Chapter 2

Boolean Representations and Combinatorial Equivalence

This chapter introduces different representations of Boolean functions. It then discusses the applications of these representations for proving Boolean equivalence between two circuits.

2.1 Representing Boolean functions

The objective is to decide whether two circuits compute the same function. We will consider pure Boolean circuits, that is, circuits with no states. The result of the circuit only depends on the *current* values of its outputs. It does not depend on previous values of these inputs.

Regarding terminology, the inputs of a circuit are called *primary inputs* and the outputs of a circuit are called *primary outputs*. This is to make the difference with inputs and outputs at internal gates (also called nodes) of a circuit.

2.1.1 Truth Tables

Truth tables are an easy way to represent a Boolean function. The idea is to construct a table that for each possible combination of primary inputs values, gives the resulting values of the primary outputs. Truth tables are very easy to construct, but they are very large. There are exponentially many rows as inputs. Given a Boolean function of n inputs, its truth table will have 2^n rows.

A simplification is to represent rows that only yield positive results. We will only represent the rows representing combinations of primary input values that result in a positive primary output. Depending on the circuit, this can reduce the truth table quite dramatically.

2.1.2 Disjunctive Normal Form

An alternative is to use the reduced truth table to represent the Boolean function using the Disjunctive Normal Form (DNF). In DNF, a Boolean function is represented by a disjunction of conjunctions.

Note that converting a truth table to DNF has an exponential blow-up.

For instance, the DNF representation of a two-input `xor`-gates is the following:

$$x \text{ x}or\ y \equiv (x \cdot \neg y) + (\neg x \cdot y)$$

Very often, the '$\cdot$' is omitted and negation is denoted using a bar on top of a variable:

$$x \text{ xor } y \equiv x\overline{y} + \overline{x}y$$

The DNF representation is canonical and can serve as a normal form. The main drawback is that DNF expressions are exponential in the size of the original expressions. For every `xor`-gate, the number of literals doubles.

Exercise 2.1.1. To get a sense of how fast DNF expressions are growing, simply compute the DNF expression of two consecutive `xor`-gates. That is, compute the DNF representation of the following expression: $(x \texttt{ xor } y) \texttt{ xor } z$.

2.1.3 Sum of Products

The Sum of Products (SoP) is a generalisation of the DNF. Instead of using AND and OR as building blocks, one can also choose to have AND and XOR as building blocks. For circuits with many XOR-gates – like in arithmetic circuits – such a well-chosen SoP can yield compact representations.

2.1.4 Conjunctive Normal Form

Another normal form can be obtained by writing products of sum. This translates a Boolean expression into its Conjunctive Normal Form (CNF), that is, a conjunction of disjunctions. The CNF is the input format of SAT solvers. It is possible to convert an expression in CNF to an expression in DNF and vice versa. These conversions induce an exponential blow-up of the size of the resulting expression.

The general objective is to translate each gate of a circuit into a CNF formula such that the solutions of the formula coincide with the values of the gate wires that make the gate true. Consider an AND gate with wire a and b as inputs and wire c as output. We need to find a CNF encoding that will be compact and faithfully represent the truth values of the AND gate. We will consider the following cases: (1) if any input is 0, then the output must be 0 and (2) if all inputs are 1, the output must be one.

Case (1). If input a is false then output c must be false. This expression "if $\neg a$ then $\neg c$" is logically equivalent to $\neg\neg a \vee \neg c$, hence $a \vee \neg c$. A similar clause is generated for b: $(b \vee \neg c)$.

Case (2). If all inputs are true, then the output must be true. This logically means "if $(a \wedge b)$ then c", which is logically equivalent to $\neg(a \wedge b) \vee c$. Hence, $(\neg a \vee \neg b \vee c)$.

Finally, an AND gate with inputs a and b and output c is represented by the following CNF formula:

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$$

Consider another example: a XOR gate with inputs a and b and output c. Let's split on whether the output is 1 or 0.

Case 1: the output is 1 (that is c is true). This happens when one of the input is 1 and the other 0. We have "if a is true and b is false then c is true", which is equivalent to the clause $(\neg a \vee b \vee c)$. We have a similar clause for the opposite case: $(a \vee \neg b \vee c)$.

Case 2: the output is 0 (that is $\neg c$ is true). This happens when the two inputs are 1 or the two inputs are 0. We have "if a and b are true, then c is false". This is represented by the clause $(\neg a \vee \neg b \vee \neg c)$. The clause representing the case when both inputs are false is: $(a \vee b \vee \neg c)$.

Finally, the CNF encoding of a XOR-gate is the following CNF formula:

$$(\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (a \vee b \vee \neg c)$$

When translating an entire circuit, the output of a gate will be used as a "helper" variable representing the input to the next gates. By doing so, the CNF representation of a circuit is linear – and not exponential – in the size of the input circuit.

2.1.5 BDDs

Reduced Ordered Binary Decision Diagrams (ROBDDs) provide a canonical representation of Boolean functions. They often are more compact that other canonical representations like Sum-Of-Products for instance. The presentation about BDD is based on Chapter 5 from the textbook by Clarke, Grunberg, and Peled [3] and part of the notes by Henrik Reif Andersen [].

Constructing BDDs

The fundamental notion underlying this section is Shannon's expansion. For a Boolean function f and a given variable x of f, it states that function f can be decomposed into two sub-functions considering the cases where either x is false or x is true. Formally, we have the following equation:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1})$$

Another way to look at Shannon's expansion is to consider the "if-then-else" operator defined as follows:

$$x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

The semantics are that if x holds then y_0, else y_1. So, the expression $x \rightarrow y_0, y_1$ is true if either x true and y_0 is true or if x is false and y_1 is true. Let $t[0/x]$ be expression t where x is replaced with 0 and similarly $t[1/x]$ be t where x is 1, then the following equivalence holds:

$$t = x \rightarrow t[1/x], t[0/x]$$

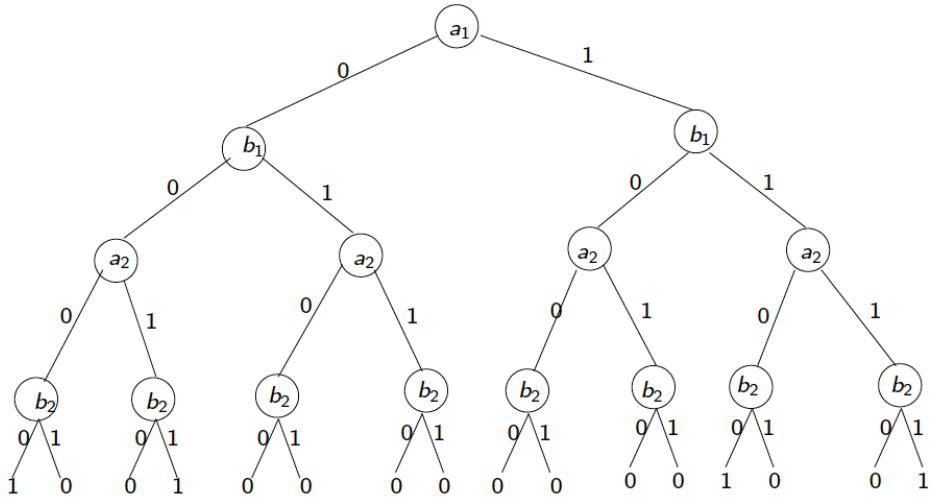

Figure 2.1: Binary decision tree for two-bit comparator.

This exactly is Shannon's expansion.

Using Shannon's decomposition, it is possible to build a binary decision tree. Figure 2.1 shows the tree for a two-bit comparator, that is, for the following Boolean function:

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$

A binary tree basically is a rooted, directed tree with terminal and nonterminal vertices. Each nonterminal vertex v is labelled by a variable noted $var(v)$. Such a vertex has two successors: a successor corresponding to the case where the variable is low (false) noted $low(v)$ and a successor corresponding to the case where the variable is high (true) noted $high(v)$. Graphically, we often identify the successor by a 0 or 1 label on the edges. Each terminal vertex v is either 0 or 1. To know the truth value of formula for a given assignment, one traverses the tree down to a leaf by following the low successor if a variable is false or the high successor if a variable is true. As one can see, such a tree is not a very compact representation.

Bryant [2] showed how to obtain a compact and canonical representation of Boolean functions by placing two restrictions on binary diagrams. The first restriction imposes an ordering among the variables. This means that variables must always appear in the same order along all paths from the root to terminals. Second, there should be no isomorphic sub-trees or redundant vertices in the diagram. To achieve the first restriction, a total order $<$ is imposed among the variables. We require that if any vertex u has a nonterminal successor v, $var(u) < var(v)$. The second restriction is achieved by recursively applying the following rules:

- *Remove duplicate terminals:* Keep only one terminal vertex for each label (0 or

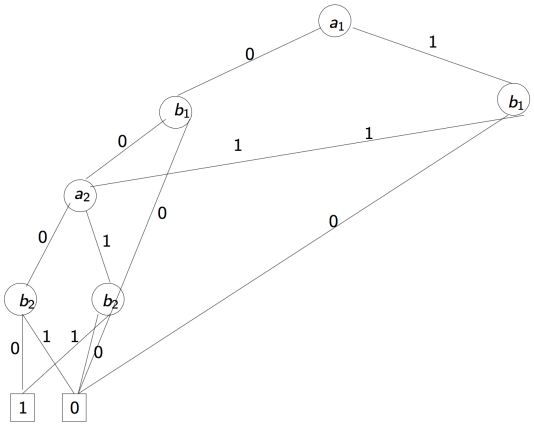

Figure 2.2: ROBDD for two-bit comparator.

1) and redirect all edges to eliminated vertices to the remaining ones.

- *Remove duplicate nonterminals:* If two nonterminals vertices u and v are labelled with the same variables, that is, $var(u) = var(v)$, eliminate one of the vertices and redirect edges to the eliminated vertex to the remaining one.

- *Remove redundant tests:* If the two successors of a nonterminal vertex v are equal, that is, $low(v) = high(v)$, remove v and redirect incoming edges of v to $low(v)$.

Starting with an ordered diagram, the reduced version is obtained by applying these rules until the size can no longer be reduced. We then obtained a Reduced Ordered Binary Decision Diagram (ROBDD). Bryant [2] showed that this reduction can be done in a bottom-up manner and in time that is linear in the size of the original BDD. Figure 2.2 shows the ROBDD obtained from the decision tree in Figure 2.1. This BDD has only 6 nodes.

The variable ordering has a strong impact on the size of the ROBDD. For instance, choosing the ordering $a_1 < b_1 < a_2 < b_2$ the 2-bit comparator results in the ROBDD shown in Figure 2.3, which is larger than the one shown in Figure 2.2. This larger BDD has 9 nodes.

ROBDDs are canonical given a variable ordering. This means that checking equivalence between two ROBDDs reduce to checking equality between two diagrams. Validity reduces to showing equivalence to the trivial diagram composed of one terminal labelled by 0.

To further illustrate the construction of BDDs we consider the following Boolean function:

$$f(a, b, c) = a \vee (b \wedge c)$$

We will use Shannon's expansion to build a Binary Decision Tree for this Boolean

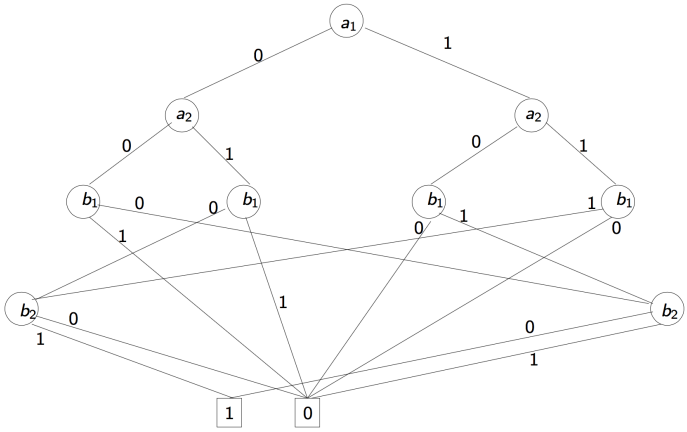

Figure 2.3: ROBDD for two-bit comparator.

function. To apply the expansion we need to determine an ordering of the variables. Let us consider the ordering $a < b < c$.

We first decompose according to a, this gives the following equation:

$$f(a, b, c) = (\neg a \wedge f(0, b, c)) \vee (a \wedge f(1, b, c))$$

The first part of the tree from this equation is pictured in Figure 2.4.

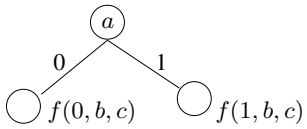

Figure 2.4: First part of the binary decision tree.

We can recursively proceed with decomposing each sub-formula. For instance, we can expand the subtree $f(0, b, c)$ and obtain:

$$f(0, b, c) = (\neg b \wedge f(0, 0, c)) \vee (b \wedge f(0, 1, c))$$

A similar expansion is obtained for $f(1, b, c)$. We then insert these terms in the tree and obtain the tree pictured in Figure 2.5.

We proceed with the four leaves and then compute the values of f where all literals have been assigned a definite value. We obtain the Binary Decision Tree pictured in Figure 2.6.

We apply the rules proposed by Bryant [2] to transform this BDT into a Reduced Ordered BDD. We first apply the rule "remove redundant tests"[1], this means that we eliminate all nodes where the value at the end of the left and the right branches is the same. There are three cases where this applies. We obtain the graph in Figure 2.7.

[1]The order in which the rules are applied does not matter.

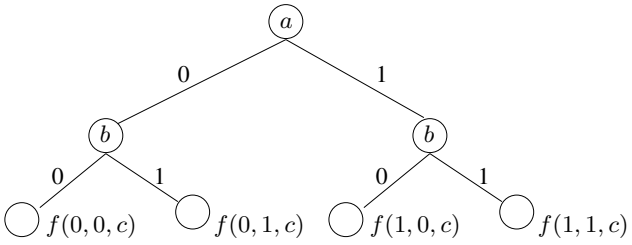

Figure 2.5: The binary decision tree continued ...

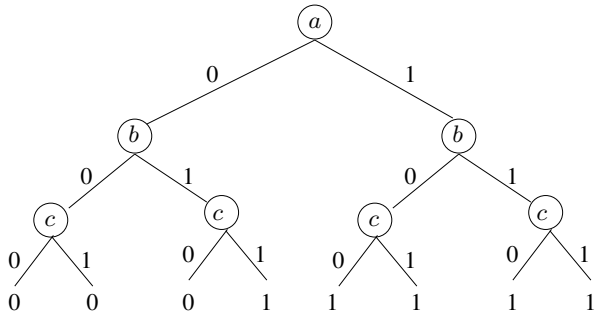

Figure 2.6: The binary decision tree

The same rule applies once more to the true branch of a. After applying this rule, we apply the rule "remove duplicate terminals" and obtain the ROBDD in Figure 2.8.

The size of a ROBDD depends on the variable ordering. The following question illustrates this point.

Exercise 2.1.2. For which ordering(s) is the size of the ROBDD for our function f maximal? (Note: The size is counted in the number of nodes and edges)?

Finding an optimal ordering is infeasible and there exist functions with no optimal ordering, like bitwise multiplication. In such cases, explicit algorithms might perform better than symbolic ones.

Operations on BDDs

The main advantage of ROBDDs is that they can be manipulated efficiently. In particular, checking that two ROBDDs are equal requires constant time. Negation is also very easy and simply requires to negate the two leaves[2] of a ROBDD.

Other operations are performed using the Apply algorithm proposed by Bryant [2] and that can be used to compute any 2-argument logical operation. The main idea is to use Shannon's decomposition to break problems into sub-problems.

[2]Note that any ROBDD has only two leaves, namely, 1 for "true" and 0 for "false".

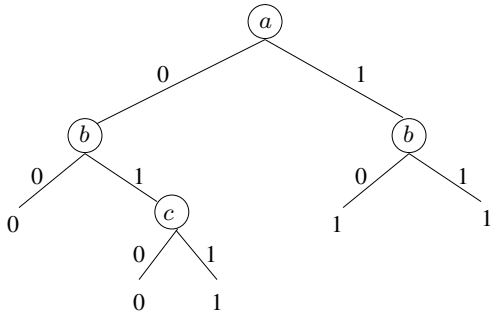

Figure 2.7: Reducing the BDT with "remove redundant tests".

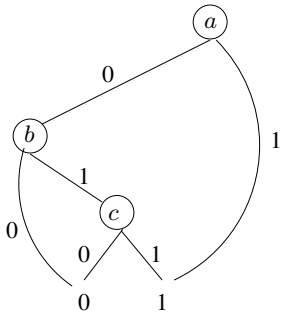

Figure 2.8: Reducing the BDT to a ROBDD.

Let $\star$ be an arbitrary 2-argument logical operation. Let f and f' be two Boolean functions. Let v be the current vertex of f and v' be the current vertex of f'. The algorithm basically is the following case distinction:

- If v and v' are both terminals, $f \star f' = value(v) \star value(v')$.
- If $var(v) = var(v')$, apply Shannon's expansion:

  $$f \star f' = \neg var(v) \wedge (f|_{var(v)\leftarrow 0} \star f'|_{var(v)\leftarrow 0}) \vee var(v) \wedge (f|_{var(v)\leftarrow 1} \star f'|_{var(v)\leftarrow 1})$$

  to break the problem into subproblems. The resulting ROBDD will have a new node w with $var(w) = var(v)$, $low(w)$ will be the ROBDD for $(f|_{var(v)\leftarrow 0} \star f'|_{var(v)\leftarrow 0})$ and $high(w)$ will be the ROBDD for $(f|_{var(v)\leftarrow 1} \star f'|_{var(v)\leftarrow 1})$.
- If $var(v) < var(v')$, $f'|_{var(v)\leftarrow 0} = f'|_{var(v)\leftarrow 1} = f'$ since f' does not depend on $var(v)$. In this case, the Shannon's expansion reduces to:

  $$f \star f' = \neg var(v) \wedge (f|_{var(v)\leftarrow 0} \star f') \vee var(v) \wedge (f|_{var(v)\leftarrow 1} \star f')$$
- If $var(v') < var(v)$, similar to previous case.

By reducing and caching intermediate results, the algorithm is polynomial. The number of sub-problems is bounded by the product of the size of the ROBDDs for f and f'.

We illustrate the *Apply* algorithm using our small example:

$$f(a) \vee f'(b,c)$$

where $f(a) = a$ and $f'(b,c) = b \wedge c$.

We consider the ordering $a < b < c$. We assume we have the ROBDDs of each function, that is, the graphs pictured in Figure 2.9.

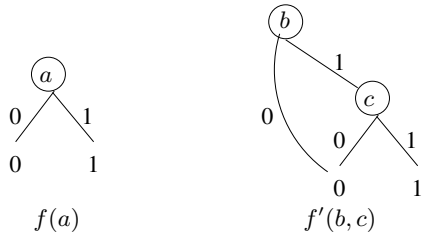

Figure 2.9: ROBDDs for f and f'.

We now use Apply and Shannon's expansion to compute the disjunction of these two ROBDDs. Since the roots of the ROBDDs are not terminal vertices and since the variables in the two roots are not the same we need to determine which of the two variables comes first in the ordering. Since we consider the ordering where $a < b$, we apply the formula from the third case of the aforementioned algorithm:

$$f(a) \vee f'(b,c) = (\neg a \wedge (f(0) \vee f'(b,c))) \vee (a \wedge (f(1) \vee f'(b,c)))$$

We now start to build a new ROBDD as illustrated in Figure 2.10.

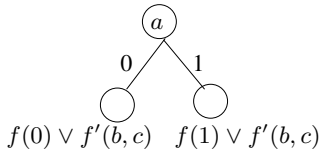

Figure 2.10: Partial ROBDDs for $f \vee f'$.

Now, we have $f(0) = 0$ and $f(1) = 1$ (since $f(a) = a$). Therefore, the left branch simplifies to $f'(b,c)$ and the right branch to 1. We already have a ROBDD for $f'(b,c)$. We simply insert it as the left branch. Finally, we obtain the ROBDD pictured in Figure 2.8 for the formula:

$$f(a,b,c) = a \vee (b \wedge c)$$

The last operation we will need on ROBDDs is existential quantification. Assume a Boolean function f and x a variable of f. Using Shannon's decomposition, we have:

$$\exists x.f = \neg x \wedge f|_{x \leftarrow 0} \vee x \wedge f|_{x \leftarrow 1}$$

That is, f is true if either f is true when x is false, or f is true when x is false. We will come back to existential quantification in Section 3.2.1 of Chapter 3.

2.1.6 Directed Acyclic Graphs

Another way to represent Boolean functions is to use Directed Acyclic Graph. Consider the 2-bit comparator, that is, the following equation:

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$

Using the if-then-else operator defined earlier, we can also expand this expression to the following:

$$\begin{aligned}
t &= a_1 \rightarrow t_1, t_0 \\
t_0 &= a_2 \rightarrow t_{01}, t_{00} \\
t_1 &= a_2 \rightarrow t_{11}, t_{10} \\
t_{00} &= b_1 \rightarrow 0, t_{000} \\
t_{01} &= b_1 \rightarrow 0, t_{010} \\
t_{10} &= b_1 \rightarrow t_{101}, 0 \\
t_{11} &= b_1 \rightarrow t_{111}, 0 \\
t_{000} &= b_2 \rightarrow 0, 1 \\
t_{010} &= b_2 \rightarrow 1, 0 \\
t_{101} &= b_2 \rightarrow 0, 1 \\
t_{111} &= b_2 \rightarrow 1, 0
\end{aligned}$$

We again obtain a tree – note that it is already smaller than the one in Figure 2.1 – where t is the root and each t_i are sub-trees corresponding to the expansion following, in order, variables a_1, a_2, b_1, b_2. Nodes in this tree are equal, as it can be seen in the above expressions. For instance, $t_{111} = t_{010}$ and $t_{000} = t_{101}$. By substituting all equal expressions, we obtain a "binary decision diagram", that is no longer a tree but a DAG. For instance, in the above example, we obtain:

$$\begin{aligned}
t &= a_1 \rightarrow t_1, t_0 \\
t_0 &= a_2 \rightarrow t_{01}, t_{00} \\
t_1 &= a_2 \rightarrow t_{11}, t_{10} \\
t_{00} &= b_1 \rightarrow 0, t_{000} \\
t_{01} &= b_1 \rightarrow 0, t_{111} \\
t_{10} &= b_1 \rightarrow t_{000}, 0
\end{aligned}$$

$$\begin{aligned} t_{11} &= b_1 \rightarrow t_{111}, 0 \\ t_{000} &= b_2 \rightarrow 0, 1 \\ t_{111} &= b_2 \rightarrow 1, 0 \end{aligned}$$

Exercise 2.1.3. Is it possible to even reduce this DAG using Bryant's rule?

Exercise 2.1.4. Re-do the construction of the tree and its reduction to a DAG using sharing with the following variable order: a_1, b_1, a_2, b_2.

2.1.7 AIGs

And Inverter Graphs (AIGs) are directed and acyclic graphs representing Boolean functions. They consist in two input nodes representing conjunction and edges with an optional label indicating negation. The size of an AIG is given by the number of AND nodes in it. The number of logic levels is the number of AND-gates on the longest path from a primary input to a primary output. If you remember your old course 2IC30 on Computer Systems, you should remind yourself about the completeness of the NAND gates. That is, any Boolean function can be represented using two-input NAND gates only. Therefore, AIGs can also represent any Boolean functions.

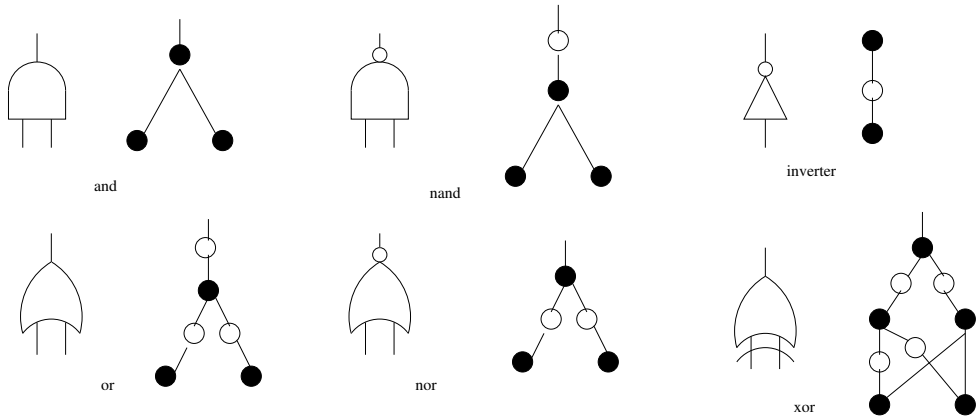

Figure 2.11: Basic gates and their representation using And Inverter Graphs

Figure 2.11 shows the AIGs representing basic gates. The only slighly more difficult one is the `xor` gate. One has to express this function using conjunction and negation only. This is given by the following expression:

$$x \text{ x}or\ y = \neg(x \wedge y) \wedge \neg(\neg x \wedge \neg y)$$

Figure 2.12 shows a circuit and its representation as an AIG. The AIG is constructed in a bottom-up manner starting from the Primary Outputs (POs). Note that each time all gates are expressed using `and`-gates and inverters only using equivalences like the ones shown in Figure 2.11. When a PI is reached, the construction stops with a leave

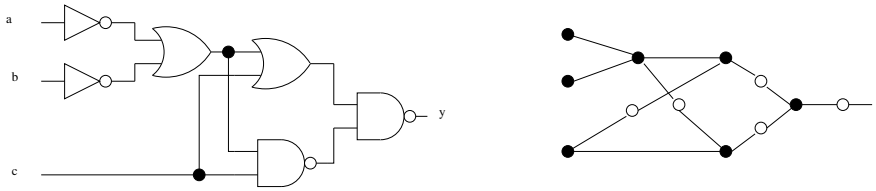

Figure 2.12: A circuit and its representation as an AIG.

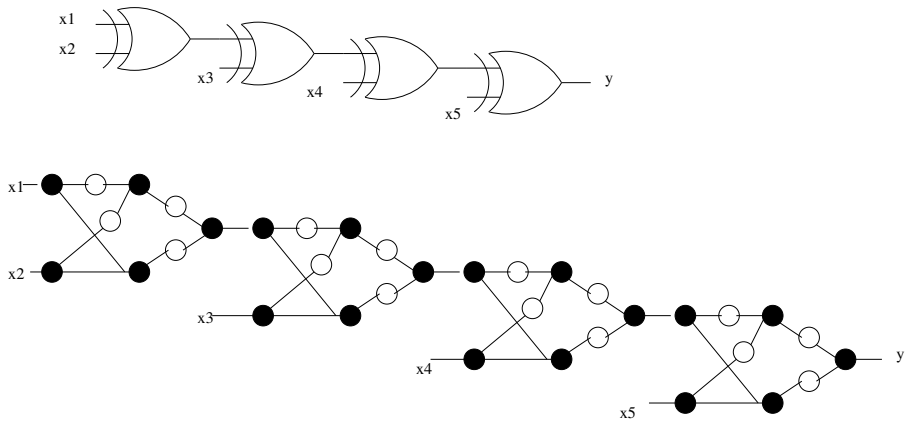

Figure 2.13: A cascade of four `xor`-gates.

node representing this input. Otherwise, the construction proceeds recursively for the fanins of the node.

The time needed to construct AIGs and their size is proportional to the size of the input, being a circuit, a SoP expression, or a BDD.

A nice feature of AIGs is that they can be smaller than Sum-Of-Products or BDDs. Figure 2.13 shows four cascaded `xor`-gates and their AIGs. The SoP representation of this function would be exponentially large, whereas the AIG is rather small. Also, there are BDDs that are exponential – e.g., multiplier circuits – while the size of an AIG always is proportional to the size of the input circuit, SoP, or BDD.

The main drawback of AIGs is that they are not *canonical*, that is, testing for equality is not trivial. AIGs may contain syntactically distinct but functionally equivalent – that is redundant – nodes. Manipulating such AIGs is also not efficient.

To solve this issue, algorithms have been developed introduce some canonicity, in particular, Functionally Reduced AIGs (FRAIGs) [5]. The idea is to reduce AIGs such that they do not exist any pairs of functionally equivalent nodes. FRAIGs are semi-canonical in the sense that a Boolean function may have distinct FRAIGs structures. Recent work [5] proposes an algorithm that generates on-the-fly an AIG that is ensured to be functionally reduced.

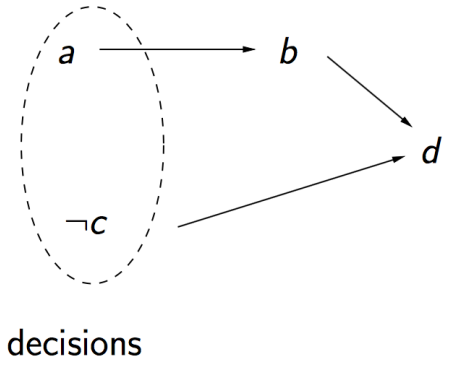

Figure 2.14: Decisions and the resulting implication graph.

2.2 Boolean satisfiability (SAT)

Boolean satisfiability consists in checking whether there exists an assignment of the variables of a Boolean formula that makes this formula "true". This problem is well-known and is very hard (NP-Complete). Still, there are several SAT solvers that can effectively check satisfiability of formulas with thousands of variables. In this section, we briefly recall the basic algorithm behind SAT solvers.

Our presentation follows a tutorial given by K. McMillan at CAV 2003.

The basic SAT algorithm has been proposed by Davis, Putnam, Loveland, and Logeman in the 60's. Therefore it is called the DPLL algorithm. The objective is to check satisfiability of a CNF formula:

- literal: p or $\neg p$
- clause: disjunction of literals
- CNF: conjunction of clauses

The basic methods consists in three steps:

- Branch: make some arbitrary decisions in assignment values to variables.
- Propagate these assignments, that is, computing an implication gap.
- Use conflicts to guide inferences steps.

Implication graph

The implication graph is obtained by performing *unit propagation* (UP) or also called *Boolean constraint propagation* (BCP). Consider the CNF formula $(\neg a \vee b) \wedge (\neg a \vee c \vee d)$. The first step always is to make arbitrary decisions, say, assign true to a – so a is true – and assign false to c – so, $\neg c$ is true. The implication graph resulting of

these decisions is shown in Figure 2.14. From the first clause, namely $\neg a \vee b$, and the assignment a is true, we need to assign true to b to make this clause true, so b holds. From the second clause, namely, $\neg a \vee c \vee d$, and the decisions a and $\neg c$, we also conclude that d must be true to satisfy this second clause. In conclusion, from the arbitrary decision a and $\neg c$, the implication graph tells us that we have to choose b and d. For this formula, we have found a satisfiable assignment $a \wedge b \wedge \neg c \wedge d$. So, the formula is SAT.

Conflicts and resolution

Another step in SAT solving is *resolution*. Resolution is used to solve conflicts. Consider the following clauses:

- $a \vee \neg b \vee c$
- $\neg a \vee \neg b \vee d$

There is here a conflict between a and $\neg a$ that might appear for some decisions, e.g., $b \wedge \neg c \wedge \neg d$. From these two clauses, resolution will remove the two occurrences of a and produce the resolved clause $\neg b \vee c \vee d$. In SAT, the implication graph is used to guide where resolution must be applied.

Consider the following CNF formula:

$$(\neg a \vee b) \wedge (\neg a \vee c \vee d) \wedge (\neg b \vee \neg d)$$

Assume we make the same arbitrary decisions as above, namely, a and $\neg c$. As explained before, the implication graph obtained by propagating the decisions through the first two clauses will produce the assignment $a \wedge b \wedge \neg c \wedge d$. This assignment conflicts with the last clause. We can apply resolution between the last two clauses. This creates the clause $\neg b \vee c$. We then apply resolution between this new clause and the first one. This gives us $\neg a \vee c$, that is, the opposite decision as the initial arbitrary one. We can then propagate this decision again. This will make the first two clause true independently of the value of b. This leaves us with free choices for b and d to make the last clause true. We found an assignment making the formula true, so it is SAT.

Implementation

In practice, many heuristics are implemented to choose variables to start with and which values they should have. Also, heuristics are used to choose how to propagate choices and in selecting clauses for resolution. Also, heuristics are needed to decide when to stop applying resolution.

The basic SAT algorithm is given in Figure 2.15. The current assignment (noted CS) initially is empty. If resolution derives the empty clause, then the problem is UNSAT and the algorithm stops. Otherwise, if a conflict is detected, resolution is used to solve and then to backtrack and change some decisions. If all variables have an assignment, then a satisfiable assignment has been found and the algorithm returns SAT.

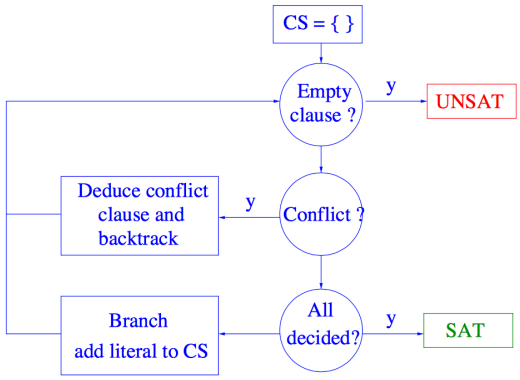

Figure 2.15: Basic SAT algorithm.

2.3 Combinatorial Equivalence Checking (CEC)

One of the main application of formal verification techniques is Boolean equivalence checking. The idea is to check whether two Boolean functions are equivalent, that is, for all possible input vectors they produce the same outputs.

2.3.1 Using SoP, CNF or DNF

SoP, CNF, and DNF are all normal forms. To check equality between two circuits, one can simply rewrite a circuit into one of these normal forms. There is an exponential blow-up created by this translation, if done naively. Section 2.1.4 presented a translation that using helper variables is linear in the size of the input circuits. In practice, CNF is the input format for SAT solvers and it will work for many cases. CNF is a popular representation for Boolean circuits. When performance are becoming an issue BDD and AIGs are modern alternative (see next sub-sections).

2.3.2 Using BDDs

ROBDDs are also a normal form. To check the equivalence between two functions, build their ROBDD and check equality of their root pointer. Building ROBDDs is more efficient than SoP. BDDs have been the main tool for equivalence checking for a long time.

2.3.3 Using AIGs

AIGs – in contrast to SoP and BDDs – do not have canonical representations. Checking for equality seems therefore more difficult. This issue is solved using a SAT solver to determine local equality between AIG nodes. When equal nodes are found, they are

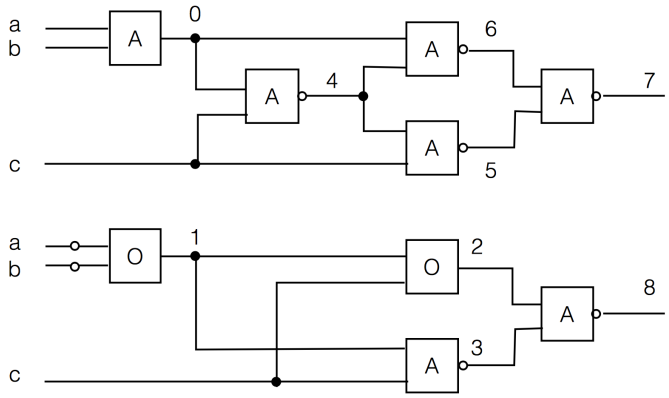

Figure 2.16: Two combinatorial circuits.

removed until the remaining AIGs are equal. Because SAT solvers are very efficient, the queries sent to the SAT solver are answered very quickly. Nowadays, AIGs are at the core of modern CEC checkers.

The technique proceeds in three steps:

1. Random Simulation: First candidate equivalent nodes are found by simulating the circuits using random test vectors
2. Build AIGs: More equivalent nodes are found by constructing an AIG of the combination of the two circuits.
3. SAT Sweeping: Candidates nodes are used to guide calls to the SAT solver

Random simulations

Consider the two circuits in Figure 2.16, where a box with A denotes an AND-gate, a box with O an OR-gate, and a small circle an inverter. The first step is to run random simulation to find potential equivalent nodes. Assume a random test vector assigns true to all inputs. These inputs propagate to give the following values to each node:

node	value
0	true
1	false
2	true
3	true
4	false
5	true
6	true
7	false
8	false

From this value we can cut the set of nodes in two sets, the sets of node that are true and the set of nodes that are false. We get the following classes:

$$\{0, 2, 3, 5, 6\}, \{1, 4, 7, 8\}$$

Assume another random test vector assigns false to all inputs. These inputs propagate and give the following values:

node	value
0	false
1	true
2	true
3	true
4	false
5	true
6	true
7	false
8	false

Node 0 has the value false, while all other nodes in its equivalence class are true (these are nodes 2,3,5,6). We need to remove node 0 from this equivalence class. Node 0 is then a singleton and is therefore not equivalent to any other node. Nodes 1 and 4 are true while nodes 7 and 8 are false. We need to split the equivalence class $\{1, 4, 7, 8\}$ into two classes. We now have the following equivalence classes:

$$\{2, 3, 5, 6\}, \{1, 4\}, \{7, 8\}$$

Assume a random test vector assigning false to a and b and true to c. This results in the following values:

node	value
0	false
1	true
2	true
3	false
4	true
5	false
6	true
7	true
8	true

Nodes 1 and 4 have the same value. Nodes 7 and 8 also have the same values. This confirms that they seem to be equivalent. Nodes 2 and 6 are true while the other nodes of their equivalence class, namely 3 and 5, are false. We therefore need to split them into two equivalence classes. We obtain:

$$\{2, 6\}, \{3, 5\}, \{1, 4\}, \{7, 8\}$$

Finally, assume a random test vector assignment true to a and b and false to c. This results in the following values for the nodes:

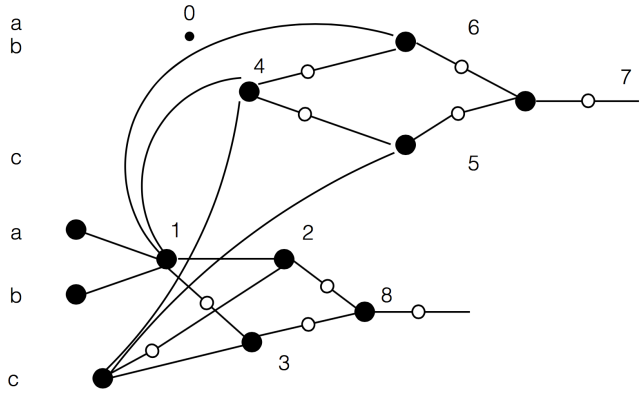

Figure 2.17: Combining two AIGs into one.

node	value
0	true
1	false
2	false
3	true
4	true
5	true
6	false
7	true
8	true

Nodes 1 and 4 have different values but form an equivalence class. These nodes are split into two classes. Each class is a singleton. So, these nodes are equivalent to no other nodes. This gives us the following equivalence classes. Each class identify a pair of potentially equivalent node.

$$\{2, 6\}, \{3, 5\}, \{7, 8\}$$

Building AIGs

The idea is to build the AIG of the two circuits and merge these two AIGs into one. In practice, one can build one AIG and then stepwise add the AIG of the other while the AIG is constructed. Figure 2.17 shows the resulting AIG of combining the two circuits in Figure 2.16.

SAT sweeping

From the simulations, we obtained potential equivalent nodes and we now have an AIG representation of the two circuits. The last step is to call a SAT solver to actually check whether the candidate nodes are indeed equivalent. Assume we first ask whether nodes

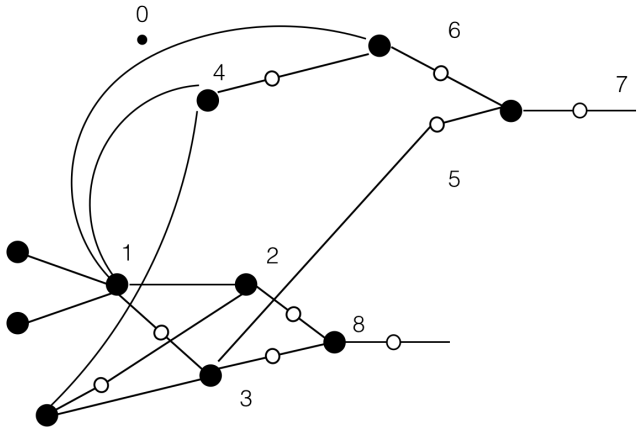

Figure 2.18: Combining two AIGs into one.

3 and 5 are equivalent and that the SAT solver proves this. We can then in the AIG remove node 5 and directly connect the open edge of node 7 to node 3 as shown in Figure 2.18. The same procedure applies to nodes 2 and 6. We can them remove node 6 and make the open edge of node 7 point to node 2. We are left with an AIG where nodes 7 and 8 structurally points to the same nodes, see Figure 2.19. These two nodes are therefore equivalent.

2.4 Conclusion

This chapter presented different ways of representing Boolean functions. When using normal forms, equivalence checking is simple. The issue is that the size of the Boolean representation grows exponentially with the size of the circuits to be compared. AIGs provide a more compact representation but are not canonical. Explicit tests using SAT solvers are needed to identify equivalent nodes.

2.5 Exercises

2.5.1 BDDs

Exercise 2.5.1. Draw the ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3)$ with orderings $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$ and $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$. Which ordering would you recommend for constructing a ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3) \wedge ... \wedge (x_k \Leftrightarrow y_k)$?

Exercise 2.5.2. Draw the ROBDD for $\neg x_1 \wedge (x_2 \Leftrightarrow x_3)$ with ordering $x_1 < x_2 < x_3$.

Exercise 2.5.3. Construct the ROBDD for x_1 and $x_1 \Rightarrow x_2$ using whatever ordering you want. Use the APPLY procedure to build the ROBDD for their disjunction.

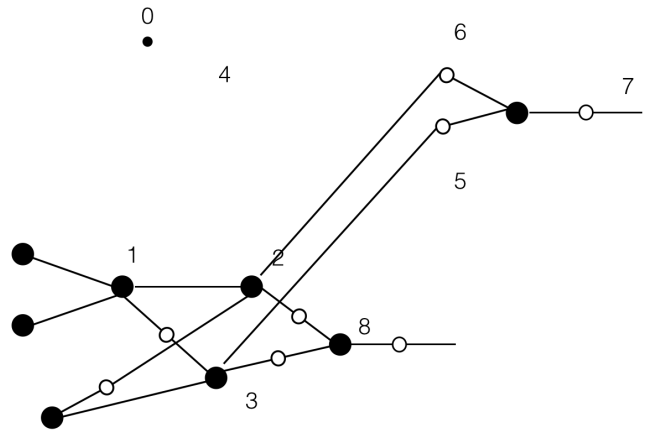

Figure 2.19: Combining two AIGs into one.

Exercise 2.5.4. Construct the ROBDD for $\neg(x_1 \wedge x_3)$ and $(x_2 \wedge x_3)$ with the ordering $x_1 < x_2 < x_3$. Use APPLY to construct the ROBDD for $\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$.

2.5.2 AIGs

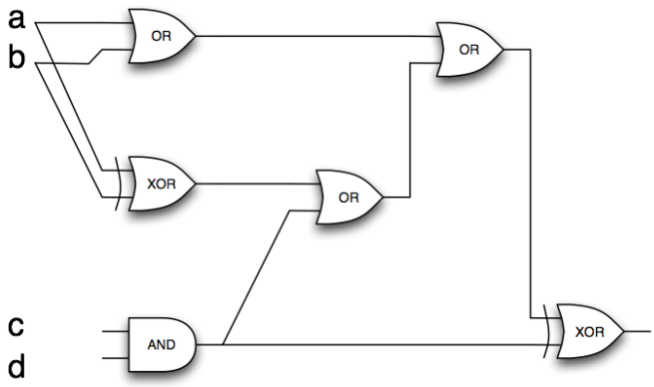

Figure 2.20: A circuit.

Exercise 2.5.5. Build an AIG for the circuit in Figure 2.20.

Exercise 2.5.6. Draw an AIG representing each one of the circuits in Figure 2.21. Then, build the combined AIG for the two circuits. These circuits are equivalent. Is the combined AIG functionally reduced? Would techniques like SAT Sweeping be effective here? What query would you submit to a SAT solver?

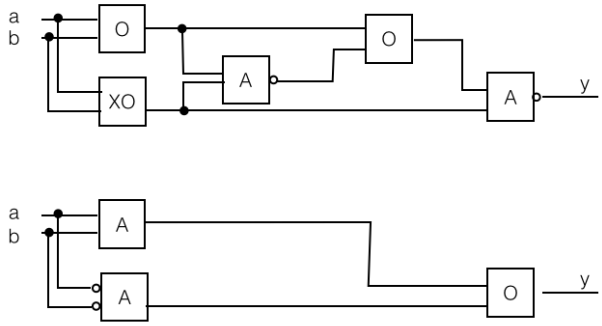

Figure 2.21: Two circuits.

2.5.3 CNF encoding

Exercise 2.5.7. Write the CNF encoding for the following gates: NAND, OR,NOR, XNOR, and NOT.