CS450
Closure Properties and DFA Minimization

While we have seen that some languages are not regular, certain operations on regular languages are guaranteed to produce regular languages. The following theorems are referred to as the **closure properties** of regular languages. Closure refers to some operation on a language, resulting in a new language that is of the same "type" as those originally operated on, i.e., regular.

We won't be using the closure properties extensively here; consequently we will state the theorems and give some examples. See the book for proofs of the theorems.

1. Closure under Union

   Let L and M be languages over Σ. Then $L \cup M$ is the language that contains all strings that are either in L or in M.

   We have already shown how to compute R+S, where R+S is the union of two regular languages. If L=L(R) and M=L(S) then L(R+S) is the same as $L \cup M$.

   Note that if Σ is different for L and M, then Σ for $(L \cup M)$ is the union of the alphabets for L and the alphabets for M.

2. Closure under Complementation

   Let L and M be languages over Σ. Then $\overline{L}$ is the complement of L, which is the set of strings of Σ^* that are not in L.

   In other words, the complement of a language is everything that is not accepted by the language over our alphabet. Here is an argument as to why complementation is closed. To complement a language:

   - First construct a DFA for that language
   - Complement the accepting states of the DFA

   Note that this requires there be no missing transitions in the DFA. If the automaton "dies" on missing edges, these states will be missing from the complemented DFA (which should be accepting states).

3. Closure under Intersection

   Let L and M be languages over Σ. Then $L \cap M$ is the language that contains all strings that are both in L and M.

To show closure under intersection, note DeMorgan's law which states:

$$L \cap M = \overline{\overline{L} \cup \overline{M}}$$

We have already shown closure under union and complementation, therefore we also have closure under intersection.

4. Closure under Difference

Let L and M be languages over Σ. Then L – M, the difference of L and M, is the set of strings that are in L but not in M.

To show closure, note that: $L - M = L \cap \overline{M}$. Since we have shown closure under intersection and complementation, we also have closure under difference.

5. Closure under Reversal

The reversal of a string $a_1a_2\ldots a_n$ is the string written backwards, that is, $a_na_{n-1}a_{n-2}\ldots a_1$. We will use w^R to denote the reversal of string w. For example, 1011^R is 1101.

The reversal of a language L, written L^R, is the language consisting of the reversals of all its strings. The reversal of a regular language produces another regular language. We can show this informally using an automaton A for L(A):

- Reverse all the edges in the transition diagram for A
- Make the start state of A be the only accepting state for the new automaton
- Create a new start state p_0 with transitions on ε to what were the original accepting states of A

This creates a reversed automaton for A, that accepts a string iff A accepts w^R.

6. Closure under Kleene Star

We have already shown this for regular expressions in the construction of an equivalent ε-NFA for the star operation.

7. Closure under concatenation

We have already shown this for regular expressions in the construction of an equivalent ε-NFA for concatenation.

8. Homomorphisms

We will show closure under homomorphisms, but first we should say what a homomorphism is.

Given a language L with alphabet $\Sigma 1$, A homomorphism h is defined from some alphabet $\Sigma 2$. For symbol(s) $a \in \Sigma 1$, h(a) is some string from $\Sigma 2$. We apply h to each symbol of a word w from L and concatenate the results in order to get a new string. h(L) is the homomorphism of every word in L.

Example: h is the homomorphism from the alphabet {0,1,2} to the alphabet {a,b} defined by h(0) = a, h(1) = ab, h(2) = ba.

h(0120) = aabbaa
h(21120) = baababbaa
$h(01^*2) = a(ab)^*ba$

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

Informally, if L is turned into a regular expression, we are substituting a regular expression for some other regular expression matching the homomorphism. The result is still a regular language.

Note: we can expand homomorphisms to more general substitution, where substituting a substring in L (instead of an individual symbol) with some new string also results in a language that is regular.

9. Inverse Homomorphism

A homomorphism may be applied backwards; i.e. given the h(L), determine what L is. This is denoted as $h^{-1}(L)$, or the inverse homomorphism of L, which results in the **set** of strings w in Σ^* such that h(w) is in L.

Note that while applying the homomorphism to a string resulted in a single string, we may get a set of strings in the inverse.

For example, if h is the homomorphism from the alphabet {0,1,2} to the alphabet {a,b} defined by h(0) = a, h(1) = ab, h(2) = ba.

Given L is the language {ababa} what is $h^{-1}(L)$?
We can construct three strings that map into ababa:
h^{-1}={ 022, 110, 102 }

The result of h^{-1} is also a regular language (see proof in book).

Decision Properties of Regular Languages

Given a (representation, e.g., RE, FA, of a) regular language L, what can we tell about L?

Since there are algorithms to convert between any two representations, we can choose the representation that makes whatever test we are interested in easiest.

Membership: Is string w in regular language L?

- Choose DFA representation for L.
- Simulate the DFA on input w.

Emptiness: Is L = Ø?

- Use DFA representation for L
- Use a graph-reachability algorithm (e.g. Breadth-First-Search or Depth-First-Search or Shortest-Path) to test if at least one accepting state is reachable from the start state. If so, the language is not empty. If we can't reach a accepting state, then the language is empty.

Finiteness: Is L a finite language?

- Note that every finite language is regular (why?), but a regular language is not necessarily finite.
- DFA method:
    - Given a DFA for L, eliminate all states that are not reachable from the start state and all states that do not reach an accepting state.
    - Test if there are any cycles in the remaining DFA; if so, L is infinite, if not, then L is finite. To test for cycles, we can use a Depth-First-Search algorithm. If in the search we ever visit a state that we've already seen, then there is a cycle.

Equivalence: Do two descriptions of a language actually describe the same language? If so, the languages are called equivalent. For example, we've seen many different (and sometimes complex) regular expressions that can describe the same language. How can we tell if two such expressions are actually equivalent?

To test for equivalence our strategy will be as follows:

- Use the DFA representation for the two languages
- Minimize each DFA to the minimum number of needed states
- If equivalent, the minimized DFA's will be structurally identical (i.e. there may be different names for the states, but all transitions will go to identical counterparts in each DFA).

For this strategy to work, we need a technique to minimize a DFA.

We say that two states p and q in a DFA are **equivalent** if:

- For **all** input strings w, δ^(p, w) is an accepting state if and only if δ^(q, w) is an accepting state.

This is saying that if we have reached state p or q, then any other string we might get that will lead to an accepting state from p will also lead to an accepting state from q. Also any string we get that does not lead to an accepting state from p also does not lead to an accepting state from q. If this condition holds, then p and q are equivalent. If this condition does **not** hold, then p and q are **distinguishable**.

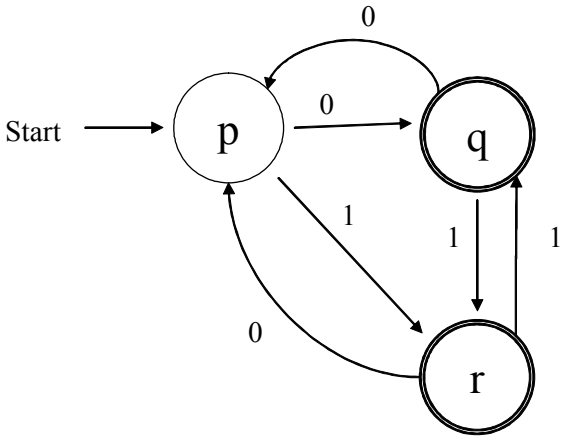
The simplest case of a distinguishable pair is an accepting state p and a non-accepting state q. In this case, δ^(p, ε) is an accepting state, but δ^(q, ∈) is not an accepting state. Therefore, any pair of (accepting, non-accepting) states are distinguishable.

Here is a table-filling algorithm to recursively discover distinguishable pairs in a DFA A:

1. Given an automaton A that has states $q_0 \ldots q_n$ make a diagonal table with labels 1 to n on the rows and 0 to n-1 on the columns.
2. Initialize the table by placing X's for each pair that we know is distinguishable. Initially, this is any pair of accepting and non-accepting states.
3. Let p and q be states such that for some input symbol a, r = δ(p, a) and s = δ(q, a). If the pair (r,s) is known to be distinguishable (i.e. they have an X in their table entry) then the pair p and q are also distinguishable. Place an X for (p,q) in the table.
4. Repeat step 3 until all pairs have been examined.
5. Repeat steps 3-4 again for any empty entries in the table. If no new X's can be placed, then the algorithm is complete. Any entries in the table without an X are pairs of states that are equivalent.

Once the equivalent states have been identified, they can be combined into a single state to make a new, minimized automaton. DFA's have unique minimum-state equivalents.

Simple example: Minimize the following automaton.



Create a table:

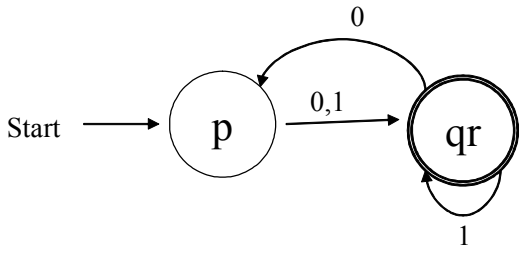
q		
r		
	p	q

Fill an X in for states we know are distinguishable. Initially this is the final state and a non-final state:

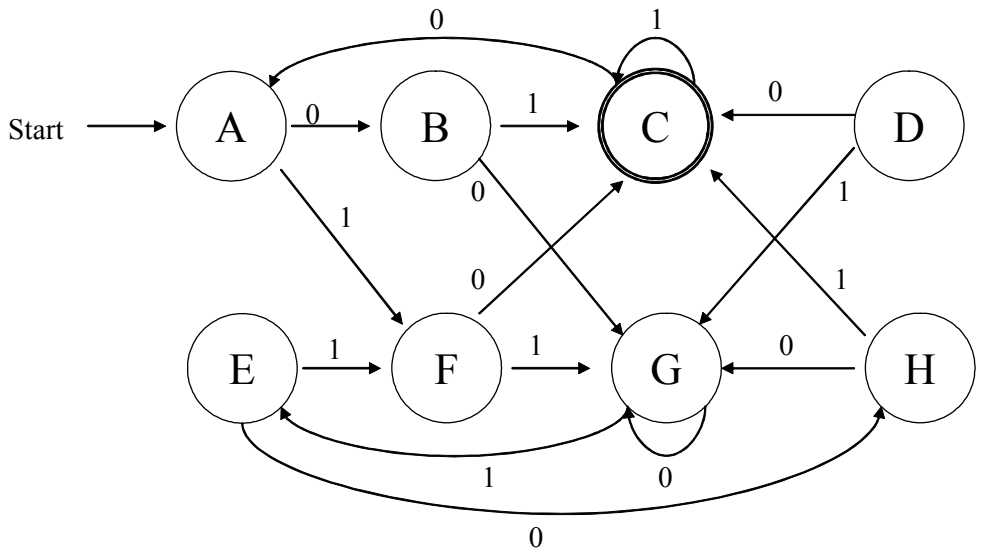
q	X	
r	X	
	p	q

Now check to see if (q,r) is distinguishable. On input symbol 0, both go to state p, so they are not distinguishable on 0. On input symbol 1, r goes to q and q goes to r, giving us the pair (p,r). Looking in the table, (q,r) has no X so we don't put in X in this box. Indeed, we will never put a X in (q,r) because on input symbol 1 we will also refer to itself and it is not distinguishable.

This means that states q and r are equivalent and can be combined. This results in the following, minimized DFA:



Here is the example from the textbook:



Construct the table and initially we can populate X's to with any pair of states that contains C, since C is the sole accepting state.

B							
C	X	X					
D			X				
E			X				
F			X				
G			X				
H			X				
	A	B	C	D	E	F	G

Let's say we look at pair (A,G) first. On input symbol 0, A goes to B and G goes to G. The pair (B,G) is not distinguishable so far, so we have no conclusion. On input symbol 1, A goes to F and G goes to E. (A,F) is also not distinguishable so far, so we have no conclusion and leave (A,G) blank for now.

Let's say we look at (A,H) next. On input symbol 1, H goes to C and A goes to F. The pair (C,F) is known to be distinguishable, so that means (A,H) is also distinguishable and we place an X in the box.

Let's look at (A,F) next. On input symbol 0, F goes to C and A goes to F. (F,C) is known to be distinguishable, so (A,F) is also distinguishable and we place an X in the box.

If we continue this way for all states, we can almost fill in the entire table on the first pass. This is because almost all of the states go to C, which we know to be distinguishable with everything else.

B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G		X	X	X	X	X	
H	X		X	X	X	X	X
	A	B	C	D	E	F	G

We can't quit yet though, we have to continue iterating until no more X's can be found. This is because when we made a check to see if a pair is distinguishable, we might have done that check before we placed an X in the box.

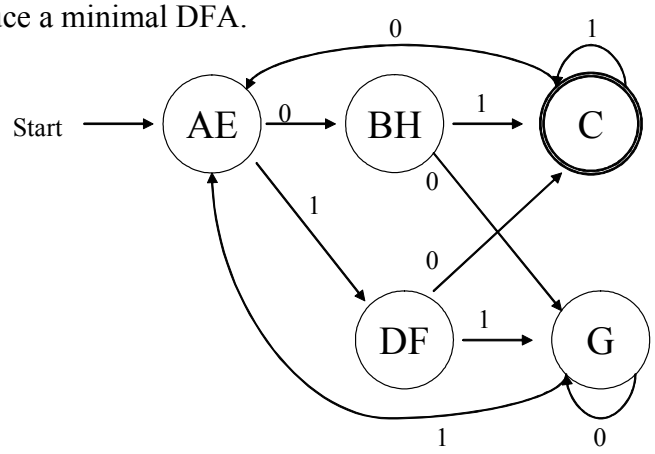
Let's examine (A,G) first. input symbol 0, A goes to B and G goes to G. The pair (B,G) we now know to be distinguishable. That means that (A,G) is also distinguishable and we can place an X in the box.

Let's examine (A,E) next. On input symbol 1, both A and E go to F so that is not distinguishable. On input symbol 0, A goes to B and E goes to H. (B,H) is not known to be distinguishable either, so we can't place an X in the box for (A,E).

If we continue to check all the unmarked states, no other states can be distinguished and we will finally be complete on the next iteration:

B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X
	A	B	C	D	E	F	G

This means that states (B,H), (A,E) and (D,E) are equivalent. These states can then be combined to produce a minimal DFA.



While we can perform the minimization technique to reduce and simplify a DFA, the reason we ended up here in the first place was to use minimization to show two DFA are equivalent. If we minimize two different DFA's and both are structurally identical, then the two DFA's (and the language they represent) are equivalent.

Why the Minimized DFA Can't Be Beaten

We took a DFA A and minimized it to produce an equivalent, minimal, DFA M. Could there be another DFA, N, that is unrelated to A that accepts the same language as A but yet has fewer states than M?

We can prove by contradiction that such a DFA N does not exist.

- First, run the state-distinguishability process on the states of M and N together, as if they were one DFA but with no interaction.
- The start states of M and N are indistinguishable since L(M) = L(N).
- If (p,q) are indistinguishable, then their successors on any one input symbol are also indistinguishable. This is because if we could distinguish the successors, then we could have distinguished p from q.
- M must not have an inaccessible state, or else we would have eliminated it when we produced M in the first place.
- We are assuming that N has fewer states than M. If this is so, then there are two states of M that are indistinguishable from the same state of N, and therefore indistinguishable from each other.
- But M was designed so that all states are distinguishable from each other. We have a contradiction, therefore our assumption that N has fewer states than M must be false.

The end result is that other DFA exists with fewer states that is equivalent to the minimized DFA. We can get an even stronger result, that there is a 1:1 correspondence between the states of N and M meaning that the minimized DFA is unique.