Logic Synthesis & Optimization
Lectures 2, 3
Review of Graph Theory & Algorithms

Instructor: Priyank Kalla
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT 84112
Email: kalla@ece.utah.edu

In this lecture, we are going to review, **very briefly**, concepts related to set theory and graph theory. Sets are taught in high-school, so you should know the basics. But not everyone has a working knowledge of Graph Theory. For this reason, I will introduce graphs, and illustrate with examples how we can apply search & decision techniques to solve fundamental CAD problems. Subsequently, we will cover algorithms and their complexity issues. Some of this stuff would appear to be very elementary but most of these CAD techniques are quite useful in Logic Design, Test and Verification.

I. Set Theory

A set is a collection of elements; e.g. $S = \{a, b, c\}$ is a set S of three elements a, b, and c. The cardinality of the set S, denoted by $|S|$, is the number of elements it contains. The *cover* of S is a set of subsets of S whose union is S. A *partition* is a cover by disjoint subsets. Partition is an often abused term - sometimes a non-disjoint cover is also considered a partition, but puritanical mathematicians would disagree with that. Set membership is denoted by $\in$, inclusion by $\subset$, union by $\cup$ and intersection by $\cap$, complement of X by $\overline{X}$.

You must be aware of Venn Diagrams, so that needs no mention. But in case you have forgotten that you ever went to high-school, here are two high-school formulas that you might find useful in switching theory:

- $|A \cup B| = |A| + |B| - |A \cap B|$.
- $A - B = A \cap \overline{B}$. Remember set difference? A simple formula, but it has applications in the VLSI domain.

A *Cartesian product* of sets X, Y, denoted by $X \times Y$, is the set of all ordered pairs (x, y) such that $x \in X$ and $y \in Y$. Cartesian product is yet another high-school result that finds a lot of application in VLSI CAD. If we represent the state-space of sequential circuits (finite state machines) using sets, then the cross product of these sets would create something called a "product machine." State space analysis of the product machine is a useful tool in equivalence verification.

A. Relations and Functions

Given two sets A, B, a *relation* R between them is a subset of $A \times B$: written $aRb, a \in A, b \in B, (a, b) \in R$. Reflexive relation: $(a, a) \in R$. Symmetric: $(a, b) \in R \Rightarrow (b, a) \in R$. Transitive: $(a, b) \in R, (b, c) \in R \Rightarrow$ (__, __) $\in R$. (You fill in the blanks - this should be easy.) An **equivalence relation** (a very important concept) is one that is reflexive, symmetric and transitive.

A *function* f is a mapping between two sets A, B, $f : A \rightarrow B$, that associates exactly one element of B to each element of A. Unlike relations, one element of A cannot have two associated elements in B. What is the *domain* of f and what is its *co-domain*? What is the difference between *co-domain* and *range*? If $y = f(x), x \in A, y \in B$, then is $f(x)$ the co-domain or the range?

B. Image of a Function

If $y = f(x) : A \rightarrow B$, we say that y is the **image** of x under f. Given a function $f : A \rightarrow B$, and a domain subset $C \subseteq A$, the **image** of C under f is defined by:

$$IMAGE(f, C) = \{f(x) | \forall x \in C\} \tag{1}$$

If $C = A$, the image of C under f is also called the range of f. Now do you get the difference between co-domain and range?

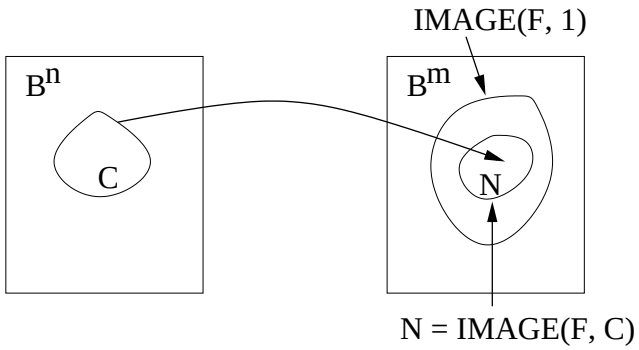


Fig. 1. Image computation.

Symmetrically, we can also define the **Pre-image**, also called the inverse image of a function:

$$PRE(f, C) = \{x \in A : \exists y \in C, y = f(x)\} \tag{2}$$

In pure and simple English: Take a point x in the domain A. According to the function f, find its mapping y in the co-domain B. y is the image of x under f. Vice-versa for inverse image. Notice, however, that image is not restricted to just one point - but can consist of a set of points. For example, let $C \subseteq A$. For EVERY point in domain subset C, you can find a mapping in the co-domain B. The set of all these (mapped) co-domain points constitute the image set! We'll use image and preimage extensively for computing *don't care* sets in multi-level circuits.

II. Graph Theory

Recall plotting graphs on (x, y) axes? Remember plotting some points on the graph paper, and connecting those points with a line and generating a curve? As it turns out, the plots that you used to draw (again, in your high-school classes) have some cool properties. In VLSI-CAD, we require a good understanding of those properties. So let us formally define graphs and look at some of their properties that we will use throughout this course.

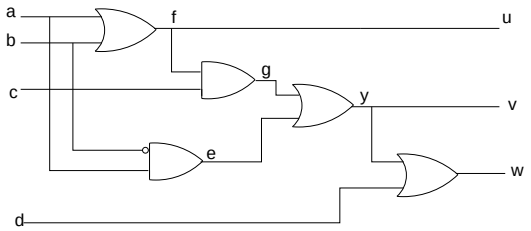
A *graph* $G(V, E)$ is a pair (V, E) of a vertex set V and an edge set E. (The points that you plotted on the graph paper are analogous to the vertices. The lines that you used to draw to connect these points, or vertices, are analogous to the edges). E is in fact a *binary relation* on V: we say $e_{ij} = (v_i, v_j) \in R; v_i, v_j \in V; e_{ij} \in E$. If the set V is bounded, i.e. it is finite, the graph is considered a finite graph. When an edge e has a vertex v as an end point, e is *incident* on v. The edges can be directed or undirected. And the *degree* of a vertex is the number of edges incident upon it. For a few examples of graphs, see lecture slides.

A *walk* is an alternating sequence of vertices and edges; when the edges encountered in a walk are distinct (never re-visited) the walk is called a *trail*. Similarly, if the vertices in a walk are distinct, it is called a *path*. A closed walk is a *cycle*. A graph is *connected* if there is a path from any vertex to every other vertex. Remember, in a path, you cannot repeat a vertex. If a closed walk in a graph contains all the edges then the graph is called an *Euler graph*. See slides for an example of a connected graph:

If there are no cycles in a graph, the graph is *acyclic*. A *connected acyclic* graph is a tree. What is a binary tree? Often, in VLSI-CAD we are going to deal with graphs with directed edges: called direct graphs or digraphs for short. The concepts of walks, paths, connectivity, etc. follow straightforwardly to digraphs. If we take a combinational circuit, represent the gates with nodes (vertices) and the connections between them by directed edges, we get a directed acyclic graph. What if there are directed cycles in a circuit?

Now that we know what is connectivity in terms of walks, trails, paths, etc., let us look at where would you use these concepts. Consider the circuit shown below. Represent the circuit as a directed graph: represent the inputs, outputs and gates with nodes and the wires with edges. Let us call each input vertex as the source and the output as the sink. Starting from source a, can you find the *shortest path* to any sink? The longest path to any sink? Can you find the shortest and longest paths from any and all sources to any and all sinks? Given a tree and starting from the root,

you have to visit all the nodes: you can do it in a depth-first manner or in a breadth-first manner - hence a depth-first search (DFS) and a breadth-first search (BFS). These are really the bread-and-butter algorithms in VLSI-CAD.



III. Algorithms and Complexity

An algorithm is a computational procedure that takes a set of inputs, performs a *finite* number of (unambiguously defined) operations and delivers a set of outputs [**?**]. Usually, the number of computational steps involved in an algorithm is defined by the number of inputs to it. This defines the computational effort of the algorithm, and it is usually measured in terms of *time* and *memory*. The behaviour of the algorithm also depends upon the data and its characteristics. That is why the *worst-case* and *average-case* behaviour of an algorithm is a good indication of the amount of effort required to solve the problem.

Consider the following problem. Given a list of 5 numbers, we want to find the largest of them. The fundamental computation involved in this search is a "comparison". What is the maximum number of comparisons required? Answer: 4. If the data size is increased to 10, the number of computations = 9. If the data size is n, we require *no more than n-1 comparisons*. That means, an upper bound on the comparisons needed is *of the order of n-1*, written as $O(n-1)$. Usually, constants are ignored in this notation, and that is because as n increases to a very large number, the constant becomes meaningless. So to compute an *upper bound* on the computational steps (or the computational complexity) of the algorithm, we can always say that: upper bound = $O(n-1) \leq O(n)$. If we have to *sort* this array, then we can do it in no more than $O(n^2)$ time.

All the algorithms that have a computational complexity of polynomial nature, i.e. $O(n), O(nlogn), O(n^2), O(n^3)$, etc., are generally considered *tractable*. Unfortunately, in Boolean logic, most problems do not have polynomial time or space complexity. For example, let there be a Boolean function of n variables. The size of the truth table for that function is going to be 2^n (exponential space complexity). If we are to search for some element in the truth table that has some property, in the worst-case we would have to go through the entire table; time complexity is $O(2^n)$ or exponential in nature. Exponential complexity generally means that we are in trouble!

Suppose I have a circuit with n inputs; i.e. my design space has possibly 2^n points. Let me randomly assign some arbitrary cost c_i to each point. I want to now pick a point in the design space that has the least (or highest) cost. In the worst-case I would have to visit each and every point and look-up its cost - exponential complexity! Now suppose that I have a "magical computer that can make correct guesses all the time". This hypothetical computer does not perform any deterministic computation: it makes guesses. Such a non-deterministic computational procedure (a euphemism for a guess) may solve the problem in polynomial time. Hence, my problem is NP, or non-deterministically polynomial-time solvable. But a deterministic machine cannot make correct guesses all the time. That means, for the above problem the data can always be arranged in such a way that a deterministic computation will end up searching the design space exhaustively, or 2^n times. Problems that exhibit this kind of behaviour are called *intractable*. And the bad news is, almost all the problems in VLSI-CAD, at least the important ones, are all (drum-roll please) intractable!

IV. Decision and Optimization Problems

A decision problem has a binary solution: true/False or Yes/No. Most problems in Verification and Circuit testing are decision problems. For example: Given a circuit with two outputs, does there exist a set of inputs that will excite the first output to **1** and the second output to **0**? This is a decision problem. Another example: Suppose a gate in my circuit is faulty, can I find a set of inputs that will excite the faulty behaviour in that gate?

An optimization problem is a problem whose solution can be measured in terms of a cost, which has to be minimized or maximized. Moreover, the solution has to *satisfy* a set of constraints; this implies that it can be solved by employing sequences of decision problems and are thus as hard as (if not more) the constituent decision problems. Two-level

logic minimization using Karnaugh-maps is an optimization problem: What is the cost to be minimized and what are the constraints?

In the class we are going to analyze solutions to various optimization problems involving:

- graph traversal
- Depth-first search and breadth-first search.
- Greedy search
- Branch and Bound search
- Any other property.....