— LECTURE NOTES —

Automata

2 Tree Automata

1 Tree Domains Assume given a finite prefix of the natural numbers, $[0, n]$. A *tree domain* t is a subset of $[0, n]^*$ (strings over the numbers in the closed interval $[0, n]$) which satisfies the following two conditions. Here $v, w \in [0, n]^*$ and $i, j \in [0, n]$.

Prefix Closure $vw \in t \Rightarrow v \in t$. (Note that therefore ϵ is in every tree domain. Indeed it is the root of every tree.)

Left Sister Closure If $vj \in t$, then, for all $0 \leq i \leq j$, $vi \in t$.

This means that a tree can be thought of as *a set of strings*, i.e., a language over the vocabulary $[0, n]$. There are some important results that hinge on this. For example, the set of derivation trees for a context free grammar, taken together and considered as a set of paths, form a regular language. And more generally this reinforces the intuition that trees are pretty much like strings stood on end.

2 Trees are Labeled Tree Domains Given a ranked vocabulary Σ and a countable set of variables X, any tree in $T_\Sigma(X)$ can be characterized as a function τ from a tree domain t to $\Sigma \cup X$. And vice-versa.

3 The Empty Tree The empty tree is just the tree labeling function with an empty domain. This is not the same thing as the tree domain containing only the empty string! Every 1-node tree has domain $\{\epsilon\}$. We will denote the empty tree as Λ.

4 A Little Algebra Given a ranked vocabulary Σ, a Σ-algebra $\mathcal{A}$ is formally presented by giving a *carrier set* A and, for each n-ary operation symbol $\sigma \in \Sigma$, an actual n-ary operation over the elements of A, i.e., an operation $\sigma^{\mathcal{A}} : A^n \to A$.

It has proven very useful to view automata from an algebraic perspective. Given a deterministic finite state automaton $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, it is tempting to think of Q as the carrier of a Σ-algebra, where the interpretation of symbols in Σ (all of which are of rank 1) is given by δ. The idea is to break up δ into a lot of little functions, one for each member of Σ. We can do this since Σ is finite. So instead of having in our transition table an entry like $\delta(q_i, \sigma) = q_j$, we now have as part of the definition of the function δ_σ that $\delta_\sigma(q_i) = q_j$. This is now formally an operation, that is, a function over a homogeneous domain.

Each of these mini-transition-functions is *monadic*: it takes a single argument. *Tree automata* arise as a natural extension of this point of view once we allow elements of Σ to take more than one argument. For example, if $\sigma \in \Sigma_2$, then we might have that $\delta_\sigma(q_i, q_j) = q_k$. In automata theoretic terms, we have an automaton that is formally in two distinct states (as if it has been reading two distinct strings) and it transfers to a single state (as if the two distinct strings it has been reading have converged into a single string – But of course that is exactly what happens in a tree domain: it is a set of distinct strings which share common prefixes, which converge after a certain point, in other words.)

5 What do we do about the intial state and the final states? First, we have a whole mess of 1-place function symbols in Σ, but so far no 0-place function symbols (constants). We take ϵ to be our sole and only constant. Then we stipulate that our "FSA Algebra" should always interpret the symbol ϵ as q_0.

> Note that this means that, if we want our automata to run from left to right, then we must write our function symbols *postfix*. The string abc is then thought of as the term $(((\epsilon)a)b)c$.

Note that this means that the empty string is treated somewhat differently here! Previously we could always think of a string uv as being equal to the string $u\epsilon v$, but we cannot do that any more. The function symbol ϵ is formally of rank zero, it takes no arguments. So, in particular, the term $(u)\epsilon$ is syntactically ill-formed.

To avoid confusion, when we are thinking of strings as nonbranching trees, since we have already reserved Λ to be the empty tree, we will use Λ as the sole element of Σ_0 instead of ϵ.

6 Now every string in Σ^* (or, more accurately, $\{\Lambda\} \cup \Sigma^+$) gets an interpretation in our algebra. That is, every string is assigned a state in the usual algebraic way. Given a string $w\sigma$, its interpretation in an algebra $\mathcal{A}$ (written $(w\sigma)^\mathcal{A}$) is just $\delta_\sigma(w^\mathcal{A})$. And of course for the empty string (now written Λ) we have stipulated that $\Lambda^\mathcal{A} = q_0$.

So, for example, given the string abc, we compute $abc^\mathcal{A}$ as follows.

$$\begin{aligned}(\Lambda abc)^\mathcal{A} &= \delta_c((\Lambda ab)^\mathcal{A}) \\ &= \delta_c(\delta_b((\Lambda a)^\mathcal{A})) \\ &= \delta_c(\delta_b(\delta_a(\Lambda^\mathcal{A}))) \\ &= \delta_c(\delta_b(\delta_a(q_0)))\end{aligned}$$

Remember that what we are now writing $\delta_a(q_0)$ is just the same as what we used to write $\delta(q_0, a)$. So it should be clear that, given an actual set of states and transitions on the letters a, b and c, the calculation of the value of the term

$$\delta_c(\delta_b(\delta_a(q_0)))$$

is formally the same computation as running the underlying FSA over the string abc.

7 The role of the final states is to turn this system into a language description. Given a string w and a "state algebra" $\mathcal{A} = \langle Q, q_0, (\delta_\sigma)_{\sigma\in\Sigma}\rangle$ and $F \subseteq Q$, we say that w is in $L(\mathcal{A})$ if and only if $w^{\mathcal{A}} \in F$.

8 The mechanisms of algebraic interpretation do not change if we generalize from ranked vocabularies with symbols only of arity 0 and 1 to ranked vocabularies with symbols of greater arity. To keep things simple, however, we will consider only ranked vocabularies with at most binary symbols. That is the crucial step, from unary to binary. The rest is an easy exercise.

So, suppose I have one of these "state algebras" $\mathcal{A} = \langle Q, q_0, (\delta_\sigma)_{\sigma\in\Sigma}\rangle$ only now Σ_2 is no longer empty. So now our terms, in T_Σ, are not just linear, "stringy" terms, but include some "bushy" terms as well. What changes? Basically, nothing. Just as every string ended with Λ, so now every branch of our binary trees will end with Λ: it is, after all, our only constant.

As before, our algebra will interpret every term in T_Σ as a state in Q. That is what algebras do: they interpret expressions in a given language as values in a given carrier set. We can define a language (now a tree language, not a string language) as before by specifying a set of final states F, and accepting into our language only those trees which our state algebra evaluated to members of F.

9 Just as with strings, this process of evaluating a tree in terms of states can be thought of in automata theoretic terms. Our automaton is allowed to read many strings at once – in effect it is a process with a number of different threads – and when those strings merge into a common prefix the relevant automaton threads also merge.

The terminals of the tree (it is probably better to think of them as *post-terminals* are all Λ and are assigned the initial state q_0. Then, given a subtree $\sigma(t_1, t_2)$ such that $t_1^{\mathcal{A}} = q_1$ and $t_2^{\mathcal{A}} = q_2$, that is, after processing sub-subtree t_1 we are in state q_1 and after processing sub-subtree t_2 we are in state q_2, then we read symbol σ and transit to a new state q such that $\delta_\sigma(q_1, q_2) = q$. We continue in this way until we have read the symbol at the root of the tree. We check to see what state we are in and, if it is a final state, we have accepted the tree.

10 One thing that is lost in this generalization is the connection to Graph Theory. It is easy to draw FSAs as graphs. It is much harder to draw Tree Automata (FSTAs) as graphs because they contain *hyperedges*, i.e., edges that connect more than two nodes.

11 The kind of automaton just sketched out runs from the leaves of a tree to its root, and is deterministic. It is possible to specify nondeterministic tree automata, and tree automata that run top-down, from the root to the leaves.

However, while bottom up deterministic, bottom up nondeterministic and top down nondeterministic FSTAs all accept the same sets of trees, *top down deterministic* FSTAs accept only a much smaller class of trees.

So we lose one more result from string automata: for string automata there is an absolute equivalence between automata that run from left to right and automata that run from right to left.

12 It is easiest to think about tree automata, and it is certainly easiest to prove things about them, if you only consider trees with a fixed branching factor. So if you have a tree that you really think is unary branching, like

$$\begin{array}{c} \textit{Det} \\ | \\ \textit{the} \end{array}$$

then it is often convenient to think of this as having a "hidden sister":

$$\begin{array}{ccc} & \textit{Det} & \\ / & & \backslash \\ \textit{the} & & \Lambda \end{array}$$

Then the transition function is always binary. However, the mechanisms of algebraic interpretation are completely agnostic about the arity of the symbols involved. They work fine for trees with mixed branching factors.

From an automata theoretic perspective, this means that we have a *family* of transition functions: one for unary branching nodes, one for binary branching nodes, one for ternary branching nodes, etc. For any particular tree, it should be clear at any point which transition function to use.

13 An Example Consider the following very simple context free grammar.

$$S \longrightarrow NP\ VP \tag{1}$$
$$S \longrightarrow S\ \textit{and}\ S \tag{2}$$
$$NP \longrightarrow \textit{John} \tag{3}$$
$$VP \longrightarrow \textit{sneezed} \tag{4}$$

This generates the infinite language *John sneezed*, *John sneezed and John sneezed*, etc.

From this we can induce the following signature. Note that the only reason to insist on every tree branch ending in Λ is so that our automata have a single initial state. If we are prepared to work with automata with many initial states, then we can have the terminals of the grammar as real terminals. Otherwise, we have to stipulate what their arity is. If we are working with a fixed arity, then that is already decided. For example, we might treat the symbol *John* as an abbreviation for the term $\textit{John}(\Lambda, \Lambda)$.

In this example we will use multiple initial states, and treat terminals as terminals.

$$\begin{aligned} \Sigma_0 &= \{\Lambda, \textit{John}, \textit{sneezed}, \textit{and}\} \\ \Sigma_1 &= \{NP, VP\} \\ \Sigma_2 &= \{S/2\} \\ \Sigma_3 &= \{S/3\} \end{aligned}$$

Note that we disambiguate binary S from trinary S here. Then the derivation trees of the above grammar are just those trees recognized by the following automaton $\mathcal{G}$.

$$Q_0 = \{q_J, q_{sn}, q_{\&}\} \qquad F = \{q_S\} \qquad Q = Q_0 \cup F \cup \{q_{NP}, q_{VP}\}$$

$$\begin{aligned} John^{\mathcal{G}} &= q_J \\ sneezed^{\mathcal{G}} &= q_{sn} \\ and^{\mathcal{G}} &= q_{\&} \end{aligned} \qquad \begin{aligned} (NP(q_J))^{\mathcal{G}} &= q_{NP} \\ (VP(q_{sn}))^{\mathcal{G}} &= q_{VP} \\ (S/2(q_{NP}, q_{VP}))^{\mathcal{G}} &= q_S \\ (S/3(q_S, q_{\&}, q_S))^{\mathcal{G}} &= q_S \end{aligned}$$

Note the way in which our assignment of states undoes the disambiguation of $S/2$ from $S/3$: as one would hope, our automaton treats all kinds of S's the same.

14 Top-Down Recognizers We can also define automata that start at the root and run down the tree. Instead of multiple threads merging as the automaton works its way up the tree we think of a single process forking subprocesses as we work our way down the tree. If we are in state q_S reading S, for example, then we might spawn two subprocesses started in states q_{NP} and q_{VP}, respectively. We reverse the roles of our initial and final states, but everything else is the same. *Up to a point!* Consider the example above. If we are in state q_S reading an S, we have two choices of how to go. In this case there is local information that can tell us what to do, but in general this is not the case: we may have to make a decision high up in the tree that will not be proved right or wrong until much later.

As a result, there are some forests of trees that can only be recognized top down by a nondeterministic recognizer. Deterministic top down tree automata are inherently weaker than the other three varieties.

15 The Recognizable Forests If a set (forest) of trees can be characterized by a finite state tree automaton then we say that that forest is *recognizable*.

It turns out that the set of recognizable forests is *regular*. To see what that might mean, we obviously have to characterize what counts as concatenation and iteration for trees. (Alternation should be clear.)

16 Concatenating Trees The basic idea here is that, for trees, *concatenation is substitution*. But we have to be careful about what we mean by that if we want to make sure that the concatenation of one forest with another (by concatenating all the trees from one forest with those in the other) yields a recognizable forest.

There is actually a whole *family* of concatenation operations on sets of trees U and V in $T_\Sigma(X)$: one for each variable in X. These operations are written $U \circ_x V$ and are interpreted as follows. A tree t is in $U \circ_x V$ if and only if there is a tree t' in U such that t is the result of replacing every occurrence of x in

t' with some tree in V. The scope of our quantifiers is important here! It is possible to replace different occurrences of x with distinct trees drawn from V!

This is *not* the way substitution for a variable usually works. Before "substitution", $x = x$, of course. But after substituting t_1 for one occurrence of x and t_2 for another they are no longer equal! What they are is *equivalent* in the sense that t_1 and t_2 are both accepted by the automaton that recognizes V.

The construction is essentially the same as the construction for FSAs: in effect, you want to run the automaton for V to a final state, then jump to an initial state in automaton U and run to a final state in U.

17 Iteration for Trees The definition for iteration for trees is now forced on us. Recall that for strings we had

$$U^0 = \{\epsilon\} \qquad U^n = U^{n-1} \circ U \qquad U^* = \bigcup U^n$$

Here we have

$$U^0 = \{x\} \qquad U^n = U^{n-1} \circ_x U \qquad U^x = \bigcup U^n$$

18 The Recognizable Forests are Regular If we define the regular sets of trees to be all those that we get by closing the finite sets of trees under all the concatenation and iteration operators, then we get the expected result: the sets of trees recognizable by a tree automaton are exactly the regular forests.

19 The Recognizable Forests Yield the CFLs This is also a fact. The example automaton above should convince you that any CFG can be converted into an automaton that recognizes its derivation trees. Clearly the yield of that set of derivation trees is just the language generated by the grammar.

It turns out that the reverse is also the case: there is no recognizable forest that does not yield a context free language.

However: there are tree automata that recognize forests which are not the derivation trees for any context free grammar!

This is not a contradiction. It just means that the tree might look a little funny, but there is guaranteed to be a CFG with (slightly) different derivation trees that together all yield the same set of strings.

20 An Example There is an automaton over binary branching trees that recognizes all and only those trees where a single node is labeled b, and all the other nodes are labeled a.

$$Q = \{q_0, q_b\} \qquad F = \{q_b\}$$

$$\begin{aligned}
\Lambda^{\mathcal{B}} &= q_0 \\
(b(q_0, q_0))^{\mathcal{B}} &= q_b \\
(a(q_0, q_0))^{\mathcal{B}} &= q_0 \\
(a(q_b, q_0))^{\mathcal{B}} &= q_b \\
(a(q_0, q_b))^{\mathcal{B}} &= q_b
\end{aligned}$$

The states of this automaton can be interpreted as meaning "I have found a b!" (q_b) and "I have not yet found a b," (q_0). Finding a b moves us to q_b, after which we can find as many a's as we like, but we cannot read any more b's. But we must read one b so that we end in a final state.

There is no CFG whose derivation trees have this characteristic. Any CFG that contains a rule $A \longrightarrow B\ A$ or $A \longrightarrow A\ B$ can obviously reapply to its own output. If there is one B there can be any number of B's.

However, there is a CFG which yields the required set of strings (strings of the form $a^*(a|b)a^*$, i.e., strings with at most one b in them). And it is a very instructive grammar, as well. The nonterminals of this grammar will be *pairs* formed by combining a state and a symbol. Then the grammar is as follows.

$$(a, q_0) \longrightarrow (a, q_0)\ (a, q_0) \mid a \tag{5}$$

$$(a, q_b) \longrightarrow (a, q_b)\ (a, q_0) \mid (a, q_0)\ (a, q_b) \mid (b, q_b)\ (a, q_0) \mid (a, q_0)\ (b, q_b) \tag{6}$$

$$(b, q_b) \longrightarrow (a, q_0)\ (a, q_0) \mid b \tag{7}$$

To make this all work out right we have to allow for two start symbols, namely (a, q_b) and (b, q_b). Note that the derivation trees for this grammar are *extremely similar* to the trees recognized by automaton $\mathcal{B}$. They don't have a unique occurrence of symbol b but they do have a unique occurrence of symbol (b, q_b).

Keep this in mind: the symbols of a context free grammar may be thought of as encoding both "true" categorial information and also, simultaneously, *state*. This is the key to the treatment of long-distance dependencies in phrase structure grammars, as we will see.

21 Closure Properties The tree automata (leaving aside the nondeterministic top down recognizers) have all of the closure properties of FSAs. We have already noted that they are closed under the regular operations of concatenation, alternation and iteration. They are also closed under the boolean operations. The constructions are exactly the same in all cases, only we have to consider transition functions or relations that have more arguments than is the case with FSAs.

One of these results is particularly striking and you should keep it in mind.

1. The recognizable forests are closed under intersection.

2. The recognizable forests yield the context free languages.

3. *But* the context free languages *are not* closed under intersection.

As a witness to the last point, consider the context free languages $L_1 : a^n b^n c^m$ and $L_2 : a^m b^n c^n$. The intersection $L_1 \cap L_2$ is the language $a^n b^n c^n$, which is not context free. How can this be?

The key point is that the intersection of two recognizable forests is a set of *trees*. A given tree is in the intersection if and only if it is in both of the input forests. That means that a *string* is only in the yield of this intersection if it is the yield of a tree which is in this intersection, i.e., a tree which is in both languages. Now, the string *aaabbbccc* is in both L_1 and L_2 above, but it is parsed differently by the relevant grammars. There is no pair of CFGs that (a) yield L_1 and L_2 respectively and (b) assign the same trees to strings in $a^n b^n c^n$.

22 This is an important fact to keep in mind. One of the key advantages to FSAs that has made them popular in the field of Natural Language Processing is the range of constructions that is possible for them. It is very straightforward to build up huge and complex automata by starting from very simple pieces and combining them bit by bit into big hairy monsters. This is not something you can do with CFGs, mainly because of the impossibility of having an intersection construction.

But typically when grammarians want to specify a grammar as combining the properties of two simpler grammars, they do not want the two simpler grammars to assign radically different trees to the same strings. Indeed, they are often only casually interested in the strings and are really mainly using CFGs to describe the syntax trees for those strings. It seems pretty clear that they would be better off using Tree Automata!

There are two reasons why this hasn't happened. The first is that running a tree automaton presupposes that you have a tree to run it on. But natural language texts are given to us as sequences of strings, and the whole point is that we don't know what tree to assign. So question 1 is: How do we parse with tree automata? The other problem is that there is by now a rich mathematical and computational tradition in the study of context sensitive languages that allows us to extend many techniques from context free languages to context sensitive languages. This is important if, as most people assume, natural languages can be non-context free. So question 2 is: How do we extend tree automata to deal with mildly context sensitive languages?

Question 2 is hard, and little is known about the answers. Question 1 turns out to be easy. Indeed, you already know how to do it! For any given sentence it is possible to construct the tree automaton that recognizes all the trees in any given T_Σ which yield that sentence. Then, assuming you have an automaton $\mathcal{G}$ that recognizes all and only the well-formed syntactic structures, you just intersect $\mathcal{G}$ with the automaton for the input string. The result is an automaton which recognizes all and only those trees which (a) are well-formed with respect to the grammar and (b) yield the input string. The complexity of this procedure is $O(n^3)$, like the complexity of parsing with a CFG.