# EE677 : **Foundation of VLSI CAD**
# Comments on project problems

Dilawar Singh

Sep 22, 2012

**Abstract**

This document partly describes many of the problems listed on course wiki. This is not a technical document which explains problems in details. Algorithms used to tackle these problems can be found in two books : **Introduction to algorithms**, Cormen, Leiserson et. al. which deals with algorithms in general; and **Graph Theory** with application to Engineering and Computer Science by Narsing Deo, which explains graph algorithms lucidly.

Algorithms related with placement and routing, circuit partitioning, layout generation in the area of VLSI CAD can be found in **VLSI Physical Automation** by Sadiq M Sait et. al..

It is hard to say how much effort is precisely needed to solve problems listed in this document. It depends as much on your fluency with programming language as it depends on theoretical understanding. However, I have mentioned an integer in front of some problems which roughly indicates the effort needed to solve them. Needless to say, this number can be well wide off the mark.

# 1 Graphs in Boolean world

As an old Chinese philosopher never said, "Words about graphs are worth a thousand pictures."
– **William Saffire**, *The English Language*
The New York Times, Sep 11, 2009

**Lobbying for graphs** Data-structures such as array, list, vector, map, dictionary etc. are very useful. However, graphs has its own place among the titans. Many algorithms become easier to visualize when we make use of graphs on paper; proofs are easy to state and understand.

Moreover, from the point of view of a programmer, coding with graph is convenient since sophisticated graph libraries are available in almost all self-respecting languages. Of course, there is a price to pay for this convenience : graphs consume relatively more memory to construct. Where available memory size is limited, such as on embedded systems, one may not be able to use graphs. One has to rewrite his algorithm which works on less costly data-structures.

In 1975, there was only one book available on graph theory. It was written by Koing in German (`Theorie der Endlichen und Unendlichen Graphen`). (Later, William Tutte translated this book into English.) In its early years, graph theory was often looked upon with condescend by some mathematician. "The slum of topology" was the derisive remark they used to pass.

Since then graphs has emerged out of "the slum of topology" to become one of the most extensively studied mathematical structure.

Narsingh Deo book "**Graph theory** with applications to engineering and computer science" is a very good introductory book on graph-theory for engineers. H. Narayan's "Submodular function and Electrical Network", especially chapter 3, gives an excellent overview of graphs in networks.

**Libraries** There are many graph libraries available. No language which claims to be a contender for algorithmic design can exists without a graph library. We recommend using **Boost libraries** or **LEDA graph libraries** for C++. Boost graph libraries bindings are available for python, package **graph-tool**. Python's native graph library **networkx** can also be used for small graph applications. Networkx has great many routines but it is not recommended for very large graphs for speed of networkx is not very good compared to graph-tool.

**String graph to files** Format **dot** and **graphml** are two modern graph-formats. We'll use graphml to encode/decode our graph to a file. Both boost and LEDA libraries can read and write graphs to graphml format. This is a text-based format and can be read by humans.

## 1.1 Graph as Boolean expression trees

In many of these project problems, we'll be working with Boolean expressions as input to the program or as intermediate data. There are many ways to feed a Boolean expressions to a computer program. We need to fix a standard such that data can be transferred from one program to another without any hassle.

Boolean expressions can be encoded as expression-trees. What are expression trees? Mimicking Dodo the bird of 'Alice in Wonderland', one is temped to conclude that the best way to explain it is to show it.

On paper, Boolean expressions are written using standard symbols. Table 1 summarises them.

Consider the following expression. We are using standard Boolean symbols in these expressions.

$$y = (x_1 \vee x_2') \wedge (x_2 \vee x_3) \wedge (x_1' \vee x_3') \tag{1}$$

We'll use `not`, `and`, `or` to describe gates while writing these expressions. In special cases, such as technology mapping, one can also use additional `xor`, `nor`, and `nand` in output graph [1]. Figure 2 encodes these expression into an expression tree.

Three problems are immediate.

[1]A detailed standard should be made available on wiki

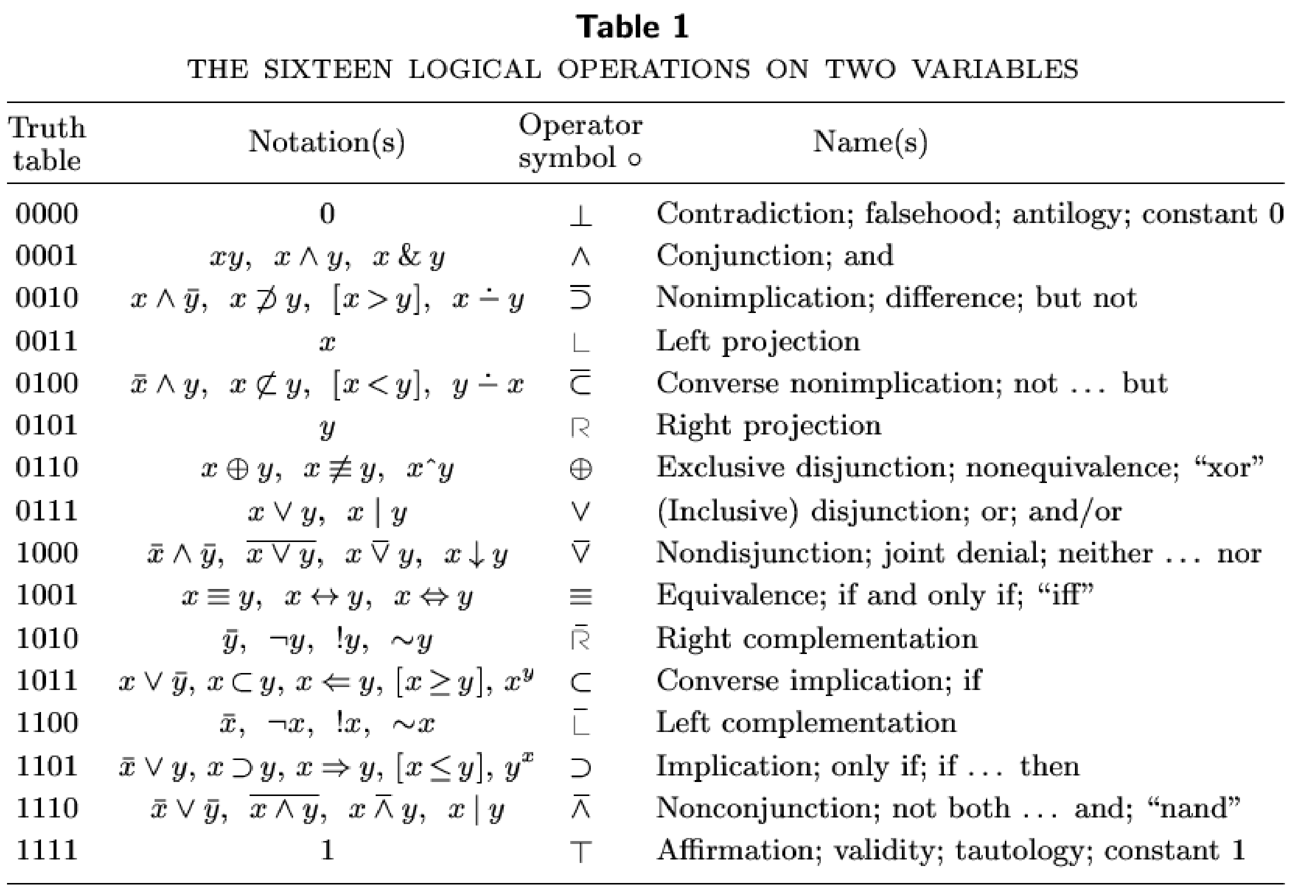


**Table 1**

THE SIXTEEN LOGICAL OPERATIONS ON TWO VARIABLES

| Truth table | Notation(s) | Operator symbol $\circ$ | Name(s) |
|---|---|---|---|
| 0000 | $0$ | $\perp$ | Contradiction; falsehood; antilogy; constant 0 |
| 0001 | $xy,\ x \wedge y,\ x \,\&\, y$ | $\wedge$ | Conjunction; and |
| 0010 | $x \wedge \bar{y},\ x \not\supset y,\ [x > y],\ x \dot{-} y$ | $\bar{\supset}$ | Nonimplication; difference; but not |
| 0011 | $x$ | $\mathsf{L}$ | Left projection |
| 0100 | $\bar{x} \wedge y,\ x \not\subset y,\ [x < y],\ y \dot{-} x$ | $\bar{\subset}$ | Converse nonimplication; not … but |
| 0101 | $y$ | $\mathsf{R}$ | Right projection |
| 0110 | $x \oplus y,\ x \not\equiv y,\ x\hat{\ }y$ | $\oplus$ | Exclusive disjunction; nonequivalence; “xor” |
| 0111 | $x \vee y,\ x \mid y$ | $\vee$ | (Inclusive) disjunction; or; and/or |
| 1000 | $\bar{x} \wedge \bar{y},\ \overline{x \vee y},\ x \mathbin{\bar{\vee}} y,\ x \downarrow y$ | $\bar{\vee}$ | Nondisjunction; joint denial; neither … nor |
| 1001 | $x \equiv y,\ x \leftrightarrow y,\ x \Leftrightarrow y$ | $\equiv$ | Equivalence; if and only if; “iff” |
| 1010 | $\bar{y},\ \neg y,\ !y,\ \sim\! y$ | $\bar{\mathsf{R}}$ | Right complementation |
| 1011 | $x \vee \bar{y},\ x \subset y,\ x \Leftarrow y,\ [x \geq y],\ x^y$ | $\subset$ | Converse implication; if |
| 1100 | $\bar{x},\ \neg x,\ !x,\ \sim\! x$ | $\bar{\mathsf{L}}$ | Left complementation |
| 1101 | $\bar{x} \vee y,\ x \supset y,\ x \Rightarrow y,\ [x \leq y],\ y^x$ | $\supset$ | Implication; only if; if … then |
| 1110 | $\bar{x} \vee \bar{y},\ \overline{x \wedge y},\ x \mathbin{\bar{\wedge}} y,\ x \mid y$ | $\bar{\wedge}$ | Nonconjunction; not both … and; “nand” |
| 1111 | $1$ | $\top$ | Affirmation; validity; tautology; constant 1 |

Figure 1: From Knuth, TAOP, 4A

**Problem 1** (15)**.** *Given a Boolean expression tree, construct another graph where each node represents a minterm of function. And an edge is drawn between two nodes whenever their Hamming distance is 1.*

Such a graph could be an input to Quine McClusky algorithm which works over graphs (as we have discussed during project-session).

**Problem 2** (25)**.** *Given a Boolean expression tree, minimize the Boolean function represented. 1.*

**Problem 3** (45)**.** *Given two Boolean expression trees, check if both expressions trees represents same Boolean function or not.*

**Remark 1** (HDL to Boolean expression-trees)**.** *Is there any way to construct Boolean expression tree out of a verilog design? Well, there seems to be one and TA of this course should give you a program to do this.*

*Computer program **vis** can convert a verilog design to a **bliff** file. All we have to do is to convert this bliff file into boolean expression tree.*

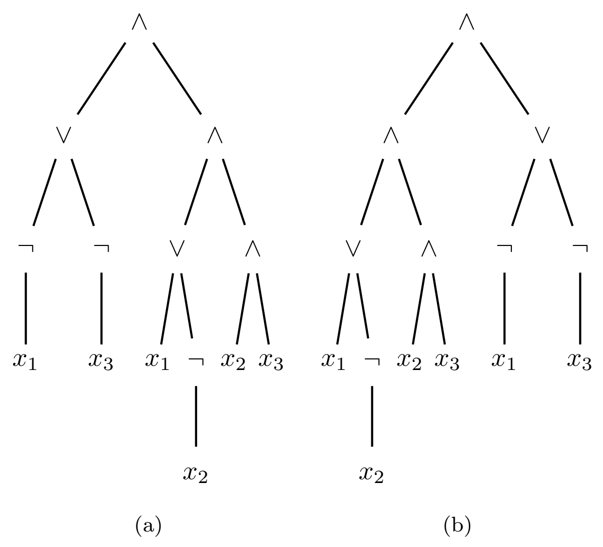


Figure 2: Two possible expression-trees of Boolean expression described in equation 1. Some other representations are also possible. Since we have proved associativity in Boolean algebra, both of these trees are functionally equivalent. Although it is not at all easy to show that two given expression trees are functionally equivalent or not.

## 1.2 Graphs as Binary Decision Diagrams

In popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram (ROBDD in the literature, used when the ordering and reduction aspects need to be emphasized). The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order. This property makes it useful in functional equivalence checking and other operations like functional technology mapping.
– **Wikipedia**, The free encyclopedia (Accessed Sep 22, 2012)

One more way to encode a Boolean expression is BDD (Binary Decision Diagrams) which can been seen as recursive composition of **If-then-else** logic.

**If-Then-Else** A Boolean function $f(x_1, \ldots, x_n)$ can be decomposed as following :

$$\begin{aligned} f(x_1,\ldots,x_n) &= x_1' \vee g(x_2,\ldots,x_n) \wedge x_1 \vee h(x_2,\ldots,x_n), \text{where} \\ g(x_2,\ldots,x_n) &= f(0,x_2,\ldots,x_n) \\ h(x_2,\ldots,x_n) &= f(1,x_2,\ldots,x_n) \end{aligned} \tag{2}$$

This can be read as (ignoring function argument) : **if** $x_1 = 0$ **then** $f = g$ **else** $f = h$. For electrical engineer, it is well known multiplexer.

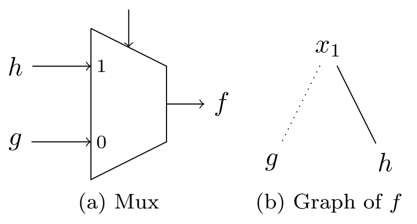
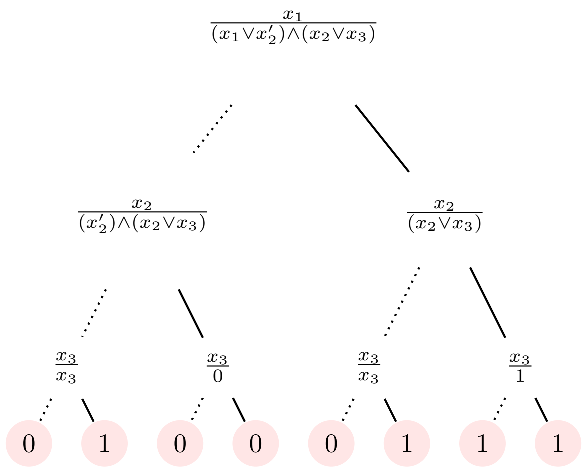


Figure 3: **If-Then-Else** operator. In (a) we have corrosponding circuit, which is a mux. In (b), we have graphical representation of $f$. Dotted line indicates that on this branch $x_1 = 0$.

Variable $x_1$ does not appear in $g$ and $h$. One application of **if-then-else** breaks a function on $n$ variables into two functions of $n-1$ variables. We can apply **if-then-else** operator recursively till we are left with constant function 0 and 1. Lets consider the following expression.

$$y = (x_1 \vee x_2') \wedge (x_2 \vee x_3) \tag{3}$$

Now, we can construct a diagram which shows the process of recursively applying the **if-then-else**. This process is illustrated below.



$\frac{x_1}{(x_1 \vee x_2') \wedge (x_2 \vee x_3)}$
$\frac{x_2}{(x_2') \wedge (x_2 \vee x_3)}$
$\frac{x_2}{(x_2 \vee x_3)}$
$\frac{x_3}{x_3}$
$\frac{x_3}{0}$
$\frac{x_3}{x_3}$
$\frac{x_3}{1}$
0
1
0
0
0
1
1
1

Figure 4: A graph diagram represents function described in equation 3. A network of mux can be built form this graph. This kind of graph are called Binary Decision Diagrams. Usually reduced expression is not written below the variable name as we have written in this diagram.

A cursory look will reveal that this is very inefficient. We don't need so many nodes for 0 and 1. One node for each of them will suffice which can be shared by multiple edges by other nodes. Other simplifications are also possible which are discussed at length in your textbook. Verify that following diagram also encode the same function but with fewer nodes.

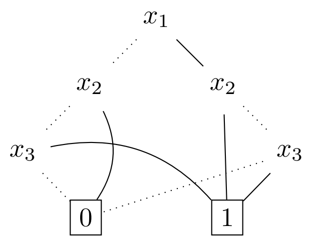


Figure 5: A **Reduced-Order BDD** of function described in equation 3. Note that we have chosen $x_1$ first, followed by $x_2$ and then $x_3$. Any other ordering will also produce a ROBDD. What is important that for a given ordering, generated ROBDD is unique. Size of ROBDD is know to be very sensitive to the ordering or variables.

**BDD packages** Colarado university package CUDD maintained by Fabio Somenzi is widely used. Donald E. Knuth has also written a literate program for BDD which can be downloaded from his homepage.

**Further reading** BDD's are discussed extensively in your textbook. You can also read Byrant paper. An elementry implemention of BDD was submitted last year. You can ask your TA for that code.

**BDD virtues** For a given ordering, ROBDD produced is unique. This is why they are theoretically attractive for representing Boolean expressions. Moreover, since two BDDs can be merged together efficiently on computer, they are very useful in solving many combinatorial problems.

- As we have seen that any BDD can be realised as a network of mux which can be implemented as 4-LUT on FPGA.
- The problem of satifiability of a Boolean expression is reduced to finding a path from root node to node $\boxed{1}$. Not only we can prove whether a function is satisfiable or not, we can enumerate all combination of variable which satisfies the function.
- Add some more here.

**Problem 4** (20)**.** *Draw BDDs which are not satifiable i.e. there is no path between root node and node* $\boxed{1}$*. If we can draw such BDDs using a program, we can definately convert it back to a Boolean expression which is not satisfiable.*

**Problem 5** (40)**.** *Given a expression tree, convert it to ROBDD for various variable ordering. Also generate all minterms from ROBDD. Write down the variable ordering and reduced Boolean expression. Convince yourself that for different ordering, differnt expressions are generated? Can you show that for the optimal ordering of the variables, size of Boolean expression generated from ROBDD is not worse that that of generated by Quine McClusky?*

**Comment 1.** *The classic problem of finding a minimum size Disjuntive Normal Form of a Boolean function can also be solved using ROBDD. Oliver Coudert has given an excellent overview in Integration (1994), pp 97-140.*

# 2 Boolean Logic

'Contrariwise', continued Tweedledee, 'if it is so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.
– **Lewis Carroll**, Through the Looking Glass (1871)

**A note from history** Modern Boolean expressions are written differently from what were originally devised by George Boole. He thought that 0 and 1 could be used to construct a calculus of logical reasoning; in his *An Investigation of Laws of Thoughts* where he wrote,

> Let us conceive, then, of an Algebra in which the symbols x, y, z, &c admit indifferently of the values of 0 and 1, and of these values alone.

. It is surprising that he never dealt with the "logical or" operation +; and he restricted himself to ordinary arithmetic on 0 and 1. In his system, $1 + 1$ was 2 and not 0. He took pain to deal with the case when both of terms in $x + y$ were 1; he wrote $x + (1 - x)y$ to ensure that result of this disjunction would never be equal to 2. He also called his + operator both "or" and "and" in his paper. These days this practice will be strange if not unacceptable. But when we realize that his usage was in fact normal English, it looks reasonable. As Knuth explains in [The art of computer programming, 4(A)], we say, for example, that "boys **and** girls are children," but "children are boys **or** girls".

## 2.1 Boolean Satisfiability problem

A Boolean [2] function is said to be **satisfiable** if it is not identically zero - i.e. it has at least one implicant. To find an *efficient* way to decide whether a given Boolean function is satisfiable or not is the most famous problem in computer science. To be precise, we ask : *Given a Boolean formula of length N and test it for satisfiability, always giving the correct answer after performing at most* $N^{O(1)}$ *steps?* [Knuth, TAOP 4(A)].

Granted that testing for satifiability in general is tough. In fact, it is difficult even in simplified form; a conjuctive normal form that has only three *literals* in each clause :

$$f(x_1, \ldots, x_n) = (t_1 + u_1 + v_1).(t_2 + u_2 + v_2) \ldots (t_m + u_m + v_m) \tag{4}$$

Here each $t_j, u_j$, and $v_j$ is $x_k$ or $x'_k$ for some $k$. This form is called **3CNF** and problem of deciding satisfiability of these formulas is called **3SAT**. We'll restrict our short attention span to 3SAT problems for any general SAT problem can be reduced to a 3SAT problem.

**Example** The shortest unsatisfiable Boolean formula in 3CNF known to humanity is $(x + x + x).(x' + x' + x')$. This is lame for it is of no challenge to the intellect of an initiated.

**Remark 2.** *Can you think of a program which generates Boolean functions in 3CNF forms which are not satisfiable?*

When we assume that three literals in all clauses in equation 4 are differnt then we run into troubled water. Each clause will rule out exactly 1/8 possibilities (why?). Thus any such formula with 7 clause is automatically satifiabile!

The shortest interesting formula in 3CNF has 8 clause and is due to R. L. Rivest which is following unless I made some mistake in copying it down.

$$\begin{gathered}(x_2 + x_3 + x'_4).(x_1 + x_3 + x_4).(x'_1 + x_2 + x_4).(x'_1 + x'_2 + x_3).\\(x'_2 + x'_3 + x_4).(x'_1 + x'_3 + x'_4).(x_1 + x'_2 + x'_4).(x_1 + x_2 + x'_3)\end{gathered} \tag{5}$$

**Algorithm and data-structures** Boolean function will be given as expression-trees stored in `graphml` format. Binary decision diagrams (BDD) can also be emplyoed in solving this problem 1. I am not aware of any algorithm which generates satisfiable 3CNF functions efficiently. It is very hard to give a list of function which are unsatisfiable. For example, look at this discussion.

One of the most widely used algorithm to solve this problem is Davis Putnam Logemann Loveland algorithm [DPLL].

[2] In old times, Boolian

**Problem 6** (30)**.** *Implement DPLL algorithm.*

**Remark 3.** *There are many SAT solvers. There is also an annual competetion where SAT solvers compete for efficiency. You can find many example problems. See this webpage.*

*The standard format to these sat solver is DIMACS format. You can also use the same format.*

One might as well do the following in her spare time.

**Problem 7** (20)**.** *Convert Boolean expression tree into DIMACS file.*

**Comment 2.** *Make sense of Tweedledee comment quoted at the begining of this section. Lewis Carroll was a logician, so it is likely that this comment is logically consistent.*

*One solution is due to C. Sartena. Tweedledee is describing the implication $x \implies y$. What he actually wants to say is "Contrariwise, if y is so, x might be, and if x were so, y would be; but as y isn't; x ain't. Thats logic!".*

## 2.2 Graph algorithms

In addition to the algorithms discussed in the classroom, following algorithms can also be taken up for the project.

**Graphs and matrix** Recall from network theory that a electrical network can be represented by a graph. Graph was usually accompanied by its incidence matrix.

**Problem 8** (15)**.** *Given a graph, compute its adjacency matrix, Laplacian matrix, and distance matrix. Also compute its eigenvalues.*

**Remark 4.** *R. B. Bapat book* ***Graphs and Matrices*** *is a great introductory book. Students who are interested in graph theory may like to buy it for future reference.*

**Graphs and circuits** Also recall that we constructed admittance matrix $A$ and then we solved $Ax = b$ to solve the circuit.

**Problem 9** (30)**.** *Given a ngspice netlist (R, V, I only) construct a graph which represents this circuit where each edge represents a device (R, V, I). Generate appropriate matrices to solve this circuit using modified nodal-analysis (MNA).*

Ngspice becomes very slow with nodes over 20-30 thousands. It was designed to use sparsed matrix techniques which does not scale over large circuits nicely. Another method $NAL - NBK$ is discussed in Prof. H. Narayana book **Submodular function and electrical network**.

**Problem 10** (35)**.** *Given a netlist (R,V,I), generate a network graph and solve it using $NAL - NBK$ method. You can also explore the possibility of extending BITSIM with Prof. Patkar.*

**Partitioning graphs** Partitioning of graph is very important. Prof. Patkar Ph.D. thesis was in the area of partitioning. You can request him to deliver an overview of partitioning techniques in a separate lecture. Partitioning of graph can be done for various purposes. As far as computation is concerned, one major philosophy behind partitioning is **divide-and-conquer** such that each partitioned problem becomes 'independent' enough to be solved on different processor. For an example see chapter on 'multi-port decomposition" in Prof. Narayanan book.

You are encouraged to partition a graph as you like (with some justification). Two of the following partition criteria, however, are too important to miss.

**Problem 11** (15)**.** *Given a graph, partition it into its strongly connected components, 1-connected components, and 2-connected components.*

**Problem 12** (25)**.** *Given a simple connected graph, partition all vertices of it into smallest possible number of disjoint, independent sets. An independent set is a set of vertices such that no two vertices in the set are adjacent.*

**Some more problems are listed in section 3.**

**Matching TAs with courses** Here is one practical problem. Each year, EE department asks its Teaching Assistants to give three preferences of courses which they would like to serve as Teaching Assistant.

- There are $n$ TAs in the department and there are $m$ courses running.
- Each TA has a credit rating between 0 and 5.
- Each course has certain requirement of TA credits e.g. EE677 needs TAs worth 20 credits.
- Each TA belongs to any of the following 3 category : DD, MTECH, RS.
- Each category of TAs has its average rating.

Construct a bipartite graph with TAs on left and on the right hand side we have courses. Draw and edge between a TA and a course if and only if, he has given that course as his preference. Assign a weight on the edge according to the preference for example if 10407007 picks EE677 as his first choice, EE309 as second and EE721 as his third choice then there are three edges going from 10407007 to EE677, EE309, EE721 with edge weight-age of 10, 7, 3 (some other numbering is also possible).

**Problem 13** (45)**.** *Allot each TA a course such that sum of edges in matching is maximised. Make sure each course gets TA's such that distribution of credit set is 'fair'. For example if EE677 which requires TAs worth 20 credit can be alloted 4 TA's worth 5 credit each (highest possible) which is a bad distribution of 10 TA's each worth 2 credit, this is equally bad. We want as many good TA's as bad one in the course.*

*Explore the possibility of adding another constraint : there is at least one TA from two different category?*

*Assign credit randomly to TA's to test your algorithm.*

*What if a TA wants a course eagerly but instructor has given a negative feedback about him in the past? Or he does not want him at all? Find a strategy which causes least heart-burn.*

**Network flow** Network flow algorithms does not look very glamorous at first reading but they are one of the most effective tool in taming combinatorial problems. For instance, bipartite graph matching, integer programming, as well as linear programming problems can be solved using network flow methods.

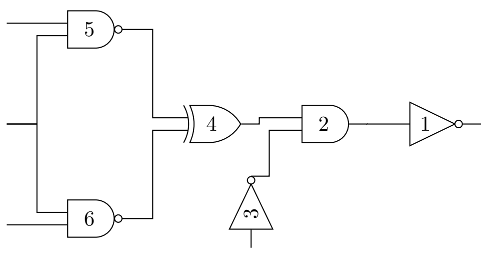
**Problem 14** (35)**.** *Implement **push-relabel** method to compute max-flow in a network. Use priority queues for iteration over vertices.*

# 3 VLSI Design Automation

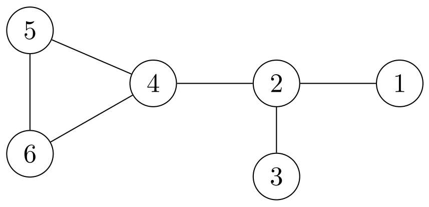
These problems can be classified into following subsections. We also hint at possible approach to their solutions.

## 3.1 Circuit partitioning

Given a circuit, we model it as a graph. Each node represents a circuit element and edge represent the interconnects. For example consider the following figure.



(a) A circuit



(b) Modeled as graph

Now one can state a generic partitioning problem.

**Problem 15** (40)**.** *Given a graph $G(V, E)$ where each vertex $v \in V$ has a size $s(v)$, and each edge $e \in E$ has a weight $w(e)$, the problem is to divide the set $V$ into $k$ subsets $V_1, \ldots, V_k$ such that an objective function is optimized, subject to certain constraints.*

One can formulate various constraints. Better known among them the listed below.

**Bounded size** Assume that a single chip does not allow more than 6 gates on average and your design has 100 of gates. Thus, you need to partition your circuit into at least 17 subsets, each of them is not having more than 6 gates. Wouldn't be great if each partition have equal number of gates?

**Minimize external wiring** Another situation is that we have many packages but there is a need to connect these packages through external wires. It is hightly desirable to minimize the lenght of external wiring (why?).

One of the most popular algorithm to partitioning problem is due to **Keninghan and Lin** which finds a partition and improves it further. A heuristic is due known as **Fiduccia Mattheyses**. Some of the most powerful algorithm are based on a technique known as **simulated annealing**.

## 3.2 Placement and routing problem

TODO : Give a small description and points to well known algorithms.

**Grid routing**

**Channel routing**

**Layout generation**

**Floorplanning**

## 3.3 Verification, validation and testing

Following is from course notes provided by Prof. Madhav P. Desai, Verification of VLSI circuits, 2010.

" Testing, verification and validation are critical processes in ensuring that the circuit that is being manufactured meets the requirements that motivated its design.

- By **testing**, we normally refer to the process used to determine that the manufactured circuit is functional. Every manufactured circuit needs to be tested, and the result of the test should, with a high degree of confidence, determine that the circuit is functional.

- By **verification**, we normally refer to the process used to confirm that refinements in the design process are consistent with the circuit specifica- tion. For example, when a logic circuit is implemented using transistors, we need to verify that the transistor network is equivalent to the logic circuit which it is supposed to implement. This is done at each refinement step in the design process.
- By **validation**, we normally refer to the process used to confirm that a functional manufactured circuit will fulfill the requirements to which it was designed. This usually involves the construction of a prototype and a test setup which mimics reality to the maximum extent possible.

Typically, the effort needed to implement these processes in the design of complex circuits is substantial; a major fraction of the total product development effort is contributed by the test, verification and validation processes. ”

Following are the major problems in testing.

- Manufacturing defects and fault models.
- The formulation of the test problem.
- The use of fault simulation in identifying fault coverage.
- Test pattern generation for combinational gate-level circuits using the stuck-at fault model.
- Testing of sequential circuits.
- The use of scan techniques to test sequential circuits.
- Built-in self-test (BIST).
- Error correction and schemes for constructing fault-tolerant circuits.

Verification is often done using simulation but this is not enough these days. Use of formal verification is rising.

**Formal Verification of combinational circuits** Given a circtuit, prove some properties holds on it. To do it, we may use Binary decision diagrams or Satisfiability solvers.

**Formal verification of sequential circuits** Given a state machine, prove that some properties holds at a given time. Prove that given two designs or two state machines are equivalent. State exploration techniques, model checking techniques

**Fault diagnosis** Refer to lecture notes. To be updated.

## 4 Automata