Overview of Sections 8.1 to 9.3

April 29, 2025

1 Section 8.1: Boolean Algebra

- How do Boolean Algebras generalize propositional logic and set theory?
- What is the definition, and how do we verify that $[B, +, \cdot, ', 0, 1]$ is a Boolean algebra?
- What do we know about **all finite Boolean algebras**?
- Can you prove additional rules, such as De Morgan's laws, universal bound, and idempotence.
- Can you use duality to save yourself work?
- Can you prove Boolean algebra equalities?

2 Section 8.2: Logic Networks

- Equivalent representations of a Logic Network:
  - Truth Functions
  - Boolean Expressions
  - Logic Network
- Can you convert between the three forms?
- Algebraic simplification of the Boolean expressions reduces the hardware needed.
- Can you draw a given Logic Network?
- Do you recall how we created a logic network to carry out binary addition (as Boolean expressions, half-adder, full-adder)?
- Do you remember some of the clever choices of Boolean expressions that reduced computation?

3 Section 8.3: Minimization

We studied two methods of simplification of Boolean expressions:

- Karnough maps – very visual (but really only works for up to four variables).
- Quine-McCluskey – works for as many variables as you have; two stage process (iterative phase, and final table)

The Karnough map illustrates the usefulness of idempotence, allowing us to introduce multiple copies of (essentially reuse) elements for matches (pairs, quads, etc.), which we can then can simplify.

Remember that you may use the techniques of the Karnough map to produce different simplified Boolean expressions: make sure that you've made things as simple as possible, but not simpler! (You can always check your Boolean expressions to make sure that they are equal by using the original "tuples" in the simplified expressions, and you should get 1.)

In Quine-McCluskey, make sure that you've found all the "reduced" expressions possible, comparing every pair that may differ in only a single place.

When you're done, you should check that every pair is essential, with the second type of table.

4 Section 9.3: Finite State Machines

Definition: A **finite-state** machine M is a structure $[S, I, O, f_s, f_o]$ where

Table 1: Elements of a finite-state machine.

S	states of the machine
I	input alphabet (finite set of symbols
O	output alphabet (finite set of symbols
f_s	$f_s : SxI \to S$, the next-state function
f_o	$f_o : S \to O$, the output function

We were able to construct finite-state machines to perform tasks, such as

- binary addition, for example; or the sloppy copy machine.
- set recognition.

Given a machine, can you determine whether a set (a "regular set") is recognized by it or not? Or determine what sets are recognized by it? Remember that machine recognition is defined as follows, with emphasis on the word "ends" below:

Definition: Finite-State Machine Recognition A finite-state machine M with input alphabet I recognizes a subset S of I^* (the set of finite-length strings over the input alphabet I) if M, beginning in state s_0 and processing an input string α, **ends** in a final state (a state with output 1) if and only if $\alpha \in S$.

Regular expressions define sets of input strings which are the ones that finite-state machines can actually recognize. The existence of reasonable sets, which one should reasonably be able to detect (e.g. $S = \{0^n 1^n\}$, where a^n stands for n copies of a), finite-state machines are obviously not sufficient to understand all of computation.

Minimization occurs by partitioning the states into $k-$equivalent subsets.

a. 0-equivalent states have the same output: you can't tell which state you're in by just inspecting the output – since 0-equivalent states produce the same output.

b. Then you ask what happens if we provide any single input to the 0-equivalent states. Would you be able to tell which one you started in? If their outputs to each single input are 0-equivalent, then you can't tell them apart by input s of a single input. Hence the states are 1-equivalent.

c. So in each case you look back "one equivalency": to find out which states are 2-equivalent, you ask which 1-equivalent states have output under a single input that are 1-equivalent (more generally, for $k+1$-equivalency, you check to see if k-equivalent states have k-equivalent next states under a single input).

d. Iterate. When the output of this process repeats (no changes) in the partition of states, you're done – and each subset of states that is still equivalent is equivalent under **any** string of input (so the states can be combined into a single state).