on concurrency, where the ramifications of asynchronous executions would be better described as "indeterminate," which means unpredictable. Quantum computing uses the language of physics, where the ideas are really probabilistic. We would argue that the automaton/algorithms definition was first and should be given priority.

[1] Recursively enumerable languages notwithstanding.

The seminal work on nondeterministic algorithms by Floyd [3] states clearly in the first sentence: "Nondeterministic algorithms are conceptual devices to simplify the design of back-tracking algorithms by allowing considerations of program bookkeeping required for back-tracking to be ignored." An even earlier work on finite automata by Rabin and Scott [4] introduces the topic with "A nondeterministic automaton has, at each stage of its operation, several choices of possible actions. This versatility enables us to construct very powerful automata using only a small number of internal states." In both cases it is the utility of hiding details that is the motivation.

Armoni and Ben-Ari, and most authors, emphasize the idea of an abstraction of sequential execution that is "unpredictable and inconsistent." We contend that this is just wrong. **Nondeterminism is an abstraction of search.** It is not an umbrella term for all notions of search, just those that make a set of choices. Floyd is explicit and it is implicit in Rabin and Scott. Floyd would not care if backtracking over a search tree was replaced massively-parallel search, with fan-in of the results (even though Floyd does give an explicit simulation of back-tracking). Rabin and Scott, in their familiar proof of simulation using a deterministic finite automaton, instead rely on precomputing a table of the how the search would go, level-after-level, down a massively parallel search tree.

We focus here on *decision problems* as is the case in a Theory course. As such, a computation will either *succeed* or *fail*. If we wanted to focus on optimization problems, the notion of an abstraction of search still hold, but with different examples and details. Even optimization algorithms have a binary termination condition.

The point is nondeterminism is not meant to be understood as sequential choices, but instead as a set of choices as a whole. This actually very natural. For example, "We can get there in 20 minutes" or "The sudoku puzzle has a solution" are understood as nondeterministic statements. They both assert that there is a set of choices (perhaps more that one) that solve the problem, i.e. a search will succeed. The first is qualitatively different since the set of choices are meant to be executed/interpreted sequentially ("turn her and then turn there") but the choices are not necessarily made that way. The second has no such interpretation; compare it to the choice of a certificate for the verification algorithm approach to nondeterminism [5].

The ubiquity of the von Neumann model causes us to fixate on making the abstraction of nondeterminism all about how "unpredictable" choices are made. Historically, we admit, that Turing machines were inherently deterministic and led directly to the von Neumann model. However if a grammatical model of recursive languages had been studied first, it is plausible that Turing machines may have begun life as nondeterministic automata. We give a hint of this next.

3 The Use of Pseudo-code

Once the abstraction of nondeterminism, as a set of choices corresponding to a successful search, is understood then we can settle back and explain what choices are made, in the familiar context of sequential algorithms. Consider this recursive definition of a set/language L of strings understood to be expressions over two operators and a set of variables V:

- If $x \in V$ then $x \in L$,
- If $x \in L$ and $y \in L$ the $(x \oplus y) \in L$ and $(x \otimes y) \in L$,
- The first two rules define every element of L

Therefore $((a \otimes b) \oplus (b \otimes c)) \in L$, where $a, b, c \in V$. Of course, the last line of the definition is crucial for any "iff" interpretation.

Most students have no trouble understanding this type of recursive definition, even though it is laced with nondeterministic choices, You can choose any variables in V. You can choose any x and y from L. And you can choose either of the two operators. A string w exists in L if and only if *there exists* choices corresponding to that w. Students understand those choices in either a bottom-up or top-down manner, as we discuss.

This pseudo-code describes how to generate a given $w \in L$ bottom-up. The process verify-1(w) continues until it discovers $w \in L$. In particular it will "succeed" to discover it after a finite number of iterations, but if $w \notin L$ the process is doomed to an infinite loop as written (string length constraints can be added to guarantee termination). The language L' is just an initial construction of L. This highlights the asymmetry between success and failure in nondeterministic algorithms for decision problems.

```
verify-1(w)
L' ⇐ V;
while  w ∉ L' do
      choose x ∈ L'
      choose y ∈ L'
      choose c ∈ {⊕, ⊗}
      L' ⇐ L' ∪ {(xcy)}
succeed
```

For contrast consider a second way to use the recursive definition, a "top-down" approach. Given w, it chooses how to decompose w and verifies its choices. Note that this "fails" if it *can not* choose an appropriate x and y (so it does not fail implicitly by going into an infinite loop). This top-down approach is confusing and unwieldy since there is no guidance on how to make the choices, especially since they are verified in the black-box of a recursive call. Top-down methods are only efficient when there is guidance (such as counting parentheses would be in this example). Let Σ be the alphabet for L.

```
verify-2(w)
if w ∈ V then
      succeed
else
      choose x ∈ Σ*
      choose y ∈ Σ*
      if w = (x ⊕ y) ∨ w = (x ⊗ y) then
            verify-2(x)
            verify-2(y)
            succeed
fail
```

These examples have explicitly introduced the novel idea of an algorithm making choices. We have used this new type of statement:

$$\textbf{choose } x \in S.$$

These choices are nondeterministic, as defined above. In particular the correctness of the pseudo-code relies only on the assertion: choices will be made leading to **succeed** if and only if such choices can be made. That is, w is verified if and only if a search can find the choices to prove it is verified. Finally such recursive definitions naturally lead to generative models which we will discuss.

4 FAQs

We have polled students about their understanding, over several sections, in several semesters with several instructors. This has revealed specific misunderstandings, addressed here.

Is a nondeterministic algorithm/process unpredictable or random? No. This belief is betrayed by a remark like "A nondeterministic process can give different answers for the same input." Such a process always gives the same answer for the same input; it will always succeed or it will always fail. What is hidden is the mechanism of the *search* to arrive at that answer. Since the mechanism is hidden, its path to the answer is unknown and so it choices are unknown. It might even search possibilities in a random order. Just because the mechanism's choices can not be predicted does not alter the fact that the process' outcomes are deterministic.

Is a nondeterministic algorithm/process ambiguous? Yes. In the context of Theory "ambiguity" means that answer is known but "why" is not fixed. For example, a string may be known to be generated by a grammar, but we may be able to generate it two different ways. So we can not state what choices the grammar made in order to construct a derivation, only that such choices exist. In the same way the nondeterministic process is ambiguous. Of course for a particular problem an algorithm that uses **choose** may not have real choices, every choice is forced, so the appearance of ambiguity is not actual.

5 Class Plan Details

After the definitions of languages and their operators, we start with regular languages spending at least two weeks on the two generative models, regular grammars and regular expressions. The general notion of grammars must be introduced. We end with the result that every regular language has a deterministic regular grammar. All generative models are shown to be equivalent.

Consider *regular grammars.* All grammatical models are based on choices (what to replace and what to replace it by). Regular grammars have two unusual properties that simplify the choices: first, there is no choice of what to replace next in a derivation, and second, it is possible to modify the grammar, at the expense of an exponential blow-up in its description, to only have one choice of what replacement to make while generating a given string. (This is analogous to the Rabin and Scott proof, only simpler.)

Next consider the popular generative technique of *regular expressions.* Explicitly regular expressions are based on a recursive definition, like above, with three language operators. So it is unsurprising that verifying a string is in the language generated by a regular expression is readily understood as a nondeterministic process.

Finite automata are just thinly-veiled mechanisms for simulating derivations using regular grammars. (The proof of correctness of the simulation is only a few lines long.) So we would be justified in concluding that deterministic finite automata exist only because "deterministic regular grammars" exist (and that is only because we are willing to tolerate exponential blow-ups in the descriptions). In other words nondeterminism is the norm. Determinism is focused on because it is realistic and in many real-life cases it only results in a sub-exponential blow-up. This point is made clearer with context-free languages, below.

When we get to finite automata we need only show how they are understood as simulating the derivation of a regular grammar. It is the same simulation for deterministic and nondeterministic automata. When we progress to the closure properties of regular languages everything is simplified to three line proofs. Since we have several models at this point we do not need proof techniques like the cross-product of automata.

We give a specific benefit: there is no need for ϵ (empty string) transitions in finite automata. The tacit reason for this unnecessary construct is the standard proof of equivalence to regular expressions. But now we prove (without mentioning automata at all) the equivalence of regular expressions and regular grammars. The ϵ now appears only as part of a production in a grammar, a very natural setting that introduces no new concepts.

For context-free languages we again start with a generative model, context-free grammars. And again we start with nondeterminism. Famously, we do not (can not) show there is an equivalent deterministic context-free grammar. It is for this reason that when we move on to the recognition model, push-down automata, that we are stuck with only a nondeterministic model. (As an aside, the *epsilon* transitions in push-down automata come from the notion of "acceptance by final state" and can be obviated by using "acceptance by empty stack," which also simplifies many of the proofs.)

The subject of deterministic push-down automata is extremely important because of the intimate relationship to the important problems of parsing and compiling. However this theoretical thread dangles away from the elegant fabric of the theory. It is a major open problem to characterize deterministic context-free languages using generative models.

The LBA (linear-bounded automata) problem, neglected by most textbooks, is perhaps the central open question about nondeterminism. It asks if nondeterministic and deterministic context-sensitive languages are the same. It gets to the core issues of nondeterminism and *computability* and *space complexity*. When we move on to Turing machines and related language models we again find that determinism is easily purchased with an exponential blow-up in cost. This leads to NP-completeness and gets to the core issues of nondeterminism and *time complexity*.

NP-complete problems give an opportunity to emphasize the relationship of nondeterminism and search. The minimum-spanning tree problem and the traveling salesman problem both have similar nondeterministic algorithms. The difference, of course, is that an efficient deterministic algorithm exists for MST that complete ignores the underlying search of the nondeterministic algorithms and TSP does not appear to exist. For every NP-complete problem all known polynomial-time algorithms are characterized by nondeterminism. Nondeterminism gives a search model for a problem, but the problem may admit more efficient deterministic searches.

6 Conclusions

In summary, we propose a new way to teach nondeterminism. First you teach the notion that it is an abstraction of search, exhaustive search with hidden details. Second you teach generative models before recognition models in order to make students understand that nondeterminism provides a natural way to describe languages (even while suppressing mechanistic details). Third by using the choose construct in pseudo-code you can make the use of choice explicit. This allows the analogous use in automata to be introduced naturally. These approaches have been used for several semesters in an undergraduate Theory course. We have used a new textbook [6].

References

[1] M. Armoni, N. Lewenstein and M. Ben-Ari, Teaching students to think nondeterministically, SIGCSE'08 (2008) 4-8.
[2] M. Armoni and M. Ben-Ari, The concept of nondeterminism, *Science and Education* 18 (2009) 1005-1030.
[3] R. Floyd, 1967, Nondeterministic algorithms, *J ACM* 14 (1967) 636-644.

[4] M. O. Rabin and D. Scott, Finite automata and their decision problems, *IBM J Research and Development* 3 (1959) 114-125.
[5] T. H. Cormen, et al., *Introduction to Algorithms,* 3rd edition, MIT Press, 2009.
[6] D. Richards and H. Hamburger, *Logic and Language Models for Computer Science,* 3rd edition, World Scientific, 2017.