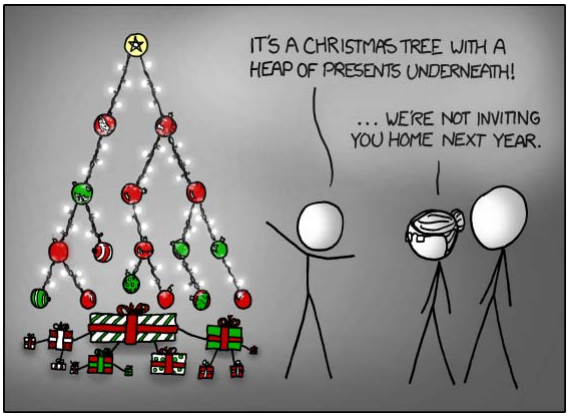
MA/CSSE 474

Theory of Computation

Minimizing DFSMs

Your Questions?

- Previous class days' material
- Reading Assignments
- HW4 or HW5 problems
- Tuesday's Exam
- Anything else



Even in the relatively narrow area of Computer science, the term "Heap" is ambiguous. From Wikipedia:

Computer science

- Heap (data structure), a data structure commonly used to implement a priority queue
- Heap (programming) (or free store), an area of memory used for dynamic memory allocation

For practice later: A Simple Example of $\approx_L$

Σ = {a, b}
L = {$w \in \Sigma^*$: $|w|$ is even}

ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\approx_L$:

Recall that x $\approx_L$ y iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.

Another Example of $\approx_L$

Σ = {a, b}
L = aab*a

Do this one for practice later

ε	bb	aabaa
a	aba	aabbba
b	aab	aabbaa
aa	baa	
	aabb	

The equivalence classes of $\approx_L$:

Recall that x $\approx_L$ y iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.

More Than One Class Can Contain Strings in *L*

$\Sigma = \{a, b\}$
$L = \{w \in \Sigma^* : \text{no two adjacent characters are the same}\}$

The equivalence classes of $\approx_L$:

[1]	[ε]
[2]	[a, aba, ababa, …]
[3]	[b, ab, bab, abab, …]
[4]	[aa, abaa, ababb…]

Recall that $x \approx_L y$ iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.

One More Example of $\approx_L$

$\Sigma = \{a, b\}$
$L = A^nB^n = \{a^nb^n : n \geq 0\}$

ε	aa	aaaa
a	aba	aaaaa
b	aaa	

The equivalence classes of $\approx_L$:

Recall that $x \approx_L y$ iff $\forall z \in \Sigma^* (xz \in L \leftrightarrow yz \in L)$.

Important results

(we may prove some of them later)

- Let *M* be a DFSM that accepts the regular language *L*. The number of states in *M* is greater than or equal to the number of equivalence classes of $\approx_L$.
- ***Theorem*:** Let *L* be a regular language over some alphabet Σ. Then there is a DFSM *M* with L(M)=L and that has precisely *n* states where *n* is the number of equivalence classes of $\approx_L$. Any other FSM that accepts *L* must either have more states than *M* or it must be equivalent to *M* except for state names.

Construct DFSM from $\approx_L$

M = (*K*, Σ, δ, *s*, *A*), where:

- *K* contains one state for each equivalence class of $\approx_L$.
- *s* = [ε], the equivalence class of ε under $\approx_L$.
- *A* = {[*x*] : *x* $\in$ *L*}. **[Aside: what is the union of all of these classes?]**
- δ([*x*], *a*) = [*xa*]. (if *M* is in state containing *x*, then, after reading the next symbol, *a*, it goes to state containing *xa)*.

We can show (but we won't right now):

- ***K* is finite.**
- **δ is a well-defined function*.** i.e. $\forall x,y\in\Sigma^*,a\in\Sigma$ ([x]=[y] → [xa]=[ya])
- ***L* = *L*(*M*).**

Example

Σ = {a, b}
L = {$w \in \Sigma^*$: no two adjacent characters are the same}

The equivalence classes of $\approx_L$:

1: [ε]	ε
2: [a, ba, aba, baba, ababa, ...]	$(b \cup \varepsilon)(ab)^*a$
3: [b, ab, bab, abab, ...]	$(a \cup \varepsilon)(ba)^*b$
4: [bb, aa, bba, bbb, ...]	the rest

- Equivalence classes become states
- Start state is [ε]
- Accepting states are all equivalence classes in L
- $\delta([x], a) = [xa]$

The Myhill-Nerode Theorem

Theorem: A language is regular iff the number of equivalence classes of $\approx_L$ is finite.

Proof: Show the two directions of the implication:

L regular $\rightarrow$ the number of equivalence classes of $\approx_L$ is finite: If L is regular, then

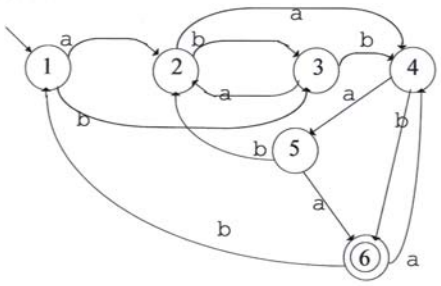
The number of equivalence classes of $\approx_L$ is finite $\rightarrow$ L regular: If the cardinality of $\approx_L$ is finite, then

So Where Do We Stand?

1. We know that for any regular language L there exists a minimal accepting machine M_L.
2. We know that $|K|$ of M_L equals the number of equivalence classes of $\approx_L$.
3. We know how to construct M_L from $\approx_L$.
4. We know that M_L is unique up to the naming of its states.

But is this good enough?

Consider:



Minimizing an Existing DFSM (Without Knowing $\approx L$)

Two approaches:

1. Begin with M and collapse redundant states, getting rid of one at a time until the resulting machine is minimal.
2. Begin by overclustering the states of M into just two groups, accepting and nonaccepting. Then iteratively split those groups apart until all the distinguishable states have been distinguished.

The Overclustering Approach

We need a definition for "equivalent", i.e., mergeable states.

Define $p \equiv q$ iff for every string $w \in \Sigma^*$, either w takes M to an accepting state from both q and p or it takes M to a rejecting state from both q and p.

Is ≡ an equivalence relation?

Construct ≡ as the Limit of a Sequence of Approximating Equivalence Relations $\equiv^n$

(Where n is the length of the input strings that have been considered so far)

Consider input strings, starting with ε, and increasing in length by 1 at each iteration. Start by overclustering the states. Then split them apart as it becomes apparent (with longer and longer strings) that they are not equivalent

Constructing $\equiv^n$

- $p \equiv^0 q$ iff they behave equivalently when they read ε. In other words, if they are both accepting or both rejecting states.
- $p \equiv^1 q$ iff $p \equiv^0 q$ *and* they behave equivalently when they read any string of length 1, i.e., if any single character sends both of them to an accepting state or both of them to a rejecting state. Note that this is equivalent to saying that any single character sends them to states that are $\equiv^0$ to each other.
- $p \equiv^2 q$ iff $p \equiv^1 q$ and they behave equivalently when they read any string of length 2, which they will do if, when they read the first character they land in states that are $\equiv^1$ to each other. By the definition of $\equiv^1$, they will then yield the same outcome when they read the single remaining character.
- And so forth.

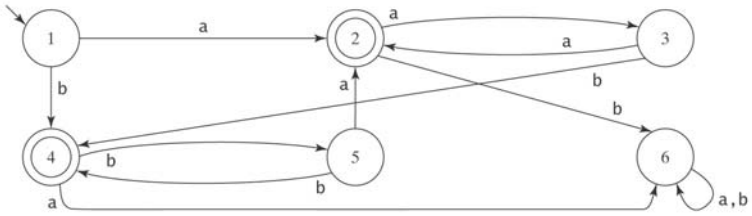
Constructing $\equiv$, Continued

More precisely, $\forall p,q \in K$ and any $n \geq 1$, $q \equiv^n p$ iff:

1. $q \equiv^{n-1} p$, and
2. $\forall a \in \Sigma\ (\delta(p, a) \equiv^{n-1} \delta(q, a))$

An Example

Σ = {a, b}



Equivalence classes:

$\equiv^0$ =

$\equiv^1$ =

$\equiv^2$ =

MinDFSM

MinDFSM(*M*: DFSM) =

1. *classes* := {*A*, *K-A*};
2. Repeat until no changes are made
   - 2.1. *newclasses* := $\varnothing$;
   - 2.2. For each equivalence class *e* in *classes*, if *e* contains more than one state do
     - For each state *q* in *e* do
       - For each character *c* in Σ do
         - Determine which element of *classes q* goes to if *c* is read
     - If there are any two states *p* and *q* that need to be split, split them. Create as many new equivalence classes as are necessary. Insert those classes into *newclasses*.
     - If there are no states whose behavior differs, no splitting is necessary. Insert *e* into *newclasses*.
   - 2.3. *classes* := *newclasses*;
3. Return M^* = (*classes*, Σ, δ, $[s_M]$, {[*q*: the elements of *q* are in A_M]}), where δ_{M^*} is constructed as follows:
   - if $\delta_M(q, c) = p$, then $\delta_{M^*}([q], c) = [p]$

Summary

- Given any regular language *L*, there exists a minimal DFSM *M* that accepts *L*.
- *M* is unique up to the naming of its states.
- Given any DFSM *M*, there exists an algorithm *minDFSM* that constructs a minimal DFSM that also accepts *L*(*M*).

Canonical Forms

A ***canonical form*** for some set of objects *C* assigns exactly one representative to each class of "equivalent" objects in *C*.

Further, each such representative is distinct, so two objects in *C* share the same representation iff they are "equivalent" in the sense for which we define the form.

In order for a canonical form to be useful, there must be a procedure which, given an object from the set, computes its canonical form.

A Canonical Form for FSMs

buildFSMcanonicalform(*M*: FSM) =

1. *M′* = *ndfsmtodfsm*(*M*).
2. *M** = *minDFSM*(*M′*).
3. Create a unique assignment of names to the states of *M**.
4. Return *M**.

The simple algorithm for unique name assignment is in the textbook; we will illustrate it here by doing an example.

Given two FSMs M_1 and M_2:

buildFSMcanonicalform(M_1)
=
buildFSMcanonicalform(M_2)

iff $L(M_1) = L(M_2)$.