Reminders

• 121.5: Ryan O’Donnell, Analysis of Boolean Functions. Today @ 4:30
• Section 3 cycle begins today
• Extra-length sections:
  • Will (Thursdays: 6-7:30pm), Max & Zuzanna (Tuesdays: 7:30am-9am).
Today

- Infinite vs. Finite functions
- Example: Addition as finite state algorithm
- (Deterministic) Finite Automata:
  - Break 1: Understand DFA
  - Break 2: Design DFA
- Preview of next lecture: Regular Expressions
• Have seen Circuits/NAND-CIRC Programs
• Compute all finite functions:
  • Given \( f: \{0,1\}^n \rightarrow \{0,1\}^m \), exists NAND-CIRC \( C \), s.t. \( \forall x \in \{0,1\}^n, C(x) = f(x) \)
  • Sounds great?
XOR on 2 variables
XOR on 3 variables
XOR on 4 variables
XOR on 5 variables
XOR on 6 variables
XOR on 7 variables
XOR on 8 variables
XOR on 9 variables
XOR on 10 variables
So far

- Have seen Circuits/NAND-CIRC Programs
- Compute all finite functions:
  - Given $f : \{0,1\}^n \rightarrow \{0,1\}^m$, exists NAND-CIRC $C$, s.t. $\forall x \in \{0,1\}^n, C(x) = f(x)$
  - Sounds great?

- Weakness: Only computes finite functions.
  - No generalization?
  - Given circuits for ADD_1, ADD_2, ADD_3, ... ADD_n - do we know what circuit for ADD_{n+1} looks like?
  - Our favorite algorithms generalize!!
Today: Algorithms with finite state

What should an algorithm be?
- Sequence of simple steps
- Different steps for different inputs
- Exactly which step to take must be determined (based on input, easily, locally)
- Different # steps for different input lengths
- When to stop must be determined (based on input, easily, locally)

Finite state algorithms: What step to take, when to stop determined by “finite state” (constant # bits of memory).
Example: Addition as finite state algorithm

- Advantage: O(1)-sized description. Tells how to compute an infinite function $\text{ADD}: \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$
- Can you do anything else?
  - Multiplication? NO 😞
  - ... but can do modular counting, pattern matching
Boolean functions

• From now will focus only on Boolean functions: $G: \{0,1\}^* \rightarrow \{0,1\}$

• Why?
  • Given $F: \{0,1\}^* \rightarrow \{0,1\}^*$, can design $bF: \{0,1\}^* \times \mathbb{N} \rightarrow \{0,1\}$ or $BF: \{0,1\}^* \times \mathbb{N} \rightarrow \{0,1\}$, that are roughly "equally easy/hard".
  • Idea: $BF(x, i) = F(x)_i$
  • If $F(x) \in \{0,1\}^m$ for some $m$:
    • Can go from $F(x)$ to $BF(x, i)$ (for any single $i$) by erasing other parts of output.
    • Can go from $BF(x, i)$ to $F(x)$ by $m$ calls to algorithm for $BF(\cdot, \cdot)$
Exercise Break 1

• Booleanize $Mult: \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$, where $Mult$ is the multiplication function for integers (given in “little-endian”).

• What is domain of your function?
• What is the range?
Deterministic Finite Automata (DFA)

- Finite algorithms computing Boolean functions: $f : \{0,1\}^* \rightarrow \{0,1\}$

- Operation:
  1. Finite number of states: $C$
  2. Starts in state 0, reads $x_0$
  3. At any stage has current state $q$, last read input symbol $\sigma$
  4. Moves to state $T(q, \sigma)$; moves to read next input symbol
  5. If input not done, repeat from Step 3.
  6. When done: Accept (output 1) if current state $q \in S$ and reject (output 0) otherwise.

- Specification: $(T, S)$ where $T : [C] \times \{0,1\} \rightarrow [C]$, $S \subseteq [k]$

- (more elaborate spec. in Sipser): $(Q, q_0, \Sigma, T, S)$ [$Q = [C]$, $q_0 = 0$, $\Sigma = \{0,1\}$]
Example:

\[ f(x) = 1 \Leftrightarrow x \text{ contains 011 as a subsequence} \]
Exercise Break 2:

1) Convert the following diagram to transition function:

2) Describe the function $f$ computed by this DFA.
Regular Expressions

• Motivation: DFA detects simple patterns in strings. Can it do more complex ones?
• Regular expressions:
  • A generalization of “Patterns”.
  • Succinct descriptions of subsets of \(\{0,1\}\)^*

• Definition:
  • Basic cases:
    • 0 is a regular expression
    • 1 is a regular expression
  • Compound cases: If \(r_1, r_2\) are regular expressions, then so are:
    • \(r_1 r_2\): “\(r_1\) followed by \(r_2\)” (or “concatenation”)
    • \((r_1| r_2)\): “\(r_1\) or \(r_2\)”
    • \(r_1^*\): “Concatenation of finite number of \(r_1\)’s”
  • End Cases:
    • \(\emptyset\) (empty set) is regular.
    • “” (null string) is regular.
Regular Expression Matching

- Basic
  - 0 matches 0
  - 1 matches 1
  - "" matches ""
  - No string matches \(\phi\)

- Compound:
  - \(s\) matches \(r_1r_2\) if there exists \(s_1, s_2\) such that \(s = s_1s_2\) and \(s_1\) matches \(r_1\) and \(s_2\) matches \(s_2\)
  - \(s\) matches \((r_1|r_2)\) if \(s\) matches \(r_1\) or \(s\) matches \(s_2\)
  - \(s\) matches \(r_1^*\) if there exists \(s_1, s_2, ..., s_\ell\) such that \(s = s_1s_2 ... s_\ell\) and \(s_i\) matches \(r_1\) for every \(i \in [\ell]\)
Examples:

- (0|1)*011(0|1)*
Examples:

- $(0|1)^*1(0|1)^*1(0|1)^*1(0|1)^*$
Examples:

• \((0 \cdot 10 \cdot 10 \cdot 1)\)
Regular expressions = sets (languages) = functions

- Can think of a regular expression as a set or as a Boolean function:
  - Given regular expression $r$ can look at set (language)
    - $L(r) = \{x \in \{0,1\}^* \mid x \text{ matches } r\}$
    - $f_r: \{0,1\}^* \rightarrow \{0,1\}$ where $f_r(x) = 1 \iff x \in L(r) \iff x \text{ matches } r$
  - We prefer the last version
Next two lectures:

- Understanding DFA via regular expressions:
  - For which regular expressions \( r \) is \( f(r) \) computable by a DFA
    - (Note: # states can depend on \( r \), but not on \( x \) or \(|x|\))
  - What are some functions computable by DFA that are not regular

- Limits of DFA
  - What are some functions that are not computed by DFA?