— LECTURE NOTES —

Closure, Reachability and Tabular Parsing

1 FSA Reachability

1 We return (briefly) to the problem of deciding what states in a given FSA are reachable from an initial state. (We allow ourselves any number of start states.)

For simplicity we assume that all ϵ-transitions have already been removed from the automaton. This will also help us compare what we are doing with FSAs to what we intend to do with tree automata and phrase structure grammars.

2 The computation of the reachable states of an FSA is a typical example of an *agenda driven* computation. That is, the computation involves keeping track of a set of tasks that remain to be completed. Each step of the computation involves selecting a task to attempt next, and executing it. Execution of a task may result in the discovery of new tasks that will need to be accomplished. These are added to the agenda. Computation proceeds until the agenda is emptied out. At that point, since there is nothing left to do, it follows that we must be done.

3 The agenda supplies us with a sort of control structure, which helps us to decide what to do next and to tell when we are done. It doesn't tell us how to accomplish a given task, and it doesn't tell us how to decide whether we have been successful or not, once the agenda is empty and we know that we are done.

4 Our agenda driven computations are all *constructions*, and this will remain true throughout our study of tabular and chart parsing. That means that each step in the computation involves adding a piece to the construction, and success or failure of the computation is dependent on some particular property of the construction that can be evaluated at the end of the day.

To take a concrete example, it can be determined whether or not the language accepted by a FSA contains any strings (the *emptiness problem*) by constructing a new automaton which is the restriction of the old one to its reachable states, and then seeing whether the newly constructed reachable state restricted automaton contains any final states. If it does not, then it does not accept any strings; otherwise it does.

5 For the specific problem of constructing a reachable states restricted FSA from a given FSA, which may contain unreachable states, the agenda is treated as a list of *states*. Each state stands for a task, namely the task of determining which states of the input automaton are reachable from the agenda state. *Reachability* means that we abstract away from the vocabulary items that allow us to go from one state to another. We are asking only "Does there exist some vocabulary item that would allow us to go from this state to some other state?"

Formally, we can define an *immediate reachability relation*, which is like the arc relation in an ordinary automaton, except we ignore the arc labels.

$$R := \{(q_1, q_2) \mid \exists \sigma\, \delta(q_1, \sigma, q_2)\}$$

6 As with any relation, we can consider this as a "nondeterministic function". Let us write qR to mean the set of q' such that $R(q, q')$ (also written $q\ R\ q'$). We can extend this quasi-functional interpretation of R so that it applies to a set. Suppose we have a set Q, then let

$$QR := \bigcup_{q \in Q} qR \ .$$

Now we can define the *closure* of a set of states under this (nondeterministic) operation in the usual way. Suppose that the set of states we want to close is called I. (In our particular example, of the FSA reachability construction, I is the set of initial states of the input automaton.)

$$Q^0 := I \tag{1}$$

$$Q^{i+1} := Q^i \cup Q^i R \tag{2}$$

$$Q := \bigcup_{i < \omega} Q^i \tag{3}$$

Let us write R^* for the operation that maps I to Q, and call it "the closure of I under R."

It should be clear that in the particular case where I is the set of initial states of a finite state automaton, and R is the immediate reachability relation as defined above, then IR^* will be the set of reachable states of the given automaton.

This provides us with the mathematics of what we want to do, and as usual in Prolog programming the implementation of the construction is a relatively straightforward exercise in translating from "LaTeX math mode syntax" to "Edinburgh syntax".

7 Search The agenda, as it evolves over time, acts as the trace of a search for the solution to the reachability problem. That it represents a "search" is not perhaps immediately obvious, since the construction in question is pretty much deterministic. Nonetheless, there are differences based on the way the agenda evolves. Also, automata are graphs, and, as we already know, the same problems that appear in searching a graph appear in constructing automata as well.

In particular, where the automaton has loops in it, some computations of the closure under R will be infinite. Of course we know that because the automaton is finite, there must also be computations of the closure under R which are finite. Our job is to find those, and avoid the infinite ones.

One clue is to return to the definition of closure in (1–3). Observe that the Q that we are constructing must be a subset of the states of the input automaton. It is a set, so it doesn't contain duplicates. So at some point, for some particular choice of i, Q^{i+1} will contain no new states. That is, for all q in Q^i, there is some $j \leq i$ such that $q \in Q^j$. Once we reach this i, no new states will ever be added (if they could be added, they must have been added already), so we are done.

The role of the *agenda* is played, in this mathematical construction, by the set theoretic difference $Q^{i+1} - Q^i$. That difference contains just the states which are new. That difference is empty if and only if no new states were added in this step, and we know that therefore no new states will ever be added in any later step, so we are done, in a finite amount of steps.

> Exercise: *Given an automaton with n states, this step, in which no new states are added, must be less than or equal to n. Why?*

We can get the same effect computationally if we follow the math and *make sure the agenda only ever contains new states*. This is, of course, another way of stating the well known technique for searching a graph: mark the states you have visited, and never visit a marked state.

8 Search Order Note that the agenda can describe either a depth first search or a breadth first search. If you add new states to the agenda at the front, and remove them from the front as well, then the agenda is behaving like a stack, and so you are doing depth first search. If you add new states at the back of the agenda, and remove them from the front, then the agenda is behaving like a queue, and so you are doing breadth first search. And of course any other search strategy (like best-first search) is possible, with appropriate operations on the agenda.

There is another sense in which you can go depth first or breadth first. Given the next agenda item (a state, in our case), you can find all of the states reachable from there, and add them to the agenda all at once, or else you can just select one reachable state and add it. In that case you must also (a) make a note of which reachable states have not yet been considered and (b) retain the original state on the agenda until you have run out of states reachable from it. This situation is much like the situation faced by the Prolog interpreter: every clause whose head matches the current goal is "reachable" from the current goal. It solves both (a) and (b) by keeping choice points. For our purposes, however, it is almost always easier to go "breadth first" in collecting reachable states: grab them all at once and add them to the agenda in a batch.

9 Why Reachability? Consider constructions like the cross-product construction underlying the intersection and (sometimes) union operations, or the subset construction underlying the determinization operation. Both of these constructions create hugely many possible states from the states of the input automata or automaton. Many—often most—of these states are in fact unreachable.

It turns out to be rather easy to "coroutine" such constructions with the reachable states construction in such a way that the main construction (cross product or subset constructions for example) is always restricted to the reachable states. Basically, all of these constructions tell us how to make the initial states of the new automaton out of the initial states of the old one(s). That gives us our initial agenda. The definition of the relevant construction must also tell us how to construct new states from old and new transitions from old. Using this information, we can take a state from the agenda and construct new states that are reachable from it by new transitions. We never construct states that are unreachable, and we construct all of the states that are. Basically, the relation of reachability, as an implementation of a closure operation, gives these more sophisticated problems (intersection, determinization) a powerful and flexible control structure. We will see this same idea again in one of the most fundamental context free parsing algorithms: the Cocke Younger Kasami (CYK) algorithm.

10 **The Cross Product Construction** Given a Σ-automaton $\mathfrak{A} = \langle Q_{\mathfrak{A}}, I_{\mathfrak{A}}, F_{\mathfrak{A}}, \delta_{\mathfrak{A}} \rangle$ and $\mathfrak{B} = \langle Q_{\mathfrak{B}}, I_{\mathfrak{B}}, F_{\mathfrak{B}}, \delta_{\mathfrak{B}} \rangle$, we form the cross product automaton $\mathfrak{A} \times \mathfrak{B} = \langle Q_{\mathfrak{A}\times\mathfrak{B}}, I_{\mathfrak{A}\times\mathfrak{B}}, F_{\mathfrak{A}\times\mathfrak{B}}, \delta_{\mathfrak{A}\times\mathfrak{B}} \rangle$ as follows.

$$Q_{\mathfrak{A}\times\mathfrak{B}} := Q_{\mathfrak{A}} \times Q_{\mathfrak{B}} \tag{4}$$

$$I_{\mathfrak{A}\times\mathfrak{B}} := I_{\mathfrak{A}} \times I_{\mathfrak{B}} \tag{5}$$

and, for the intersection construction,

$$F_{\mathfrak{A}\times\mathfrak{B}} := F_{\mathfrak{A}} \times F_{\mathfrak{B}} \tag{6}$$

We construct the transition relation as follows.

$$\begin{aligned} ((q_{\mathfrak{A}}, q_{\mathfrak{B}}), \sigma, (q'_{\mathfrak{A}}, q'_{\mathfrak{B}})) \in \delta_{\mathfrak{A}\times\mathfrak{B}} \iff \\ (q_{\mathfrak{A}}, \sigma, q'_{\mathfrak{A}}) \in \delta_{\mathfrak{A}} \wedge \\ (q_{\mathfrak{B}}, \sigma, q'_{\mathfrak{B}}) \in \delta_{\mathfrak{B}} \end{aligned} \tag{7}$$

Clearly, it is easy to construct the set of initial states to give us our initial agenda. Then every agenda item gives us a pair of states—one from $\mathfrak{A}$ and one from $\mathfrak{B}$—which we can use to look up transitions in the input automata. For each σ for which there are transitions in both $\mathfrak{A}$ and $\mathfrak{B}$, we have the material we need to construct a new (possibly new, anyway) reachable state of $\mathfrak{A} \times \mathfrak{B}$. Note that we do not construct any states of $\mathfrak{A} \times \mathfrak{B}$ unless (a) they are initial states or (b) we are forced to add them by the reachability construction. So our output automaton is relatively unlikely to actually contain the whole of the cross product $Q_{\mathfrak{A}} \times Q_{\mathfrak{B}}$.

11 Prolog code for the main loop of the cross product construction is illustrated in Fig. 1. This particular implementation blindly adds all reachable states found

```
xprod_loop( [], _, _, _, _ ).
xprod_loop( [Q0-Q1|Rest], Aut0, Aut1, Aut, Max1 ):-
        (   pair_already_visited( Q0-Q1, Aut, Max1 ) ->
            true
        ;
            (   setof( Q2-Q3,
                       simul_reach(Q0,Q1,Q2,Q3,
                                   Aut0,Aut1,Aut,Max1),
                       QQs
                     ) ->
                xprod_loop( QQs, Aut0, Aut1, Aut, Max1 )
            ;
                true
            )
        ),
        xprod_loop( Rest, Aut0, Aut1, Aut, Max1 ).
```

Figure 1: Cross product construction: main loop.

by the call to `setof` to the agenda, and only checks to see if a state has already been visited when it is popped off of the agenda for further processing.

Observe that this is a basically depth first strategy. Given our agenda state `Q0-Q1`, once we get the states reachable from it (in `QQs`) we immediately go to work on them, before returning to work on `Rest`.

2 Parsing with Tree Automata

12 Parsing with tree automata is essentially a matter of intersecting a grammar automaton (which recognizes all and only the well formed phrase structure trees) with an automaton that recognizes all and only the trees that yield the input string. The intersection is then by definition a tree automaton which recognizes all and only the well formed trees that yield the input string. This set is generally referred to as the *parse forest* for the input sentence since, if the sentence is structurally ambiguous, the set will have more than one member.

Parsing with tree automata is a matter of intersecting automata, which in turn is a problem best tackled by using the reachable states construction to drive an agenda based closure construction.

13 The main difference between the reachability construction for FSAs and the reachability construction for tree automata is, of course, that the transition relation for a tree automaton is “wider” than the 3-place transition relation for FSAs.

Let us restrict our attention to strictly binary branching trees, for a moment.

That means that the transition relation will deal only with the nullary and the binary cases, assigning initial states to the nullary nodes (i.e., the leaves).

The agenda here, as before, will contain single states. So now the question of whether any states are reachable from a given agenda state q is a slightly more complicated question. Not only does this mean finding some vocabulary symbol, it also means finding a second state to fill the other slot of the transition relation. Furthermore, we have two cases to consider: q is assigned to the left subtree or q is assigned to the right subtree. (Not to mention the case where it is assigned to both!)

14 A little more formally, the *reachability relation* R for tree automata will involve existentially quantifying over both a variable that ranges over vocabulary symbols (as in the FSA case) and a variable that ranges over states. In this way R is still a binary relation. This is important because we need a relation that holds between an agenda item and a (possibly) new state.

$$R := \{(q_1, q_2) \mid \exists\sigma\exists q' \, (\delta(q_1, q', \sigma, q_2) \vee \delta(q', q_1, \sigma, q_2))\} \tag{8}$$

If we have trinary branching nodes, then we need to find *two* other states, and our agenda item can appear in any of *three* positions in δ, and so on.

Given this definition of R, however, the definition of R^* is exactly the same as for FSAs, and so the basic structure of the algorithm remains the same as well.

15 There is one further slight wrinkle that needs to be taken care of. We noted in passing that q' above might very well be q_1 itself. That is, there may very well be edges in the δ relation of the form $\delta(q_1, q_1, \sigma, q_2)$. Observe that this edge will be used twice in our computation, and so we will consider q_2 (at least) twice, even if only to reject it as having already been processed.

So, we must be sure to consider the pair (q_1, q_1) once, but we want also to be sure that we do not consider it twice.

The following strategy does the trick. The main fact is this: at some point we have to add q_1 to the set of states we are constructing. There are a number of points at which we can do this. First we can do it the moment we discover that q_1 is reachable, that is, at about the time that we add it to the agenda. We can also do it at some point after we remove q_1 from the agenda. Under this second option there are a few sub-options, since we will be considering q_1 as a left input state and then again as a right input state.

If we assert that q_1 is a state of the new automaton between the time when we consider it as a left state and as a right state then we will achieve the correct behavior. When we are considering it as a left state we will be looking for possible q_2's from among the already asserted states, but q_1 itself is not yet one of those. So the pair (q_1, q_1) will never come up for consideration. Then we assert q_1 (or rather assert that q_1 is a state), and consider all the states that we have asserted (now including q_1) as possible left states while we make q_1 the right state. This time the pair (q_1, q_1) will come up. The lefthand q_1 is drawn from our database of already constructed states, while the righthand q_1 is our original agenda item.

16 The algorithm I have just sketched out, for closing an initial set of data elements under a binary operation, is called the *Semi-Naïve Algorithm*. (I believe the "Naïve Algorithm" takes the entire cross-product of all known states, applies the relevant operation to those pairs, adds the results to the set of known states and then does it all again, until none of the states it derives are new. This way the consequences of the same pairs of states get evaluated over and over again.)

In our current context, we can see the semi-naïve algorithm as a specialization of depth first search to the particular problem of computing the closure of a given set under a binary operator. In particular, computing reachability for unary operators—of course—is just exactly like the depth first search algorithm that showed up in Fig. 1

17 So now we are in a position to consider how to do parsing by intersecting tree automata. Once we see that, we will already have seen the Cocke Kasami Younger algorithm, though dressed up in different clothes.

18 Suppose we are given two tree automata $\mathfrak{A}$ and $\mathfrak{B}$, as above, and we wish to intersect them. The construction of the states, initial states and final states is exactly as for FSAs: they are just the cross products of the relevant sets from $\mathfrak{A}$ and $\mathfrak{B}$. It is only the construction of the new transition relation $\delta_{\mathfrak{A}\times\mathfrak{B}}$ that is the least bit complex.

$$\begin{aligned} ((a_1, b_1), (a_2, b_2), \sigma, (a, b)) \in \delta_{\mathfrak{A}\times\mathfrak{B}} \iff& \\ (a_1, a_2, \sigma, a) \in \delta_{\mathfrak{A}} \wedge& \\ (b_1, b_2, \sigma, b) \in \delta_{\mathfrak{B}}& \end{aligned} \tag{9}$$

In an agenda driven computation, our agenda item may be playing the role of either (a_1, b_1) or (a_2, b_2), with the other being drawn from the set of states that we have already constructed.

19 **The Yield Automata** For any given string w and signature Σ there is a tree automaton $yields_w$ that accepts all and only those trees in T_Σ that yield w.

The nullary operations of Σ (i.e., elements of Σ_0) will be labels of leaf nodes, in this case lexical items. So $yields_w$ will have to assign some initial state to every word in the sentence. Furthermore, it must keep track of the order in which words—and substrings of w, more generally—appear in the input. Accordingly, states of $yields_w$ will be triples consisting of a substring of w, its start position, and its end position. We will write states as ${}_iu_j$, which would be assigned to a tree which yielded u and which required that u be the substring of w between position i and j. So the final state will always be of the form ${}_0w_{|w|}$, and the initial states will be of the form ${}_{i-1}(w_i)_i$, where w_i is the ith word of w.

The logic of the transition table is just like the logic of the standard Prolog `yields` predicate that you have seen defined on numerous occasions. We have just seen how δ treats nullary operators. Unary symbols are trivial: a unary branching tree yields exactly the same string as its sole subtree. So given $\sigma \in \Sigma_1$, we have

$$\delta(q, \sigma, q)$$

for all states q. In the case of a binary symbol $\sigma \in \Sigma_2$, the yield is the concatenation of the yields of the two subtrees. So, if the left subtree yields u and the right subtree yields v, then the left subtree must be assigned a state of the form ${}_iu_j$ and the right subtree must be assigned a state of the form ${}_jv_k$. Note the role that j plays here! It is that shared j that assures us that u and v can actually be concatenated together to form a continuous substring of the yield of the σ-labeled subtree. Then we have:

$$\delta({}_iu_j, {}_jv_k, \sigma, {}_iuv_k)$$

From this example it should be clear how to treat symbols from Σ_n for $n > 2$.

20 Actually, this automaton is richer than it needs to be. For one thing, note that if we ignore the string component of our states, we still have exactly as many states as we had before. In fact, we have an isomorphic automaton. To see that this is true, note that all the control is coming from the indices; the string is just being carried along as a convenient way of seeing what is actually going on.

So the only role that the string plays is at the very beginning, where we assign to every word in w a pair of indices indicating where it occurs in w. Given w, and its length $|w|$, there are about $n^2/2$ states in $yields_w$. (I think I got that right ...) However, as you have already seen, the automaton has a very compact description, which translates almost directly into Prolog. Ignoring the assignment to elements of Σ_0, every single instance of a $yields_w$ automaton can be described by the Prolog code in Fig. 2.

```
( w :: $( I-J, J-K, S, I-K ) ) :-
        binary( _, S, _, _ ).

( w :: $( I-J, S, I-J ) ) :-
        unary( _, S, _ ).
```

Figure 2: Transition table for $yields_w$.

21 Because the yields automata have such a simple general description, it will be possible to unfold all calls to their transition tables in the intersection construction so that the parsing automaton really takes only a single input argument, namely the grammar automaton.

22 **The Grammar Automaton** Assume that we have a context free grammar G. Call the corresponding automaton $\mathfrak{G}$. $\mathfrak{G}$ must accept all and only the derivation trees of G. In practice this means that the states of $\mathfrak{G}$ will correspond exactly to the nonterminal symbols of the grammar. As a result, Σ itself need only contain a single symbol in each Σ_n, which we would think of as an n-ary concatenation symbol (where "unary concatenation" is just the identity function).

The main reason not to do this is so that we can think of $\mathfrak{G}$ as really recognizing derivation trees of G. This is a very weak reason, since there is a grammar G' which is almost exactly the same as G except that its nonterminals are pairs, consisting of an n-ary concatenation symbol and a state of $\mathfrak{G}$. This grammar is exactly the one we would get by associating with each nonterminal the actual arity that it has in a particular rule. (For example, if we had a rule $S \longrightarrow NP\ VP$ and a rule $S \longrightarrow Aux\ NP\ VP$ then G' would have nonterminals $S/2$ and $S/3$.)

23 For our purposes today we will assume that $\mathfrak{G}$ assigns to a word $w_i \in w$ a state w_i, corresponding to a grammar with "lexical productions" like $Det \longrightarrow the$. It would save a step to have $\mathfrak{G}$ assign to the symbol *the* the state *Det*, of course.

24 **The combined automaton** Putting the pieces together is mainly a matter of turning the input string into the initial assignments of the parse forest automaton. If states of $\mathfrak{G}$ are grammatical categories and states of $yields_w$ are pairs of string position indices, then states of the parse forest automaton $\mathfrak{G} \times yields_w$ will be triples consisting of a grammatical category and the indices of the substring assigned that category. So, for example, the sole final state will obviously be $(S, 0, |w|)$, which we could just as easily write ${}_0S_{|w|}$.

Expressions which connect a grammatical category to the part of the string that it spans are often called *items* (or, more accurately, *completed items*). So the states of the parse forest automaton are items.

25 The transition table for the parse forest automaton $\mathfrak{G} \times yields_w$ is now easily constructed. Given a transition in $\mathfrak{G}$ of the form

$$\delta_{\mathfrak{G}}(B_1, B_2, A, A)$$

(remember that states and symbols are identical in $\mathfrak{G}$: this transition corresponds to the grammar rule $A \longrightarrow B_1\ B_2$), and given that states (B_1, i, j) and (B_2, j, k) are both reachable in the parse forest automaton, we conclude that (A, i, k) is reachable in the parse forest automaton, and that that automaton has in its transition table the transition

$$\delta_{\mathfrak{G} \times yields_w}((B_1, i, j), (B_2, j, k), A, (A, i, k))$$

26 Observe the close connection between transitions in $\mathfrak{G}$ and rules in G. We actually need never construct $\mathfrak{G}$ if we have G itself. We can use the rules of the grammar directly as a slightly peculiar language for writing tree automata.

This is exactly what the Cocke Younger Kasami algorithm does. (This presentation is of course not historically accurate, but is mathematically much more accurate than the historical account would be!) The chief advantages of our approach are that it is easier to construct automata than it is to construct grammars, so we may in fact have the automaton in our hands rather than the grammar. Also it is easier to generalize our procedure to grammars with multiple branching factors: the standard presentation of the CKY algorithm is restricted to grammars in Chomsky Normal Form (i.e., with binary branching nonpreterminals). Of course, it is so easy to translate between automata and grammars that, as soon as you discover interesting approaches from one angle you can translate them to the other if that is more comfortable.

27 Our construction of the parse forest automaton $\mathfrak{G} \times \mathit{yields}_w$ implements a bottom up parse, but one with some interesting properties. Mainly, it behaves as if it were proceeding in parallel, or breadth first, by considering blindly *all* of the possible analyses of the substrings it considers. It does this in a deterministic way, but it is easy to get carried away by the apparent advantages of deterministic computation. Because all analyses of every substring are carried out, the parse forest automaton will contain many *useless states*, i.e., states which can be reached from the input, but from which the final state cannot be reached.

The way to remove useless states is to do a reachability construction *in reverse*, starting from the final states and working backwards. Just as the forwards reachability construction corresponds to bottom up parsing, the backwards reachability construction corresponds to top down parsing. But that is another story . . .