Derivatives of Automata

Justin Hu

April 21, 2022

Abstract

A method of computing the derivative of any automaton is presented, based on Brzozowski's work on derivatives of regular expressions and Might, Darais, and Spiewak's work on derivatives of context-free grammars. Equivalence with Brzozowski's and Might et al.'s derivatives are proven, but practical applications depend on other means of constructing automata.

1 Introduction

When computers read data, the data is often read as a flat sequence of bytes. Frequently, those bytes need to be converted into a structured form, like a tree, in a process called parsing. A similar activity would be diagramming sentences, where a sequence of words is converted into a grammatical tree. Indeed, many terms used in describing the parsing of computer data are taken from linguistics, mostly because of the linguist Noam Chomsky.

Often, a string gets parsed to determine whether or not it's an acceptable input. A set of acceptable strings is called a formal language, and it can be described by a formal grammar. Formal languages are mathematical objects; while the set of possible English sentences is up for debate, whether or not a string is in a formal language can be rigorously proven. One way of proving a string is in a formal language is to show that it can be parsed using the formal grammar describing the language. As such, given some input (in the form of a string), to determine if it's an acceptable input or not, first construct a formal language that contains all acceptable inputs, construct a formal grammar from that formal language, and parse the input using the formal grammar. If the parse succeeds, then the input is acceptable. If the parse fails, then the input is unacceptable.

One question remains – how to parse a string using a formal grammar? For a formal language described by a regular expression, there exist algorithms to determine if a string matches (is parsed by) any regular expression. For a formal language described by certain kinds of context-free grammar, there also exist algorithms to determine if a string matches. A more general tool, however, is recognition using derivatives [4]. Taking the derivative of a language with respect to some character constructs a new language. The new language is constructed from the original language by taking those strings in the original language starting with the character we are taking the derivative with respect to, then removing the first character of those strings. If you repeatedly take the derivative with respect to subsequent characters in the string, and end up with a language that can accept the empty string, then the string is in the original language.

Existing work by Brzozowski [1] and Might et al. [4] give algorithms for recognition using derivatives of regular expressions and context-free grammars, respectively. Regular languages and context-free languages, however, have limitations. An example would be the language consisting of any strings containing the character `a` some number of times followed by an equal number of the character `b` and then followed by an equal number of the character `c`. This and similar languages cannot be defined using regular expressions or context-free grammars, and so, cannot be recognized using existing methods of computing derivatives. This is a limitation this thesis seeks to address.

1.1 Formal Languages and Grammars

Chomsky [2] defines a formal language as a set of strings. A formal language can be described by a formal grammar. A formal grammar is defined as a tuple of:

- N, the set of nonterminal symbols
- Σ, the set of terminal symbols; $\Sigma \cap N = \emptyset$
- P, the set of rules, where a rule is of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
- s, the start symbol; $s \in N$

For some grammar G, the relation $\Rightarrow_G$ (G derives in one step) is defined as $x \Rightarrow_G y = \exists u, v, p, q, \in (\Sigma \cup N)^*. \; x = upv \wedge p \rightarrow q \in P \wedge y = uqv$.

In other words, a string of symbols x, composed of substrings u followed by p followed by v derives a second string of symbols y, composed of substrings u followed by q followed by v if there is a rule from p to q in the grammar. The reflexive transitive closure of the $\Rightarrow_G$ relation starting from s defines the set of strings described by the formal grammar.

1.2 Derivatives of Formal Languages

The derivative, $D_c(L)$, of a formal language L with respect to some character c is described by Might et al. [4] as follows:

$$D_c(L) = \{w \mid cw \in L\}$$

Informally, to construct the derivative of a language, take all of the strings in the language, keep only those whose first character is the character the derivative is taken with respect to, and remove the first character from each string. Note that the derivative of a formal language is another formal language.

Derivatives are useful because of their properties when chained [4]:

$$\forall u \in \Sigma^*.\ u \in L \Leftrightarrow (u = \epsilon \wedge \epsilon \in G) \vee (\exists c \in \Sigma, r \in \Sigma^*.\ cr = u \wedge r \in D_c(G))$$

Informally, for any string s with characters $a, b, c, \ldots, x, y, z$ and grammar G, if $D_z(D_y(D_x(\ldots D_c(D_b(D_a(G))))))$ contains the empty string, then s is in G. Repeated computation of the derivative is a method of checking a string's membership in a grammar incrementally, instead of computing whether or not the grammar can derive the string in question from the starting symbol.

2 Previous Work

Brzozowski [1] describes derivatives of regular expressions, describing a direct method for testing membership of a string in any regular language. Brzozowski specifies the process of computing a derivative by giving recursive derivation rules operating on the syntax of a regular grammar:

- $D_c(c) = \epsilon$ (the derivative with respect to some character c of a regular expression expecting just c is the regular expression expecting the empty string)

- $D_c(\epsilon) = \emptyset$ (the derivative of a regular expression expecting the empty string is the regular expression that always rejects)
- $D_c(\emptyset) = \emptyset$ (the derivative of a regular expression that always rejects is one that always rejects)
- $\forall x \in \Sigma.\ x \neq c \Rightarrow D_c(x) = \emptyset$ (the derivative of any single that isn't c is also the regular expression that always rejects)
- $D_c(P*) = D_c(P)P*$ (the derivative of a repetition is the derivative of the repeated expression followed by the original repetition)
- $D_c(PQ) = D_c(P)Q \mid \delta(P)D_c(Q)$ (the derivative of concatenation is the derivative of the first part followed by the second part alternated with the derivative of the second part, but only if the first part accepts the empty string)
- $D_c(P \mid Q) = D_c(P) \mid D_c(Q)$ (the derivative distributes over alternation)

The function $\delta(P)$ is defined as

$$\delta(P) = \begin{cases} \epsilon & \epsilon \in P \\ \emptyset & \epsilon \notin P \end{cases}$$

The function $\delta(P)$ is used to special-case situations where some subexpression in an operator might already accept the empty string.

Might et al. [4] later describe derivatives of context-free languages, again describing a direct method for testing membership of a string in any context-free language. Might et al. also specify the process of efficiently computing a derivative by adding memoization and laziness rules to allow Brzozowski's rules to operate on context-free grammars. Might et al. note that the derivative of a context-free language is almost identical to that of a regular language because a context-free grammar can be viewed as a recursive regular grammar.

3 Derivatives of Automata

It is well-known that formal languages have corresponding nondeterministic automata [3]. For example, the regular languages correspond to nondeterministic finite automata, and the context-free languages correspond to

nondeterministic push-down automata. Furthermore, the context-sensitive languages correspond to nondeterministic linear bounded automata, and unrestricted languages correspond to Turing machines. The common thread through all of these automata is that they have a finite number of states, a set of rules for transitioning between states based on input symbols, and a set of rules for manipulating the machine's context while transitioning between states.

A nondeterministic automaton is described as a tuple of:

- Q, the finite, non-empty set of states
- F, the finite set of accepting states; $F \subseteq Q$
- Σ, the finite set of symbols allowed in the input
- C, the possibly infinite, non-empty set of possible contexts
- δ, the transition relation; $\delta : Q \times \Sigma \times C \to \mathcal{P}(Q \times C)$
- S, the set of tuples of starting states; $S \subseteq Q \times C$

The set of starting states is rendered unnecessary if an ϵ-transition is allowed from the starting state. Representing that transition as a set of starting states and contexts allows for a more direct implementation of this algorithm.

The definition of contexts, and what changes to the context are allowed within the transition relation determine the power of the language in question. For example, if the context has only one value, thus carrying no meaningful information, the automaton described is equivalent in power to a nondeterministic finite automaton.

Informally, automaton accepts a string if, starting from any of the starting states and transitioning through the transition relation once per character of the string, in order, there is any state from the set of accepting states in the states reached by the transition relation. To describe this more formally requires a multi-transition function.

For some fixed automaton, let $\delta^* : Q \times \Sigma^* \times C \to \mathcal{P}(Q \times C)$ be the multi-transition function.

$$\delta^*(q, s, c) = \begin{cases} \{(q, c)\} & s = \epsilon \\ \bigcup\{\delta^*(q', s[1:], c') \mid (q', c') \in \delta(q, s[0], c)\} & \text{otherwise} \end{cases}$$

The multi-transition function describes all of the states reachable by transitioning along the characters in the given string, one at a time, from some fixed starting state and context. If the string is empty, stop - the current state and context is the only reachable one. If the string is not empty, then for every state and context reachable by transitioning down the first character of the string, union together the states reachable by transitioning along the characters in the rest of the string.

Finally, for some fixed automaton, let A be the acceptance predicate on Σ^*.

$$A(s) = \exists f \in F, c \in C.\ (f, c) \in \bigcup\{\delta^*(q, s, c) \mid (q, c) \in S\}$$

In other words, the automaton accepts a string if and only if the set of states reachable from all of the starting states and contexts and multi-transitioning from the string contains at least one final state.

3.1 Definition of the Derivative

Previously, methods for computing derivatives based on formal grammars were introduced. A method for computing derivatives based on automata also exist. The derivative of the automaton $(Q, F, \Sigma, C, \delta, S)$ with respect to some symbol σ is another automaton, $(Q, F, \Sigma, C, \delta, S')$ such that $S' = \bigcup\{\delta(q, \sigma, c) \mid (q, c) \in S\}$. That is, the derivative of the automaton is the same as the original automaton, but with different starting states. The starting states of the new automaton are those states and contexts reached by transitioning once using the transition function from the starting states of the original automaton. As an aside, if the resulting automaton has an empty set for its starting states and contexts, then the automaton automatically rejects any and all inputs immediately.

3.2 Equivalence with Generalized Derivatives

A method for computing derivatives of automata has been given, but does it really do what it claims? In order to compare automata, formal languages, and grammars, define an automaton, language, or grammar to be equal to another automaton, language, or grammar if the same set of strings is accepted by both.

Theorem 1. *Let D_c : Language $\rightarrow$ Language be the derivative of a language as described in section 1.2.*

Let A_c : Automaton $\rightarrow$ Automaton be the derivative of an automaton as described in section 3.1.

Let L be any context-free language, and let Automaton(L) be its corresponding automaton.

Then:

$$D_c(L) = A_c(Automaton(L))$$

Proof. 1. By the definition of $D_c(L)$, it suffices to prove for any arbitrary $w \in \Sigma^*$, $cw \in L$ if and only if $w \in A_c(\text{Automaton}(L))$

2. To prove the bidirectional implication, it suffices to prove the implication in both directions - that is, assuming some $w \in \Sigma^*$, $cw \in L$ implies $w \in A_c(\text{Automaton}(L))$ and $w \in A_c(\text{Automaton}(L))$ implies $cw \in L$
   - 2.1. To prove the implication in the forwards direction, it suffices to assume the antecedent $cw \in L$ and show $w \in A_c(\text{Automaton}(L))$
     - 2.1.1. Decompose the automaton Automaton(L) into its components $(Q, F, \Sigma, C, \delta, S)$, and decompose the derived automaton $A_c(\text{Automaton}(L))$ into its components $(Q, F, \Sigma, C, \delta, S')$
     - 2.1.2. We know $cw \in \text{Automaton}(L)$ since $cw \in L$ and Automaton(L) and L accept the same strings by definition
     - 2.1.3. So by the definition of an automaton accepting a string, there exists some state and context (q, d) in the starting states such that $(q', d') \in \delta(q, c, d)$ and $\exists f \in F, c_f \in C.\ (f, c_f) \in \delta^*(q', w, d')$
     - 2.1.4. And if $(q', d') \in S'$ and $\exists f \in F, c_f \in C.\ (f, c_f) \in \delta^*(q', w, d')$ then $w \in \text{Automaton}(L)$ by the definition of automaton acceptance
     - 2.1.5. And we know $(q', d') \in S'$ by the definition of $A_c(\text{Automaton}(L))$ and 2.1.2
     - 2.1.6. Thus $w \in A_c(\text{Automaton}(L))$
   - 2.2. To prove the implication in the backwards direction, it suffices again to assume the antecedent $w \in A_c(\text{Automaton}(L))$ and show $cw \in L$
     - 2.2.1. Again decompose the automaton Automaton(L) into its components $(Q, F, \Sigma, C, \delta, S)$, and decompose the derived automaton $A_c(\text{Automaton}(L))$ into its components $(Q, F, \Sigma, C, \delta, S')$
     - 2.2.2. We know there exists some $(q, c, d) \in Q \times \Sigma \times C$ and some $(q', d') \in S'$ such that $(q', d') \in \delta(q, c, d)$ and $(q, d) \in S$ by the definition of the derivative.
     - 2.2.3. And if $(q', d') \in S'$, $w \in A_c(\text{Automaton}(L))$, $(q', d') \in \delta(q, c, d)$, and $(q, d) \in S$ then $cw \in \text{Automaton}(L)$ by definition of automaton acceptance
     - 2.2.4. And we do know all of that, so $cw \in L$ since $cw \in \text{Automaton}(L) \wedge$

$L = \text{Automaton}(L)$

2.2.5. Thus $cw \in L$

□

As a corollary to Theorem 1, since Brzozowski's derivative and Might et al.'s derivative both compute the derivative as defined above [1, 4], the automaton view of the derivative computes the exact same thing as Brzozowski's derivative and Might et al's derivative. That is, the set of strings accepted by the derivative computed on the automaton of a regular expression is the same set of strings accepted by the derivative computed using Brzozowski's rules, and likewise, the set of strings accepted by the derivative computed on the automaton of a context-free grammar is the same set of strings accepted by the derivative computed using Might et al.'s rules.

4 Conclusion

Automata can be used to compute the derivative of an arbitrary language or grammar - convert it into its corresponding automaton, and then use the above constructive definition of the derivative on an automaton to compute the automaton describing the derivative of the language, and then convert the automaton back into a language or grammar. Unlike Might et al., this process does not necessarily lead to faster computation of string membership in a language; consider the automaton corresponding to a context-sensitive language. One known conversion is to have the automaton nondeterministically derive strings from the starting string, and accept only if the derived string is equal to the input. This automaton, however, does very little computation as it consumes its input; it does all of its computation once it has seen all of its input, and needs to nondeterminstically derive strings from the starting string. Computing the derivative of such an automaton results in another automaton that is almost identical to the original, but with an updated starting context reflecting the input of a single character. Repeatedly applying the derivative copies the input string into the context, and does not even have the possibility of rejecting the input early. The derivative of an automaton is useful if the automaton does computation as it consumes input, and can thus reject a string purely based on a prefix.

An issue is the use of the automaton corresponding to a language – if there exists some automaton corresponding to a language, the automaton could be utilized directly. This is true. However, viewing automata as differentiable may be useful when developing derivative rules for generalized

grammars; the structural similarity between the language and the automaton may result in parallels between the derivative rules based on the grammar and the derivative rules based on the automaton.

In practical terms, almost no commonly used formal languages are more powerful than context-free languages, and there already exists derivative rules allowing for computation of the derivatives of regular and context-free languages. Natural languages may require more powerful formal languages to be parsed; this may be a potential application of automata-based derivatives.

References

[1] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

[2] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[3] J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages, and computation*. Reading, Mass: Addison-Wesley, 1979.

[4] M. Might, D. Darais, and D. Spiewak, "Parsing with derivatives: a functional pearl," *Acm sigplan notices*, vol. 46, no. 9, pp. 189–195, 2011.