EECS 376
MIDTERM STUDY GUIDE

by Evan Hahn, Scott Godbold, Cam Herringshaw, Stephen Rolfe, Zachariah Gerth (add your name if you helped!)

Good luck on the test, everyone!

Exam logistics

- One side of a 8.5” × 11” piece of paper (Evan and Scott made this one)
- In class
- List of concepts: part 1, part 2

At a high level

There are many different computational models of computers, just like there are many mathematical models of physics and other stuff. These include *finite automata* and *Turing machines*, the two major parts of this exam.

Deterministic finite automata

Deterministic (DFA): each state *must* have exactly one transition arrow for every item in the alphabet, and it may only occupy a single state at a time.

Formally described by the 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

1. Q: a finite set of *states*. For example, if there are 3 states, you would have the set $\{q_0,q_1,q_2\}$.
2. Σ: a finite set called the *alphabet*, containing all symbols that automata accepts. For example, {0,1} could be the alphabet.
3. δ: transition function. The domain (inputs) is $Q \times \sum$ and the range is Q, which can be written as $Q \times \sum \rightarrow Q$.
4. q_0: Q is the start state. $q_0 \in Q$.
5. F: set of accept states. Can be empty set. $F \subseteq Q$.

Here's a DFA with the corresponding 5-tuple describing it:

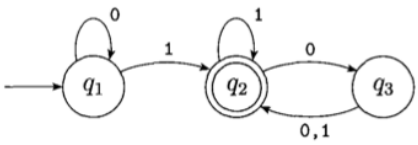


FIGURE 1.6
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

- equivalent statements about the set of strings A and machine M:
  - M accepts A
  - M recognizes A
  - A is the language of machine M
  - L(M) = A

- regular language: any language that has a finite automata (DFA or NFA) able to recognize it

Regular operations

Union ($\cup$): returns a set containing all elements that appear in either set (usage: $A \cup B$). Formally, $A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$.

Concatenation ($\circ$): returns a set containing all combinations of an element from set A and an element from set B (usage: $A \circ B$)

Star (*): returns a set containing all words over the alphabet A. (usage: A^*)

An example of each:

EXAMPLE 1.24

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,

$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$, and

$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

Non-deterministic finite automata

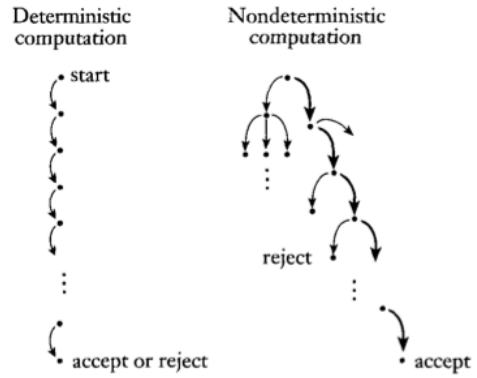
A more generalized form of a DFA, each state does not need a transition arrow for each element in the alphabet, the NFA may have more than one active state, it also may have more than one transition arrow for a given element in the alphabet,

NFA's also have a special symbol ε which is also taken when present and does not "consume" a letter out of the string being passed in.

Generally speaking NFA's are much less complex than its corresponding DFA.

Formally described by the 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

1. Q is a finite set called states, so if there are 3 states, you would have the set $\{q_0, q_1, q_2\}$
2. Σ is a finite set called the alphabet, containing all symbols that automata accepts
    a. (ie. {0,1} could be the alphabet)
3. $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function. So each item in δ describes what each item of the alphabet does at each state in the automata. Where Σ_ε is $\Sigma \cup \{\varepsilon\}$, and P(Q) is the powerset of Q.
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept (final) states



An NFA and the corresponding DFA:

EXAMPLE 1.30

Let A be the language consisting of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N_2 recognizes A.

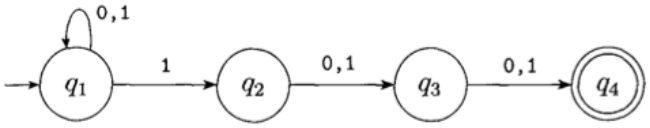


FIGURE 1.31
The NFA N_2 recognizing A

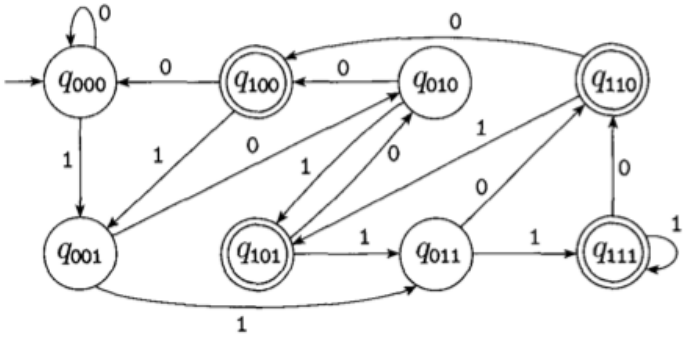
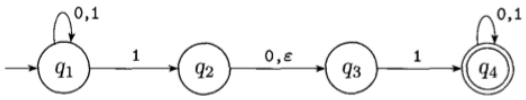


FIGURE 1.32
A DFA recognizing A

A NFA and its corresponding 5-tuple:

EXAMPLE 1.38

Recall the NFA N_1:



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 is the start state, and
5. $F = \{q_4\}$.

Closure & projection

Closure: the idea that any of the regular operations performed on two regular languages will result in another regular language.

Projection: Best way to think of projection as far as I am concerned is like a shadow, so {A × B × C} projected with {A × C} will leave you with {A × C}. You can also think of it as the intersection where all elements in both sets will be present in the new set.

If *s* is a string, and Σ is an alphabet, the **string projection** of *s* is the string that results by removing all letters which are not in Σ. It is written as $\pi_\Sigma(s)$. It is formally defined by removal of letters from the right hand side:

$$\pi_\Sigma(s) = \begin{cases} \varepsilon & \text{if } s = \varepsilon \text{ the empty string} \\ \pi_\Sigma(t) & \text{if } s = ta \text{ and } a \notin \Sigma \\ \pi_\Sigma(t)a & \text{if } s = ta \text{ and } a \in \Sigma \end{cases}$$

Regular expressions

Formal definition:

1. *a* for some a in the alphabet Σ

2. ε
3. $\emptyset$
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*), where R_1 is a regular expressions

Where *a* and ε represent the languages {a} and {ε} accordingly, and $\emptyset$ is the empty language. The remaining show what happens when a close property is used on the 3 base languages.

Symbols

- Σ: any symbol in the alphabet (for most example purposes this will be {0,1})
- *: repeat the previously stated character 0 or more times (ie. 1*, could be “”, “1”, “11”, “111”...)
- +: repeat the previously stated character 1 or more times (ie. the same as before but this time it must appear at least once)

Definitions

- $R \cup \emptyset = R$
    - Adding the empty language to any other language will not change it (if that wasn’t apparent)
- $R \circ \varepsilon = R$
    - Joining the empty string to any string is the exact same string

DFA ↔ NFA ↔ regex

Every DFA can be represented by a corresponding NFA and regular expression.

Example of regex → NFA:

EXAMPLE 1.58

In Figure 1.59, we convert the regular expression $(a \cup b)^*aba$ to an NFA. A few of the minor steps are not shown.

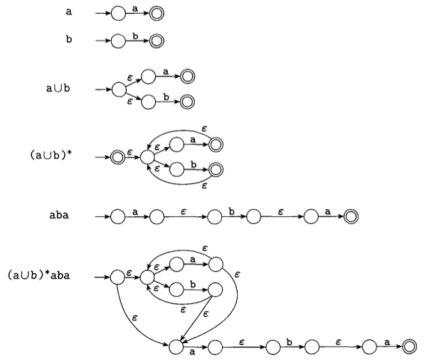


FIGURE 1.59
Building an NFA from the regular expression $(a \cup b)^*aba$

DFA to Regex

This is aided by a special type of finite automata, called a ***generalized nondeterministic finite automaton*** defined as the following 5-tuple (Q, Σ, δ, q_{start}, q_{accept})

1. Q is the finite set of states
2. Σ is the input alphabet
3. δ: $(Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \rightarrow R$ is the transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

The conversion follows the following function:

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$, where

$$Q' = Q - \{q_{rip}\},$$

and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

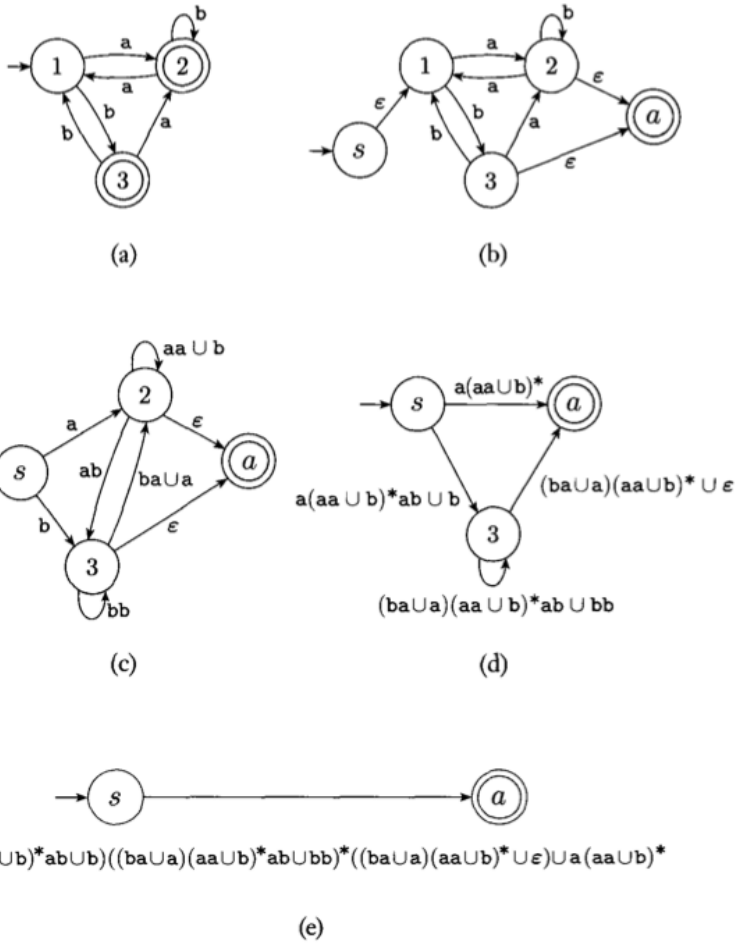
for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute CONVERT(G') and return this value.

And of course the obligatory example problem:

EXAMPLE 1.68

In this example we begin with a three-state DFA. The steps in the conversion are shown in the following figure.



Nonregular Languages

Any language that cannot be determined in a finite amount of states on a DFA or NFA is called a *Nonregular Language.* Good examples are languages that require the knowledge of a previous count to prove them (this is not all inclusive) such as a language that has some number of 0's followed by the same number of 1's.

Pumping Lemma

*This is completely copy-pasted from [this great Stack Overflow answer](): *

The pumping lemma is a simple proof to show that a language is not regular, meaning that a Finite State Machine cannot be built for it. The canonical example is the language `(a^n)(b^n)`. This is the simple language which is just any number of `a`s, followed by the same number of `b`s. So the strings

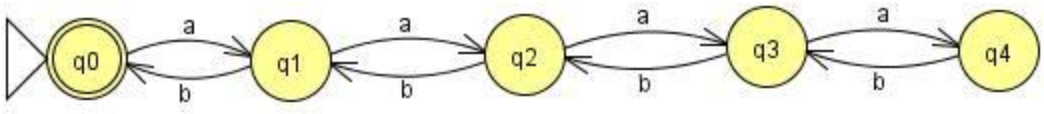
```
ab
aabb
aaabbb
aaaabbbb
```

etc. are in the language, but

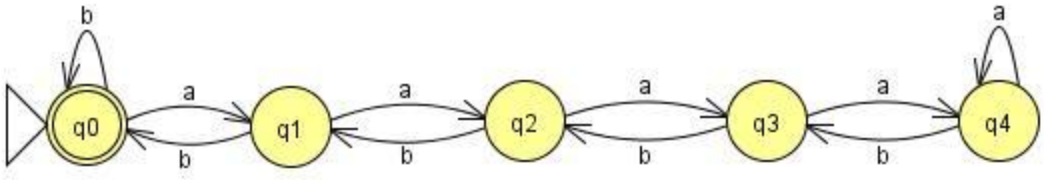
```
aab
bab
aaabbbbbb
```

etc. are not.

It's simple enough to build a FSM for these examples:



This one will work all the way up to n=4. The problem is that our language didn't put any constraint on n, and Finite State Machines have to be, well, finite. No matter how many states I add to this machine, someone can give me an input where n equals the number of states plus one and my machine will fail. So if there can be a machine built to read this language, there must be a loop somewhere in there to keep the number of states finite. With these loops added:



all of the strings in our language will be accepted, but there is a problem. After the first four `a`s, the machine loses count of how many `a`s have been input because it stays in the same state. That means that after four, I can add as many `a`s as I want to the string, without adding any bs, and still get the same return value. This means that the strings:

`aaaa(a*)bbbb`

with `(a*)` representing any number of `a`s, will all be accepted by the machine even though they obviously aren't all in the language. In this context, we would say that the part of the string `(a*)` can be pumped. The fact that the Finite State Machine is finite and n is not bounded, guarantees that any machine which accepts all strings in the language MUST have this property. The machine must loop at some point, and at the point that it loops the language can be pumped. Therefore no Finite State Machine can be built for this language, and the language is not regular.

Peggy versus Victor

- P: select p ≥ 0
- V: select S inside of A
- P: split S into xyz, where |xy| ≤ p |y| > 0
- V: select how many times to repeat y
- If the new S violates A, V wins, else P wins
- V winning == not regular

Strategy: ONLY PLAY ONE SIDE.

If you pick a move for Peggy, when trying to prove a language not regular, you ruin the proof. If you play as Peggy - ie. prove that the Pumping Lemma doesn't apply (**winning the game as Peggy does not say a language is regular, just that it might not be ..not regular.)** then you try to define your y such that it doesn't matter if it exists, or if it's large or small, etc.

If you play as Victor, try to define a length p such that Peggy's choice of y is constrained, or worse, contains a necessary part of the regular language. If the latter, you can define the times to repeat y as 0, and now it is proved not-regular. If the former, you can make it repeat until invalid - but only if it is not a regular language.

Turing machines

Differences between a Turing Machine and finite automata:

1. A Turing Machine can both read from / write to the tape.
2. The read-write head can move both left and right.
3. The tape is infinite.
4. The special states for accepting/rejecting take effect immediately.

Formal Definition:

A Turing machine is a 7-tuple ($Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}$)

1. Q is the set of states
2. Σ is the input alphabet not containing the blank symbol $\sqcup$
3. Γ is the tape alphabet where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. δ is $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state
7. $q_{reject} \in Q$ is the reject state where $q_{reject} \neq q_{accept}$

When talking about Turing Machines there are 3 main kinds that we speak of:

1. Decider: A turing machine that halts on all inputs(either accepts or rejects). These are similar to a DFA in that sense.
2. Recognizer: A turing machine that either reaches accept or loops. A recognizer can run infinitely (see the Halting Problem). They are also the default for a turing machine
3. Enumerator: Lists elements of a language rather than considering strings.

Decidability

DFA Acceptance Problem

Let $A_{DFA} = \{ < B, w > \mid B \text{ is a DFA that accepts input string } w \}$

- Need to present a TM *M* that decides A_{DFA}.
- *M* = "On input <*B*,*w*>, where *B* is a DFA and *w* is a string:
    a. Simulate *B* on input *w*.
    b. If the simulation ends in an accept state, *accept*. Otherwise, *reject*."
- When *M* receives its input, it checks whether the input is actually a DFA and string. If not, it rejects.
- *M* then carries out the simulation, keeping track of *B*'s state and position on its tape.
- When *M* finishes processing the last symbol of *w*, it *accepts* if *B* is in an accepting state and *rejects* otherwise.

The Halting Problem (Theorem 4.11)

$A_{TM} = \{ < M, w > \mid M \text{ is a TM and } M \text{ accepts } w \}$

Theorem: A_{TM} is undecidable.

Proof:

- Assume A_{TM} is decidable.

- Suppose *H* is a decider for A_{TM}.
    - $H(< M, w >) = \{ accept\ if\ M\ accepts\ w,\ reject\ otherwise \}$
- Construct a new TM *D* that calls *H* to determine what *M* does when *M* is its own input.
- *D* = "On input <*M*>:
    - Run *H* on input <*M*, <*M*> >.
    - Output the opposite of what *H* outputs.
- Example of *D*:
    - $D(< M >) = \{ accept\ if\ M\ does\ not\ accept\ < M >$
      $\{reject\ if\ M\ accepts\ < M >$
- Run *D* with itself as input:
    - $D(< D >) = \{accept\ if\ D\ does\ not\ accept\ < D >$
      $\{reject\ if\ D\ accepts\ < D >$
- No matter what *D* does, it is forced to do the opposite ⇒ contradiction.
- Thus, neither *D* nor *H* can exist.
- Therefore, A_{TM} is undecidable.

Theorem 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable. This means that both the language and its complement are Turing recognizable.

Rice's Theorem

Rice's Theorem states that a property of recognizable languages is itself a recognizable language. Because of this, 'regularity' is a property. The theorem goes on to say that if a language L has a nontrivial property P, such that {<M> | : L(M) is in P} , it is **not** decidable.

In laymen's terms, this indicates that if a language has some nontrivial property (defined below) it is not decidable, allowing us to more quickly recognize a decidable language.

Non-Trivial Property

First, let us define a property as just a set of recognizable languages, e.g. regularity, two a's, etc. We can further define a non-trivial property, P, to be a language which contains and avoids at least one recognizable language, the two being a different recognizable language. Once again, more simply defined: P is any property which requires more computing capability than can effectively be used, and thus it cannot be decided.

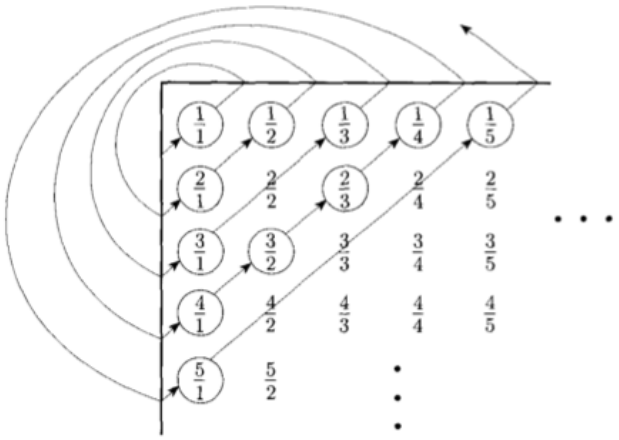
They are different for finite automata and Turing machines.

It is important to note that Rice's Theorem does not say anything about if a given language is **recognizable**.

Countability

A set is considered countable if it either has a finite number of elements or it has the same size as the set of natural numbers (ie. {0, 1, 2, 3, 4, 5, …}).

Example of a countable, infinite set:



- Theorem 4.17: Real numbers are uncountable.

Theorem list

These are lifted from the book.

- The class of regular languages is closed under the union operation
- The class of regular languages is closed under the concatenation operation
- The class of regular languages is closed under the star operation
- Every nondeterministic finite automaton has an equivalent deterministic finite automaton
- A language is regular iff some nondeterministic finite automaton recognizes it
- A language is regular iff some regular expression describes it
- If a language is described by a regular expression, then it is regular
- If *A* is a regular language, then there is a number *p* (the pumping length) where, if s is any string in *A* of length at least *p*, then *s* may be divided into three pieces, *s* = *xyz*, satisfying the following conditions:
    - for each i >= 0, xy^iz in *A*
    - |*y*| > 0

    - $|xy| <= p$
- Every multitape Turing machine has an equivalent single tape turing machine
- A language is Turing-recognizable iff some multitape Turing machine recognizes it
- A language is Turing-recognizable iff some enumerator enumerates it
- Regular expressions, NFAs, and DFAs are decidable
- Every context-free language is decidable
- The set of real numbers is uncountable
- Some languages are not Turing-recognizable
- A language is decidable iff it is Turing-recognizable and co-Turing recognizable
- Regular languages are decidable
- If M is a linear bounded automaton (Turing machine where tape head states inside of the input) where L(M) = Ø, then the machine of M is undecidable

Study guide info

The accuracy of this study guide is not guaranteed.

Enjoy!