Overview of Sections 6.3 to 9.3

May 5, 2022

1 Section 6.3: Decision Trees

- **decision tree**: a tree in which
  - internal nodes represent actions,
  - arcs represent outcomes of an action, and
  - leaves represent final outcomes.
- Examples
- Lower Bounds on Searching
  - a. Any binary tree of depth d has at most $2^{d+1} - 1$ nodes. (Proof: look at the full binary tree, as it has the most nodes per depth.)
    b. Any binary tree with m nodes has depth $d \geq \lfloor \log m \rfloor$.
  - **Theorem** (on the lower bound for searching):

    Any algorithm that solves the search problem for an n-element list by comparing the target element x to the list items must do at least $\lfloor \log n \rfloor + 1$ comparisons in the worst case.
- Binary Search Tree (Binary Tree Search - follows the same path as an algorithm as the tree creation process!)
- Sorting
  - Theorem on the lower bound for sorting: you have to go to at least a depth of $\lceil \log(n!) \rceil$ in the worst case.

2 Section 7.2: Euler Path and Hamiltonian Circuit

- **Euler Path**: a path in which each arc is used exactly once.
- **Theorem**: in any graph, the number of odd nodes (nodes of odd degree) is even.

- **Theorem**: an Euler path exists in a connected graph $\iff$ there are either two or zero odd nodes.
- Using the EulerPath algorithm (simply counts up elements in a row i of the matrix (the degree of node i), and checks whether that's even or odd; if in the end there are not zero or two odd nodes, there's no Euler path!)
- **Hamiltonian Circuit**: a cycle using every node of the graph. Brute force is often the best method, although we should exploit symmetry whenever possible to reduce the exhaustion.

3 Section 7.3: Shortest Path and Minimal Spanning Tree

- Shortest Path algorithms (for a simple, positively weighted, connected graph)
  - Dijkstra's Algorithm – shortest distance (and path) between x and y
  - Bellman-Ford Algorithm – shortest distance (and path) between x and all other nodes

– Floyd's algorithm – shortest distances (no paths) between $allpairsofnodes$

- Minimal Spanning Trees: A **spanning tree** for a connected graph G is a non-rooted tree containing the nodes of the graph and a subset of the arcs of G. A **minimal** spanning tree is a spanning tree of least weight of a simple, weighted, connected graph G.
  - Prim's algorithm
  - Kruskal's algorithm

4 Section 7.4: Traversal Algorithms

Traversing a graph (generalizes tree traversal):

- depth-first strategy
- breadth-first strategy

Remember: for the test stick with the convention that, given a choice, we should choose nodes in alphabetic order.

5 Section 8.1: Boolean Algebra

- How Boolean Algebras generalize propositional logic and set theory.
- The definition, and verifying that $[B, +, \cdot, ', 0, 1]$ is a Boolean algebra
- Additional rules, such as De Morgan's laws, and idempotence.
- Duality
- Proving Boolean algebra equalities

6 Section 8.2: Logic Networks

- Equivalent representations of a Logic Network:
  - Truth Functions
  - Boolean Expressions
  - Logic Network
- Converting between the three forms
- Algebraic simplification of the Boolean expressions reduces the hardware needed
- Adding binary numbers (as Boolean expressions, half-adder, full-adder); clever choice of Boolean expressions reduces computation.

7 Section 8.3: Minimization

Two methods of simplification of Boolean expressions:

- Karnough maps – very visual (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 phase.

8 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	$f_s : SxI \rightarrow S$, the next-state function
f_o	$f_o : S \rightarrow O$, the output function

Construction of finite-state machines to perform tasks

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

Given a machine, determine whether a set (a "regular set") is recognized by it or not. Or determine what set is 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).