CPSC 121: Models of Computation

Unit 13: DFAs

Based on slides by Patrice Belleville and Steve Wolfman

Learning Goals

For the exam:

By the end of this unit, you should be able to :

- Build a DFA to recognize a particular language.
- Identify the language recognized by a particular DFA.
- Connect the graphical and formal representations of DFAs and illustrate how each part works.

Learning Goals

For yourself:

By the end of this unit, you should be able to

- Use the formalisms that we've learned so far (especially power sets and set union) to illustrate that nondeterministic finite automata are no more powerful than deterministic finite automata.
- Demonstrate through a contradiction argument that there are interesting properties of programs that cannot be computed ***in general***.
- Describe how the theoretical story 121 has been telling connects to the branch of CS theory called automata theory.

?Addressing the Course Big Questions ?

- We will now establish **profound results about** the power (and lack thereof) of our DFA model and of general **computational systems**.
- We will discuss the very general questions:
  - What can we compute?
  - Are there problems we can not solve?

Outline

- ❑ Formally Specifying DFAs
- ❑ Designing DFAs
- ❑ Analyzing DFAs: Can DFAs Count?
- ❑ Non-Deterministic Finite Automata: Are They More Powerful than DFAs?
- ❑ CS Theory and the Halting Problem

Reminder: Formal DFA Specification

Formally, a DFA consist of :

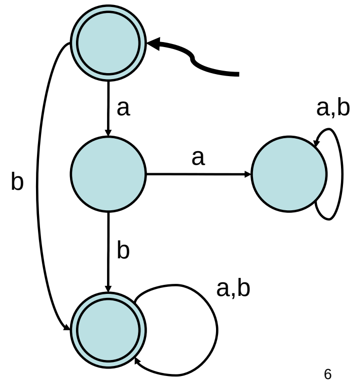
I a (finite) set of letters in the input language.

S a (finite) set of states.

$\mathbf{s_0}$ a start state; $s_0 \in S$

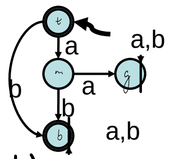
F a set of accepting states; $F \subseteq S$

N : $S \times I \rightarrow S$ is the transition function.



One Way to Formalize DFA Operation

❑ The algorithm "**RunDFA**" runs a given DFA on an input, resulting in "**Yes**" or "**No**". (We assume valid input drawn from **I**.)

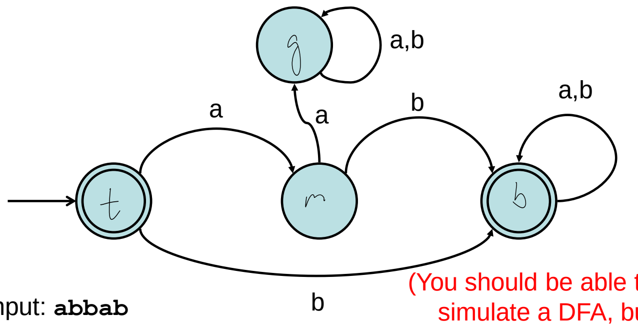


❑ **RunDFA((I, S, s$_0$, F, N), input):**

```
If input is empty, then:
    if $s_0 \in F$, return "Yes", else return "No".
Otherwise, let s' = N(s$_0$, (first input)).
    return RunDFA((I, S, s', F, N),
                (rest input)).
```

Running the DFA

How does the DFA runner work?



Input: **abbab**

(You should be able to simulate a DFA, but needn't use RunDFA.)

Outline

- ❑ Formally Specifying DFAs
- ❑ Designing DFAs
- ❑ Analyzing DFAs: Can DFAs Count?
- ❑ Non-Deterministic Finite Automata: Are They More Powerful than DFAs?
- ❑ Combining DFA Languages
- ❑ CS Theory and the Halting Problem

Strategy for Designing a DFA

To design a DFA for some language, try:

1. Write out *good* test cases of strings to accept/reject. Always include the empty string!
2. If the empty string should be accepted, draw an accepting start state; else, draw a rejecting one.
3. From that state, consider what each letter would lead to. Group letters that lead to the same outcome. (Some may lead back to the start state if they're no different from "starting over".)
4. Decide for each new state whether it's accepting.
5. Repeat steps 3 and 4 for new states as long as you have unfinished states, always trying to reuse existing states!

As you go, give as descriptive names as you can to your states, to figure out when to reuse them!

Example on Designing a DFA

- ❑ Problem: **Design a DFA to Recognize "Decimal Numbers"**
- ❑ Strategy:
  - ➢ First try some examples of strings that are good and bad decimal numbers
  - ➢ Then we will create a regular expression for them
  - ➢ Then design a DFA for the regular expression

Is this a decimal number?

Which of the following is a decimal number?

```
3.5   2.011    -6      5.       -3.1415926536

.25   --3      3-5     5.2.5    .
```

Rules for whether something is a decimal number?

Writing the Regular Expression

Something followed by `?` is optional.

Anything followed by a `+` can be repeated.

`[0-9]` is a digit, an element of the set `{0, 1, …, 9}`.

`-` is literally `-`. `\.` means `.`

`()` are used to group elements together.

Regular expression for decimal numbers:
`-?[0-9]+(\.[0-9]+)?`

Designing the DFA

Problem: Design a DFA to recognize decimal numbers, defined using the regular expression: `-?[0-9]+(\.[0-9]+)?`

More DFA Design Problems

Design a DFA that recognizes words that have two or more contiguous sequences of vowels. (This is a rough stab at "multisyllabic" words.)

Design a DFA that recognizes base 4 numbers with at least one digit in which the digits are in sorted order.

Design a DFA that recognizes strings containing the letters **`a`**, **`b`**, and **`c`** in which all the **`a`**s come before the first **`c`** and no two **`b`**s neighbour each other.

Outline

- ❑ Formally Specifying DFAs
- ❑ Designing DFAs

- ❑ Analyzing DFAs: Can DFAs Count?
- ❑ Non-Deterministic Finite Automata: Are They More Powerful than DFAs?
- ❑ CS Theory and the Halting Problem

Can a DFA Count?

Problem: Design a DFA to recognize a^nb^n: the language that includes all strings that start with some number of `a's` and end with the same number of `b's`.

For example, these should be accepted: the empty string, **ab**, **aabb**, **aaabbb**, **aaaabbbb**, and **aaaaaaaaabbbbbbbbb**.

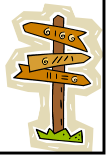
These should be rejected: **a**, **b**, **aab**, **abb**, **aaaaaaaaabb**, and **aaaaaaaabbbbbbbbbb**.

DFAs Can't Count

- **Theorem:** There is no DFA that accepts a^nb^n for any n in N
- The proof is in the appendix 1
- The proof is by contradiction. Assume some DFA recognizes the language a^nb^n and conclude that this DFA also accepts a string a^kb^n with k<n.
- The heart of the proof is the insight that a DFA can only have a finite number of states and so can only "count up" a finite number of **a's**.

Outline

- Formally Specifying DFAs
- Designing DFAs
- Analyzing DFAs: DFAs Can't Count
- Non-Deterministic Finite Automata: Are They More Powerful than DFAs?
- CS Theory and the Halting Problem



Let's Try Something More Powerful (?): NFAs

- A Non-Deterministic Finite-State Automaton (NFA) is like a DFA except:
  - Any number (zero or more) of arcs can lead from each state for each letter of the alphabet
  - There may be many start states
- As a DFA runs through each letter of its input, it moves from the **current state** to the **next state**. It is always in exactly one state.
- As an NFA runs through each letter of its input, it moves from the **current set of states** to the **next set of states**. At any time the NFA is considered to be at many states.

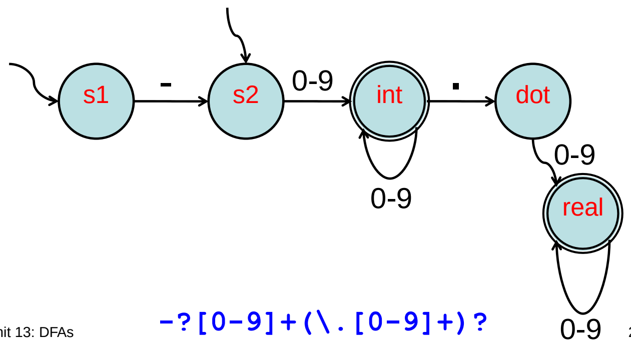
NFAs Continued

- An NFA is like a DFA except:
  - Any number (zero or more) of arcs can lead from each state for each letter of the alphabet
  - There may be many start states
- A DFA accepts when we end in an accepting state.
- An NFA accepts when at least one of the states we end in is accepting.

Another way to think of NFAs

- An NFA is a **non**-deterministic finite automaton.
- Instead of doing one predictable thing at each state given an input, it may have choices of what to do.
- If one of the choices leads to accepting the input, the NFA will make that choice. (It “magically” knows which choice to make.)

Decimal Number NFA... Is Easy!



`-?[0-9]+(\.[0-9]+)?`

Are NFAs More Powerful than DFAs?

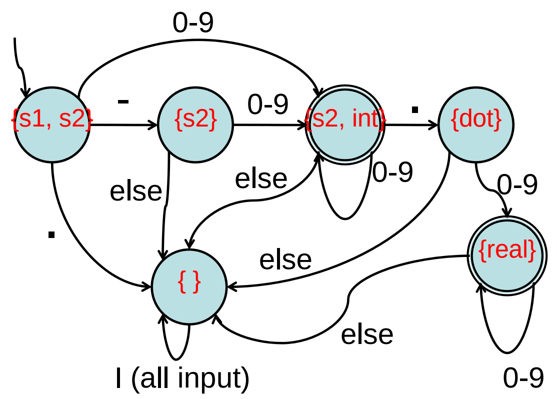
Problem: We can prove that every language an NFA can recognize can be recognized by a DFA as well.

Strategy: Given an NFA, we can show how to build a DFA (i.e., **I**, S, s_0, F, and N) that accepts/rejects the same strings.

Proof : in the appendix

The proof builds that DFA D in the following way: Each state of D corresponds to the set of states of the NFA that can be reached by the same strings.

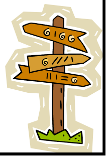
Decimal Number NFA as a DFA



LOTS of states left off that are unreachable from the start state.

Outline

- Formally Specifying DFAs
- Designing DFAs
- Analyzing DFAs: DFAs Can't Count
- Non-Deterministic Finite Automata: No More Powerful than DFAs
- CS Theory and the Halting Problem



Punch Line to one Big Story: Automata Theory

- DFAs are just as powerful as NFAs (although it may take 2^n states in a DFA to simulate an n-state NFA).
- One direction CS theory goes is exploring what different models of computation can do and how they're related: automata theory.
- Computers (at least ones with "an arbitrary amount of memory") are more powerful than DFAs;
  they *can* count.
- But… Are all models limited?

The Halting Problem

- A decision algorithm outputs only "yes" or "no" for an input (like a DFA).
- If we're really smart, we can probably write a program to solve any given decision problem (in a finite amount of time), right?
- How about this one: Given a program and its input, decide whether the program ever finishes (halts). I.e., does it go into an infinite loop?

This was one of the deepest questions in math (and CS!) in 1900, posed by David Hilbert.

The Halting Problem

- OK, let’s say we write the code for this inside the method **`halts?`**:

```
(define (halts? program input)
  ;; Don’t know what goes here,
  ;; but we assume (for contradiction!) it exists.
  )
```

The Halting Problem

- Now, remember Russell’s Paradox and the self-referential “set that contains all sets that do not contain themselves”?
- How about a program kind of like that? It takes a program as input and checks whether the program halts on itself…

```
(define (russ input)
  (if (halts? input input)
    ... ;; do something here
    ... ;; do something else here
    ))
```

But, **what** should our program do?

The Halting Problem

- Now, remember Russell’s Paradox and the self-referential “set that contains all sets that do not contain themselves”?
- How about a program kind of like that? It takes a program as input and checks whether the program halts on itself…

```
(define (russ input)
  (if (halts? input input)
    (russ input)     ; repeat this for ever
    true))
```

It halts only if its input, if run on itself, would ***not*** halt.

The Halting Problem

- Now, run Russ with Russ itself as input. Does Russ halt under those circumstances or not?
- Remember: Russ halts if and only if: its input does not halt when run on itself.

QED, Halt, Full Stop.

Proof (originally) thanks to Kurt Gödel.

APPENTICES

This part is NOT examinable

DFAs Can't Count

- **Theorem:** There is no DFA that accepts $\mathtt{a}^n\mathtt{b}^n$ for any n in N
- The heart of the proof is the insight that a DFA can only have a finite number of states and so can only "count up" a finite number of **a′**s.
- We proceed by contradiction. Assume some DFA recognizes the language $\mathtt{a}^n\mathtt{b}^n$.

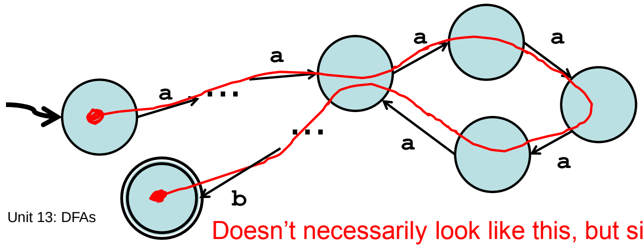
DFAs Can't Count

- Assume DFA **counter** recognizes $\mathtt{a}^n\mathtt{b}^n$.
- **counter** must have some finite number of states $\mathtt{k} = |S_{\mathtt{counter}}|$.
- Consider the input $\mathtt{a}^k\mathtt{b}^k$. **counter** must repeat (at least) one state as it processes the **k a′**s in that string. (Because it starts in $\mathtt{s}_0$ and transitions **k** times; if it didn't repeat a state, that would be **k+1** states, which is greater than $|S_{\mathtt{counter}}|$.)

(In fact, it repeats each state after the first one it repeats up to the **b**.)

DFAs Can't Count

- Consider the input $\mathtt{a}^k\mathtt{b}^k$. **counter** must repeat (at least) one state as it processes the **k a**s in that string.
- **counter** accepts $\mathtt{a}^k\mathtt{b}^k$, meaning it ends in an accepting state.

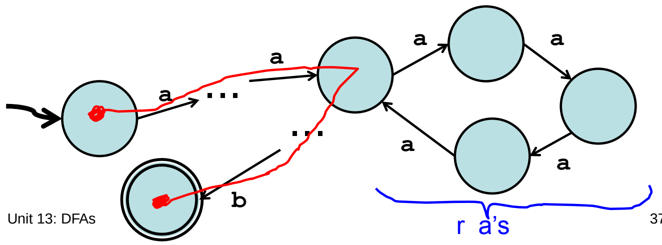


Doesn't necessarily look like this, but similar.

DFAs Can't Count

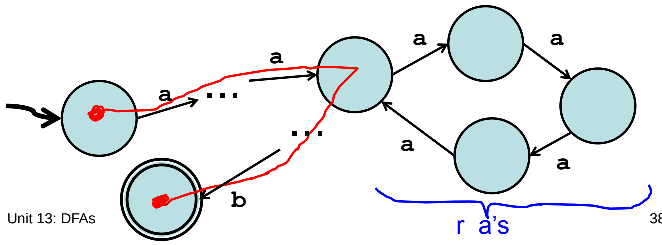
- Now, consider the number of **a**s that are processed between the first and second time visiting the repeated state. Call it **r**.
- Give **counter** $\mathbf{a}^{(k-r)}\mathbf{b}^{k}$ instead.

  (Note: **r > 0**.)



DFAs Can't Count

- Give **counter** $\mathbf{a}^{(k-r)}\mathbf{b}^{k}$ instead.
- **counter must** accept $\mathbf{a}^{(k-r)}\mathbf{b}^{k}$, which is not in the language $\mathbf{a}^{n}\mathbf{b}^{n}$. Contradiction!
- Thus, no DFA recognizes $\mathbf{a}^{n}\mathbf{b}^{n}$



Worked Problem: Are NFAs More Powerful than DFAs?

- **Problem**: Prove that every language an NFA can recognize can be recognized by a DFA as well.
- WLOG, consider an arbitrary NFA "C". We'll build a DFA "D" that does the same thing.
- We now build each element of the five-tuple defining D. We do so in such a way that D faithfully simulates N's operation.

Worked Problem: Are NFAs More Powerful than DFAs?

- C has an alphabet I_C.
- Let $I_D = I_C$.
- They operate on the same strings.

Worked Problem: Are NFAs More Powerful than DFAs?

- C's states are S_C.
- Let $S_D = \mathscr{P}(S_C)$.

"$\mathscr{P}$" is "power set"

- Each state in D represents being in a ***subset*** of the states in C.
  ("Having fingers" on some of C's states.)
- This is *the central idea* of the whole proof.

Worked Problem: Are NFAs More Powerful than DFAs?

- C starts in some subset S_{0C} of S_C.
- Let $s_{0D} = S_{0C}$.
- D's start state represents being in ***the set of*** start states in C.
- This is fine because each state in D is *already* a set of states in C anyway!

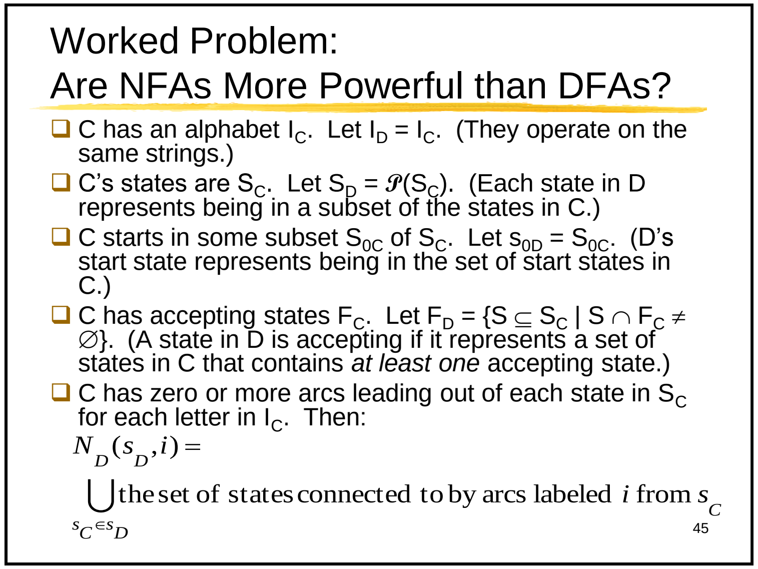
Worked Problem: Are NFAs More Powerful than DFAs?

- C has accepting states F_C.
- Let $F_D = \{S \subseteq S_C \mid S \cap F_C \neq \varnothing\}$.
- A state in D is accepting if it represents a set of states in C that contains *at least one* accepting state.
- So, every subset of C's states such that at least one of its states is an accepting state.

Worked Problem: Are NFAs More Powerful than DFAs?

- C has zero or more arcs leading out of each state in S_C for each letter in I_C.
- D's transition function $N_D(s_D, i)$ operates on a state s_D that actually represents a set of states from S_C.
- So, we just need to figure out all the states that *now* need "fingers on them":
- So $N_D(s_D, i) =$

$$\bigcup_{s_C \in s_D} \text{the set of states connected to by arcs labeled } i \text{ from } s_C$$



Worked Problem: Are NFAs More Powerful than DFAs?

- C has an alphabet I_C. Let $I_D = I_C$. (They operate on the same strings.)
- C's states are S_C. Let $S_D = \mathcal{P}(S_C)$. (Each state in D represents being in a subset of the states in C.)
- C starts in some subset S_{0C} of S_C. Let $s_{0D} = S_{0C}$. (D's start state represents being in the set of start states in C.)
- C has accepting states F_C. Let $F_D = \{S \subseteq S_C \mid S \cap F_C \neq \varnothing\}$. (A state in D is accepting if it represents a set of states in C that contains *at least one* accepting state.)
- C has zero or more arcs leading out of each state in S_C for each letter in I_C. Then:

$$N_D(s_D, i) = \bigcup_{s_C \in s_D} \text{the set of states connected to by arcs labeled } i \text{ from } s_C$$

45