COMS W3261
Computer Science Theory
Chapter 4 Notes

Alexander Roth

2014–09–16

Properties of Regular Langauges

One important kind of fact about the regular languages is called a "closure property." These properties let us build recognizers for languages that are constructed from other languages by certain operations.

Proving Languages Not to Be Regular

We have established that the class of languages known as the regular languages has at least four different descriptions. They are the languages accepted by DFA's, by NFA's, by ϵ-NFA's, and by Regular Expressions.

The Pumping Lemma for Regular Languages

Theorem 1. *(The pumping lemma for regular languages) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:*

1. $y \neq \epsilon$.
2. $|xy| \leq n$.
3. *For all $k \geq 0$, the string xy^kz is also in L.*

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

Closure Properties of Regular Languages

Closure properties express the idea that when one (or several) languages are regular, then certain related languages are also regular. Here is a summary of the principal closure properties for regular languages:

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

Closure of Regular Languages Under Boolean Operations

The boolean operations: union, intersection, and complementation:

1. Let L and M be languages over alphabet Σ. Then $L \cup M$ is the language that contains all strings that are in either or both of L and M.
2. Let L and M be languages over alphabet Σ. Then $L \cap M$ is the language that contains all strings that are in both L and M.
3. Let L be a language over alphabet Σ. Then $\overline{L}$, *the complement of* L, is the set of strings in Σ^* that are not in L.

Closure Under Union

Theorem 2. *If L and M are regular languages, then so is $L \cup M$.*

Closure Under Complementation

Theorem 3. *If L is a regular language over alphabet Σ, then $\overline{L} = \Sigma^* - L$ is also a regular language.*

Closure Under Intersection

Theorem 4. *If L and M are regular languages, then so is $L \cap M$.*

Closure Under Difference

Theorem 5. *If L and MM are regular languages, then so is $L - M$.*

Reverseal

The *reversal* of a string $a_1 a_2 \cdot a_n$ is the string written backwards, that is $a_n a_{n-1} \cdots a_1$. We use w^R for the reversal of string w.

The reversal of a language L, written L^R, is the language consisting of the reversals of all its strings.

Reversal is another operation that preservers regular languages; that is, if L is a regular language, so is L^R.

Given a language L that is $L(A)$ for some finite automaton, perhaps with nondeterminism and ϵ-transitions, we may construct an automaton for L^R by:

1. Reverse all the arcs in the transition diagram for A.
2. Make the start state of A be the only accepting state for the new automaton.
3. Create a new start state p_0 with transitions on ϵ to all accepting states of A.

The result is an automaton that simulates A "in reverse," and therefore accepts a string w if and only if A accepts w^R.

Theorem 6. *If L is a regular language, so is L^R.*

Homomorphisms

A string *homomorphism* is a function on strings that works by substituting a particular string for each symbol.

Formally, if h is a homomorphism on alphabet Σ, and $w = a_1 a_2 \cdot a_n$ is a string of symbols in Σ, then $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. That is, we apply h to each symbol of w and concatenate the results, in order.

Further, we can apply a homomorphism to a language by applying it to each of the strings in the language. That is, if L is a language over alphabet Σ, and h is a homomorphism on Σ, then $h(L) = \{h(w) \mid w \text{ is in } L\}$.

Theorem 7. *If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then $h(L)$ is also regular.*

Inverse Homomorphisms

Homomorphisms may also be applied "backwards," and in this mode they also preserver regular languages. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possibly the same) alphabet T. Let L be a language over alphabet T. Then $h^{-1}(L)$, reads "h inverse of L" is teh set of strings w in Σ^* such that $h(w)$ is in L.

Theorem 8. *If H is a homomorphism from alphabet Σ to alphabet T, and L is a regular language over T, then $h^{-1}(L)$ is also a regular language.*

Decision Properties of Regular Lagnagues

The typical language is infinite, so you cannot present the strings of the language to someone and ask a question that requires them to inspect the infinite set of strings. Rather, we present a language by giving one of the finite representations for it that we have developed: a DFA, an NFA, an ϵ-NFA, or a regular expression.

Of course the language so described will be regular, and in fact there is no way at all to represent completely arbitrary languages. However, for many of the questions we ask, algorithms exist only for the class of regular languages. The same questions become "undecidable" (no algorithm to answer them exists) when posed under more "expressive" notations (i.e. notations that can be used to express a larger set of languages) than the representations we have developed for the regular languages.

In particular, we want to observer the time complexity of the algorithms that perform the conversion. We then consider some of the fundamental questions about languages:

1. Is the language described empty?
2. Is a particular string w in the described language?
3. Do two descriptions of a language actually describe the same language? This question is often called "equivalence" of languages.

Convertin Among Representations

While there are algorithms for any of the conversions,sometimes we are interested not only in the possibility of making a conversion, but in the amount of time it takes. In particular, it is important to distinguish between algorithms that take exponential time (as a function of the size of their input), and therefore can be performed only for relatively small instances, from those that take time that is a linear, quadratic, or some small-degree polynomial of their input size. The latter algorithms are "realistic," in the sense that we expect them to be executable for large instances of the problem.

Converting NFA's to DFA's

When we start with either an NFA or a ϵ-NFA, and convert it to a DFA, the time cna be exponential in the number of states of the NFA. First, computing the ϵ-closure of n states takes $O(n^3)$ time. We must search from each of the n states along all arcs labeled ϵ. If there are n states, there can be no more than n^2 arcs.

Once the ϵ-closure is computed, we can compute the equivalent DFA by the subset construction. The dominant cost is, in principle, the number of states of the DFA, which can be 2^n. For each state, we can compute the transitions in $O(n^3)$ time by consulting the ϵ-closure information and the NFA's transition table fore ach of the input symbols.

The running time of NFA-to-DFA conversion, including the case where the NFA has ϵ-transitions, is $O(n^3 2^n)$. In practice, it is common that the number of states created is much less than 2^n, often only n states. We could state the bound on the running time as $O(n^3 s)$, where s is the number of states the DFA actually has.

DFA-to-NFA Conversion

This conversion is simple, and takes $O(n)$ time on an n-state DFA. ALl that we need to do is modify the transition table for the DFA by putting set-brackets around states and, if the output is an ϵ- NFA, adding a column for ϵ. Since we treat the number of input symbols as a constant, copying and processing the table takes $O(n)$ time.

Automaton-to-Regular-Expression Conversion

We can quadruple the size of teh regular expressions constructed, since each is built from four expression of the previous round. Thus, simply writing down the n^3 expressions can take time $O(n^3 4^n)$.

The same construction works in the same running time if the input is an NFA.

Regular-Expression-to-Automaton Conversion

Conversion of a regular expression to an ϵ-NFA takes linear time. We need to parse the expression efficiently, using a technique that takes only $O(n)$ time on a regular expression of length n. The result is an expression tree with one node for each symbol of the regular expression.

Once we have an expression tree for the regular expression, we can work up the tree, building the ϵ-NFA for each node. The construction rules for the conversion of a regular expression are, never add more than two states and four arcs for any node of the expression tree. Thus, the number of states and the arcs of teh resulting ϵ-NFA are both $O(n)$. Moreover, the work at each node of the parse tree in creating these elements is constant, provided teh function that processes each subtree returns pointers to the start and accepting states of its automaton.

We can eliminate ϵ-transitions from an n-state ϵ-NFA, to make an oridinary NFA in $O(n^3)$ time, without increasing the number of states. However, proceeding to a DFA can take exponential time.

Testing Emptiness of Regular Languages

The problem is not stated with an explicit list of the strings in L. Rather, we are given some representation for L and need to decide whether that representation denotes the language $\emptyset$.

If our representation is any kind of finite automaton, the emptiness question is whether there is any path whatsoever from the start state to some accepting state. If so, the language is nonempty, while if the accepting states are all separated from the start state, then the language is empty.

Testing Membership in a Regular Language

Given a string w and a regular language L, is w in L? While w is represented explicitly, L is represented by an automaton or a regular expression.

If L is represented by a DFA, the algorithm is simple. Simulate the DFA processing the string of input symbols w, beginning in the start state. If the DFA ends in an accepting state, the answer is "yes"; otherwise, the answer is "no". If $|w| = n$, and the DFA is represented by a suitable data structure, then each transition requires constant time, and the entire test takes $O(n)$.

If L has any other representation besides a DFA, we could convert to a DFA and run the test above. That approach could take time that is exponential in the size of the representation, although it is linear in $|w|$. However, if the representation is an NFA or ϵ- NFA, it is simpler and more efficient to simulate the NFA directly. That is, we process symbols of w one at a time, maintaining the set of states the NFA can be in after following any path labeled with that prefix of w.

If w is of length n, and the NFA has s states, then the running time of this algorithm is $O(ns^2)$. Each input symbol can be processed by taking the previous set of states, which numbers at most s states, and looking at the successors of each of these states. We take the union of at most s sets of at most s states each, which requires $O(s^2)$ time.

If the NFA has ϵ-transitions, then we must compute the ϵ-closure before starting the simulation. Then the processing of each input symbol a has two stages, each of which requires $O(s^2)$ time. First, we take the previous set of states and find their successors on input symbol a. Next, we compute the ϵ-closure of this set of states. The initial set of states for the simulation is the ϵ-closure of the initial state of the NFA.

Lastly, if the representation of L is a regular expression of size s, we can convert to an ϵ-NFA with at most $2s$ states, in $O(s)$ time. We then perform the simulation above, taking $O(ns^2)$ time on an input w of length n.

Equivalence and Minimization of Automata

Two languages are *equivalent*, when they define the same language. An important consequence of this is that we may find ways to minimize a DFA. That is, we can take any DFA and find an equivalent DFA that has the minimum number of states. In fact, this DFA is essentially unique: given any two minimum-state DFA's that are equivalent, we can always find a way to rename the states so that the two DFA's become the same.

Testing Equivalence of States

Our goal is to understand when two distinct states p and q can be replaced by a single state that behaves like both p and q. We say that states p and q are *equivalent* if:

- For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

Less formally, it is impossible to tell the difference between equivalent states p and q merely by starting in one of the states and asking whether or not a given input string leads to acceptance when the automaton is started in the (unknown) state. Note that we do *not* require that $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ are the *same* state, only that either both are accepting or both are nonaccepting.

If two states are not equivalent, then we say they are *distinguishable.* That is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. This is where the *table- filling algorithm* comes into play. This algorithm is a recursive discovery of distinguishable pairs in a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

Theorem 9. *If two states are not distinguished by the table-filling algorithm, then the states are equivalent.*

Testing Equivalence of Regular Languages

The table-filling algorithm give us an easy way to test if two regular languages are the same. Supposed languages L and M are each represented in some way, e.g., one by a regular expression and one by an NFA. Convert each representation to a DFA. Now, imagine one dFA whose states are the union of the states of the DFA's for L and M. Technically, this DFA has two start states, but actually the state state is irrelevant as far as testing state equivalence is concerned, so make any state the lone start state.

Now test if the start states of the two original DFA's are equivalent, using the table-filling algorithm. If they are equivalent, then $L = M$, and if not, then $L \neq M$.

The time to fill out the table, and thus to decide whether two states are equivalent is polynomial in the number of states. If there are n states, then there are $\binom{n}{2}$, or $n(n-1)/2$ pairs of states. In one round, we consider al pairs of states, to see if one of their successors pairs has been found distinguishable, so a round takes no more than $O(n^2)$. Moreover, if on some round, no additional

x's are placed in the table, then the algorithm ends. Thus there can be no more than $O(n^2)$ rounds, and $O(n^4)$ is surely an upper bound on the running time of the table-filing algorithm.

Minimization of DFA's

Another important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent.

Theorem 10. *The equivalence of states is transitive. That is, if in some DFA $A = (Q, \Sigma, \delta, q_0, F)$ we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.*

We can use this theorem to justify the obvious algorithm for partitioning states. For each state q, construct a block that consists of q and all the states that are equivalent to q.
First, observe that all states in any block are mutually equivalent. That is if p and r are two states in the block of states equivalent to q, then p and r are equivalent to each other.

Suppose that there are two overlapping, but not identical blocks. That is, there is a block B that includes states p and q, and another block C that includes p but not q. Since p and q are in a block together, they are equivalent. Consider how the block C was formed. If it was the block generated by p, then q would be in C, because those states are equivalent. Thus, it must be that there is some third state s that generated block C; i.e., C is the set of states equivalent to s.

We know that p is equivalent to s, because p is in block C. We also know that p is equivalent to q because they are together in block B. By the transitivity of the above theorem, q is equivalent to s. But then q belongs in block C, a contradiction. We conclude that equivalence of states partitions the states; that is, two states either have the same set of equivalent states (including themselves), or their equivalent states are disjoint.

Theorem 11. *If we create for each state q of a DFA a* block *consisting of q and all the states equivalent to q, then the different blocks of states form a* partition *of the set of states. That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent.*

We are no able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filing algorithm to find all the pairs of equivalent states.
2. Partition the set of state Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by theorem 10. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:
   (a) The start state of B is the block containing the start state of A.
   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Not that if one state of a block is accepting then all the states of that block must be accepting. The reason is that nay accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one bloc of equivalent state.

Why the Minimized DFA Can't Be Beaten

Suppose we have DFA A, and we minimize it to construct a DFA M, using the partitioning method of Theorem 10. That theorem shows that we can't group the states of A into fewer groups and still have an equivalent DFA.

Theorem 12. *If A is a DFA, and M the DFA constructed from A by the algorithm described in the statement of Theorem 10, then M has as few states as any DFA equivalent to A.*

There must be a one-to-one correspondence between the states of any other minimum-state N and the DFA M. The reason is that no state of M can be equivalent to two states of N. We can similarly argue that no state of N can be equivalent to two states of M, although each state of N must be equivalent to one of M's states. Thus, the minimum-state DFA equivalent to A is unique except for a possible renaming of the states.

Summary of Chapter 4

The Pumping Lemma If a language is regular, then every sufficiently long string in the language has a nonempty substring that can be "pumped," that is, repeated any number of times while the resulting strings are also in the language. This fact can be used to prove that many different languages are *not* regular.

Operations That Preserver the Property of Being a Regular Language There are many operations that, when applied to regular languages, yield a regular language as a result. Among these are union, concatenation, closure, intersection, complementation, difference, reversal, homomorphism (replacement of each symbol by an associated string), and inverse homomorphism.

Testing Emptiness of Regular Languages There is an algorithm that, given a representation of a regular language, such as an automaton or regular expression, tells whether or not the represented language is the empty set.

Testing Membership in a Regular Language There is an algorithm that, given a string and a representation of a regular language, tells whether or not the string is in the language.

Testing Distinguishability of States Two states of a DFA are distinguishable if there is an input string that takes exactly one of the two states to an accepting state. By starting with only the fact that pairs consisting of one accepting and one nonaccepting state are distinguishable, and trying to discover additional pairs of distinguishable states by finding pairs whose successors on one input symbol are distinguishable, we can discover all pairs of distinguishable states.

Minimizing Determinisitc Finite Automata We can partition the states of any DFA into groups of mutually indistinguishable states. Members of two different groups are always distinguishable. If we replace each group by a single state, we get an equivalent DFA that has as few states as any DFA for the same language.