Lecture Notes for

Logic and Computability

Course Number: IND04033UF

Contact

Bettina Könighofer
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
bettina.koenighofer@iaik.tugraz.at

Table of Contents

3 Binary Decision Diagrams **3**

3.1 Binary Decision Diagram 3

3.2 Reduced Ordered BDDs 8

3.3 Construction of Reduced Ordered BDDs 13

3

Binary Decision Diagrams

In this chapter, we introduce an efficient data structure to store and manipulate propositional formulas in the form of *reduced ordered binary decision diagrams* (*BDDs*). For a given variable order, a ROBBD gives a *canonical* (unique) representation for a given formula. This means that semantically equivalent formulas (formulas can have a different syntax but have the same semantic meaning) are represented by the same BDD.

BBDs are used in formal verification tools for hardware and software where BDDs are used to represent models of the system.

3.1 Binary Decision Diagram

A *binary decision diagram* (BDD) represents a formula using a graph-based data structure. We start this chapter by introducing a basic version of BDDs and discussing its individual elements. We then extend the concept to Reduced Ordered BDDs (ROBDDs) which are a more efficient data structure that we will use from there on.

Definition 3.1 (Directed Acyclic Graph) A *directed acyclic graph* (DAG) is a directed graph that does not contain any directed cycle. A node of a DAG is *initial* if the node has no incoming edges. A node is called *terminal* if the node has no outgoing edges.

Definition 3.2 (Binary Decision Diagram.) A *binary decision diagram* (BDD) is a DAG that represents a propositional formula f. A BDD consists of a *function node* representing the formula f, *internal* nodes that represent individual

variables of f, and two *terminal* nodes representing the truth values 1 and 0. The function node connects to the internal node on the first level. All internal nodes have exactly two outgoing edges, namely the *then-edge* and the *else-edge*, connecting the internal node to either a different internal node or a terminal node.

We follow the convention that else-edges are marked by circles, whereas then-edges are simple connections without annotations.

Example. Figure 3.1 shows a binary decision diagram for a formula with four variables a, b, c, and d.

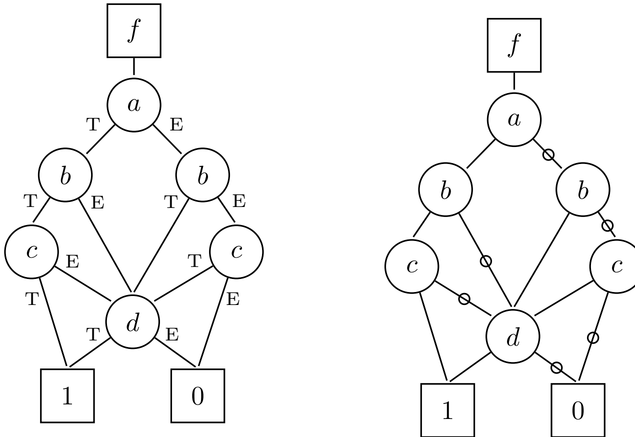


Figure 3.1: A simple BDD with different edge labeling.

Evaluating a Model using a BDD. To evaluate a formula f under a given model $\mathcal{M}$, we traverse the BDD according to the truth value assignments in the model. For example, when reaching an inner node labeled with a and $\mathcal{M}$ assigns a to **T**, we follow the *then*-edge. If $\mathcal{M}$ assigns a to **F**, we follow the *else*-edges. Since the BDD is a DAG, we eventually reach a terminal node. If the terminal node 1 is reached, $\mathcal{M}$ is a satisfying assignment for f ($\mathcal{M} \models f$). Otherwise, $\mathcal{M}$ is a satisfying assignment for f ($\mathcal{M} \not\models f$).

Constructing formulas from BDDs. We can use the same technique to find a formula $\hat{f}$ that is semantically equivalent to the formula f represented by the BDD. The formula $\hat{f}$ is then in the so-called *disjunctive normal form.* We therefore call $\hat{f}$ the DNF of f, or DNF(f).

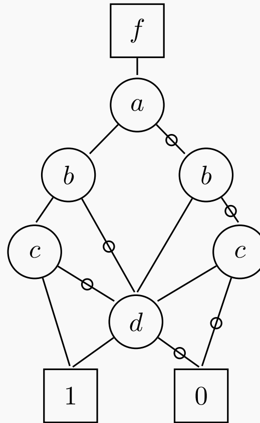
Definition 3.3 (Disjunctive Normal Form) A formula f is in *disjunctive normal form* (DNF) if it is a *disjunction of conjunctions.*

To compute DNF(f), we enumerate all paths through internal nodes that lead to the terminal node 1. Each path represents a model $\mathcal{M}$ that makes the f true, thus a path will be represented as a conjunction of literals.

- If a path follows the *then-edge* of a node labeled with a (i.e., $a = \mathbf{T}$ in the corresponding model), we add a to the conjunction representing this path.

Exercise 3.1

Consider the following BDD:



Evaluate the formula f represented by the following BDD under the given models:

$$\mathcal{M}_1 = \{a = \mathbf{T}, b = \mathbf{T}, c = \mathbf{F}, d = \mathbf{T}\}, \mathcal{M}_2 = \{a = \mathbf{T}, b = \mathbf{F}, c = \mathbf{T}, d = \mathbf{F}\}.$$

Solution.

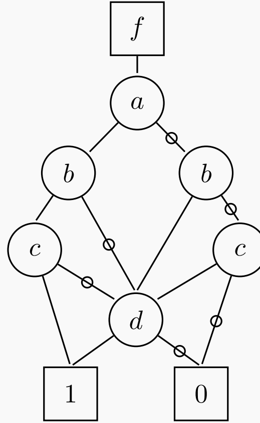
- $\mathcal{M}_1$: We follow the edges according to the truth assignments in $\mathcal{M}_1$ and reach the terminal node 1. Therefore $\mathcal{M}_1 \models f$.
- $\mathcal{M}_2$: We follow the edges according to the truth assignments in $\mathcal{M}_2$ and reach the terminal node 0. Therefore $\mathcal{M}_2 \not\models f$. Note that the truth value for c did not influence the outcome.

- If a path follows the *else-edge* of a node labeled with a (i.e., $a = \mathbf{F}$ in the corresponding model), we add $\neg a$ to the conjunction representing this path.

To obtain DNF(f), the conjunctions representing the individual paths (satisfying models) are connected via disjunctions.

Exercise 3.2

Consider the following BDD:



Compute the DNF of the formula f represented by the BDD.

Solution.
The first path that ends in the terminal node 1 is given by always taking the *then-edges.* This results in the first conjunction $(a \wedge b \wedge c)$. The second path is given by taking the *then-edges* on levels 1 and 2, the *else-edge* on level 3, and the *then-edge* on level 4. This path gives us the second conjunction $(a \wedge b \wedge \neg c \wedge d)$. We construct the formula f by enumerating and encoding all paths that end in the terminal node 1:

$$\begin{aligned} \text{DNF}(f) = &(a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c \wedge d) \vee \\ &(a \wedge \neg b \wedge d) \vee (\neg a \wedge b \wedge d) \vee (\neg a \wedge \neg b \wedge c \wedge d) \end{aligned}$$

Final Representation of a BDD. We are adapting our initial representation of a BDD to match the BDD representation used in several literature. We will discuss the following changes in the representation:

1. Complement attribute and single terminal node,
2. dangling edges, and
3. complements only at else-edges.

Complement attribute and single terminal node. An edge can be *negated* by marking the edge with a *full circle*. *Completed* edges allow us to represent a BDD using only one terminal node. We will use the terminal node 1. Whenever an edge would lead to the terminal node 0, we complement the edge and direct it to 1. By doing so, the terminal node 0 can be removed since all edges can be

redirected to the terminal node 1.

An example BDD using the complement attribute is shown in Figure 3.2.

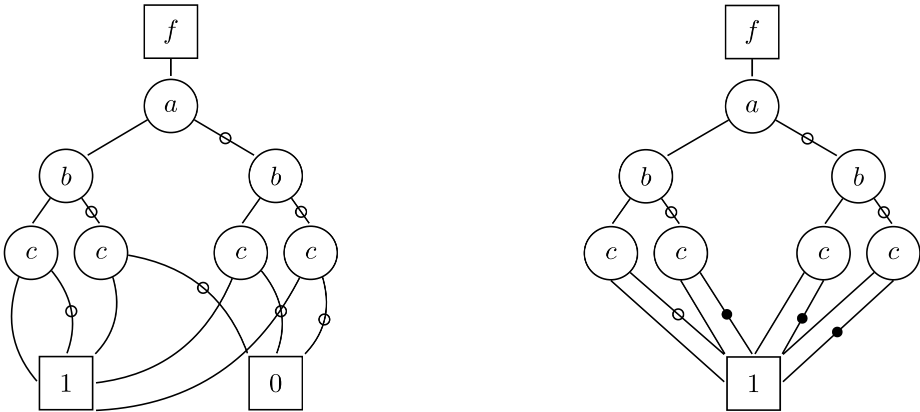


Figure 3.2: Left: Simple BDD. Right: BDD *with* complemented edges.

Dangling edges. Since we only have a single terminal node, there is no need to draw the terminal node anymore. Instead, we simply draw dangling edges, i.e., every dangling edge connects to the terminal node 1. Note, that this is only done to simplify the drawing of BDDs on paper. Figure 3.3 shows the example BDD from above with dangling edges.

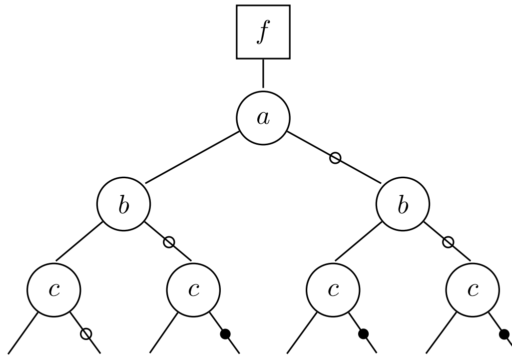


Figure 3.3: BDD with dangling edges.

Complements only at else-edges. Finally, we require that the complement attribute (full circle) is only present at *else-edges*. If during the construction of a BDD, a compliment is introduced at a then-edge, the complement of this edge can be removed by pushing the negation upwards. This approach is discussed in detail in Section 3.3.

3.2 Reduced Ordered BDDs

Using the definitions from above, let's analyze the worst-case size of a BDD.

Size of a BDD. Each internal node representing a variable a has exactly two sub-trees, one for the assignment $a = \mathbf{T}$ and one for $a = \mathbf{F}$. In the worst case, for a propositional formula f with n variables, the BDD representing f may have $2^n - 1$ internal nodes. Hence, BDDs have the same worst-case size as truth tables.

However, BDDs often contain *redundancies* that can be removed which is the topic of this section. *Reduced BDDs* still have an exponential worst-case size complexity, however, they are often small for many practical examples. Additionally, we require that BDDs are ordered such that the resulting BDD provides a canonical representation of the formula. This leads to *reduced and ordered BDDs*.

We start by defining what it means that a BDD is ordered.

Definition 3.4 (Ordered BDDs) . Let $[x_1, \ldots, x_n]$ be an ordered list of variables without duplicates. We say that a *BDD has the variable order* $x_1 < x_2 < \cdots < x_n$ if for every occurrence of x_i followed by x_j along any path in the BDD, we have $i < j$.

Next, we discuss two types of redundancies that need to be removed for a BDD to be reduced:

- Duplicate Sub-BDDs, and
- Redundant Nodes.

Duplicate Sub-BDDs. A BDD may have two or more identical sub-BDDs which represent the same sub-formula. If a BDD contains such identical sub-BDDs, we remove all but one of these sub-BDDs. All edges that lead to one of the removed sub-BDDs are redirected to the sub-BDDs left in the BDD.

Example. Consider the BDD from Figure 3.3. Three sub-BDDs are representing the formula $\neg c$. We can remove two of these sub-BDDs representing $\neg c$ and redirect the edges to the remaining one. The result after removing this redundancy is shown in Figure 3.4.

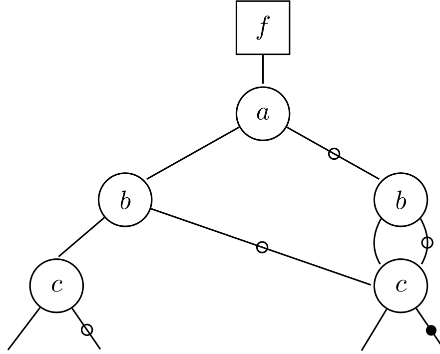


Figure 3.4: BDD with no duplicate sub-BDDs.

Redundant Nodes. A node is redundant if its *then-edge* and its *else-edge* both lead to the same node and neither of the edges is negated (or both of them are negated). Special cases are nodes with two dangling edges with both or none having the complement attribute. Redundant nodes can be removed from the BDD without changing its meaning. Note, that a node is not redundant if one of its outgoing edges has the compliment attribute since the decision in such a node (following the then or else-edge) makes a difference in whether paths passing this node evaluate to true or to false.

Examples of redundant nodes and non-redundant are illustrated in Figure 3.5.

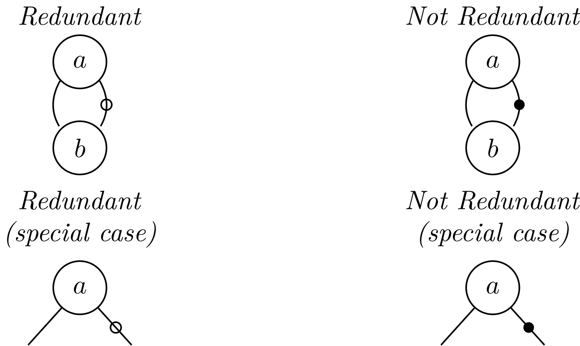


Figure 3.5: Examples of redundant and not redundant nodes.

Example. Figure 3.6 shows the BDD after removing redundant nodes from Figure 3.4. We can remove the node representing b on the right as it is redundant, as well as the node representing c on the left, as both of its outgoing edges are not negated.

Canonicity. In case a BDD is *reduced and ordered*, a BDD forms a canonical representation of a formula in propositional logic. For a BDD to be reduced, a

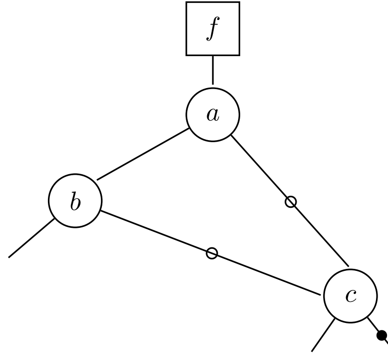


Figure 3.6: Optimization 3: BDD with no redundant nodes.

BDD cannot have any redundant node or duplicate sub-BDD.

A BDD that is reduced and ordered is also referred to as ROBDD. From here on, we will always assume that a BDD is reduced and ordered = is an ROBDD.

Example. Consider the formulas $f_1 = \big(a \wedge (b \vee c)\big)$ and $f_2 = \big(a \wedge (a \vee b) \wedge (b \vee c)\big)$. The two formulas are syntactically different but semantically equivalent. Therefore, f_1 and f_2 are represented by the same reduced and ordered BDD.

Satisfiability and Validity

The canonicity of ROBDDs leads to a useful side effect. To decide whether a formula is satisfiable or valid can be done in constant time, assuming that the ROBDD for the formula has been constructed.

We can easily see this, as a valid formula needs to lead to the terminal node 1 on *every path starting from the function node*. Therefore, any internal node needs to be redundant and only the function node and terminal node representing 1 are left.

Likewise, an unsatisfiable formula needs to be represented by the BDD consisting of the function node and a complemented edge to the terminal node 1. Therefore, any BDD that is different from this BDD (that represents unsatisfiable formulas) needs to have at least one satisfying model and is, therefore, a satisfiable formula.

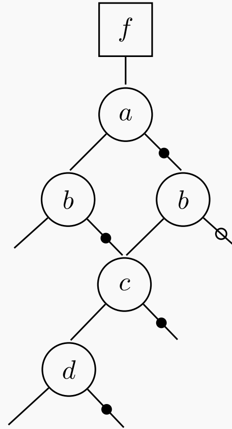
Evaluating a Model using a BDD

Due to optimization 1, we only have a single terminal node representing the truth value *true*. In addition, we have introduced complemented edges, which represent a *flip* in the resulting truth value. We therefore need to count the number of flips when evaluating a formula f under model $\mathcal{M}$.

Negation of a BDD. From this, we can deduce that for any formula for which the BDD has been computed, its negation can be computed in *constant time*. In

Exercise 3.3

Consider the following ROBDD:



Check whether the models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ are satisfying models or falsifying models.

- $\mathcal{M}_1 : \ a = \top, b = \bot, c = \top, d = \top$
- $\mathcal{M}_2 : \ a = \bot, b = \top, c = \bot, d = \bot$
- $\mathcal{M}_3 : \ a = \top, b = \top, c = \bot, d = \bot$

Solution. Consider $\mathcal{M}_1$: We traverse the ROBDD and count the number of complemented edges. There is a single complemented edge on the else-edge of the left node represeting b, therefore $\mathcal{M}_1 \not\models f$. Both $\mathcal{M}_2$ and $\mathcal{M}_3$ are satisfying models, since for both evaluations we encouter an even amount of complemented edges, 2 and 0, respectively.

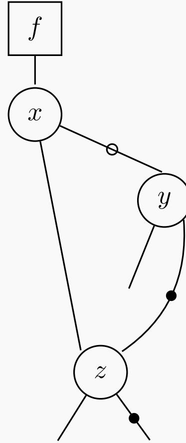
order to negate a BDD, we simply negate the edge connecting the function node and the node on the first level.

Constructing Formulas from BDDs

In order to compute DNF(f) from a BDD representing f, we follow the same steps as discussed in Section 3.2. We again search for all paths such that the formula evaluates to true, and compute DNF(f) from the resulting conjunctions.

Exercise 3.4

Consider the following BDD:



Compute the formula represented by the BDD.

Solution. The first path follows the else-edge from x and the then-edge from y. This path has no negations and therefore represents a satisfying assignment. Subsequently, we find two different paths that feature an even amount of complemented edges, and the resulting DNF is:

$$\text{DNF}(f) := (\neg x \land y) \lor (\neg x \land \neg y \land \neg z) \lor (x \land z).$$

Variable Order

The variable order chosen to construct a BDD heavily influences the space needed to store the BDD in memory. Consider the following formula

$$(x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \land (x_3 \leftrightarrow y_3) \ldots (x_n \leftrightarrow y_n).$$

Using the variable order $x_1 < y_1 < x_2 < y_2 < \ldots <$ yields an BDD with $3n + 2$ nodes, whereas the order $x_1 < x_2 < \ldots < y_1 < y_2 < \ldots <$ results in an BDD with $3 \cdot 2^n - 1$ nodes.

The problem of finding an optimal variable order is an NP-complete problem [2] and is therefore being researched extensively.

3.3 Construction of Reduced Ordered BDDs

In the final section of this chapter, we are going to discuss an algorithm to construct BDDs from formulas. The optimization discussed so far shows us the characteristics of a BDD. Constructing a BDD from an already computed BDD is of no use since this would entail the need for an exponential amount of space to construct the BDD in the first place. We are therefore looking for a way to construct the BDD directly from the formula f.

There are different ways to construct BDDs, another interesting approach recursively builds the BBD from the subformulas of f. This is done by merging two BDDs, based on the logical operator that connects the respective subformulas in f. This approach is out of the scope of this lecture, but we refer the interested reader to [1].

The algorithm we are going to discuss for this lecture is based on *cofactors*.

Definition 3.5 (Cofactor.) A *cofactor* of a formula f for a given assignment A is a formula $f_A = f[x_1 \leftarrow \mathbf{F}/\mathbf{T}, x_2 \leftarrow \mathbf{F}/\mathbf{T}, \ldots]$ that is computed from f by substituting the variables in f by their respective assignment in A. We call $f_x = f[x \leftarrow \mathbf{T}]$ the *positive* cofactor of f w.r.t. x, and define the *negative* cofactor $f_{\neg x}$ likewise.

One can view a cofactor as the formula being evaluated under a, potentially, partial assignment.

Example. We compute the positive and the negative cofactor w.r.t. the variable x for the following formula:

$$f = (x \wedge y) \vee (\neg x \wedge z).$$

Setting x to true results in the positive cofactor

$$f_x = (\top \wedge y) \vee (\bot \wedge z) = y,$$

and setting x to false gives us the negative cofactor

$$f_{\neg x} = (\bot \wedge y) \vee (\top \wedge z) = z.$$

The Algorithm to Construct a reduced and ordered BDD

Step 1) Compute all cofactors.
In the first step, we recursively compute all cofactors with respect to the given variable order. Given the order $a < b < c < d <$, we first compute f_a with $f_a = f[a \leftarrow \mathbf{T}]$, followed by f_{ab}, where $f_{ab} = f_a[b \leftarrow \mathbf{T}]$, etc.
If a cofactor matches a cofactor or the negation of a cofactor that we have already seen, we take note of this equality and backtrack. This takes care of the optimizations that we have introduced.

Step 2) Draw the BDD from the cofactors.

The goal is to draw a BDD such that each node represents a cofactor. The root node of the BDD represents the entire formula f. We therefore connect it with the function node representing f. The internal node connected via the *then-edge* represents the *positive cofactor* f_a. The internal node connected via the *else-edge* represents the *negative cofactor* $f_{\neg a}$.

We construct the BDD such that each node representing a cofactor leads us to the next one in the recursive computation. If a cofactor resolves to true, we draw a dangling edge. If a cofactor resolves to false, we draw a complemented dangling edge. If a cofactor is equivalent to a cofactor we have already seen, we connect the outgoing with the node that represents this cofactor. If a cofactor is equivalent to a negation of a cofactor we have already seen, we connect the outgoing with the node that represents this cofactor and complement the edge.

Step 3) Shift negations upwards

In the final step, we follow the convention that only else-edges may be complemented, in order for the BDD to be canonical. The execution of Step 2 might cause complemented dangling then-edges, as any positive cofactor may evaluate to **F**.

In order to remove the complement from a then-edge, we shift the complement upwards by distributing it to all edges connected to the source node of the then-edge. Note that this does not change the formula, as for any path the amount of complemented edges is not changed by this operation.

Edges that are complemented twice in this process do not need to be complemented. As this process might in turn complement an adjacent then-edge, we repeat this process.

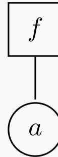
Exercise 3.5

Consider the following formula $f = (a \wedge b \vee \neg a) \wedge \neg c \wedge d \vee c$ and the variable order $a < b < c < d$. Construct the BDD that represents f.

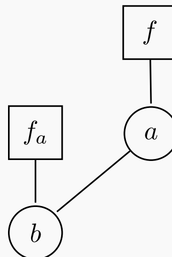
Solution.
We start by recursively computing all the cofactors and taking note of any equivalent cofactors.

$$
\begin{aligned}
f &= (a \wedge b \vee \neg a) \wedge \neg c \wedge d \vee c \\
f_a &= b \wedge \neg c \wedge d \vee c \\
f_{ab} &= \neg c \wedge d \vee c \\
f_{abc} &= \top \\
f_{ab\neg c} &= d \\
f_{ab\neg cd} &= \top \\
f_{ab\neg c\neg d} &= \bot \\
f_{a\neg b} &= c \\
f_{a\neg bc} &= \top \\
f_{a\neg b\neg c} &= \bot \\
f_{\neg a} &= \neg c \wedge d \vee c = f_{ab}
\end{aligned}
$$

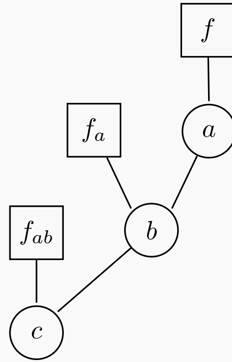
We can now continue to construct the BDD from the computed cofactors. We start by introducing the function node f and connect the first-level node representing a.



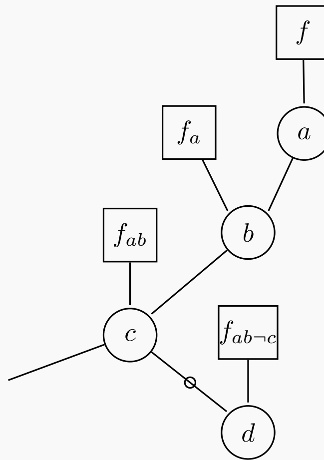
The positive cofactor of a does not evaluate to a truth value yet, we therefore add the node representing the next variable b. Note that we also annotate the cofactor with a function node accordingly.



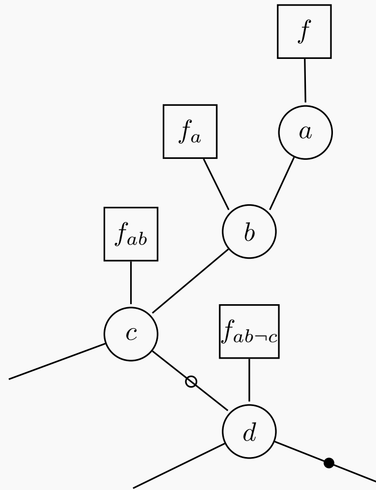
Again, the positive cofactor of b does not evaluate to a truth value and we add a node for the next variable c.



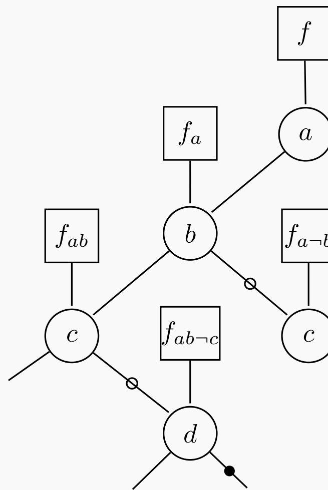
The cofactor f_{abc} evaluates to true and we therefore finish this branch by adding a dangling then-edge. The cofactor $f_{ab\neg c}$ does not evaluate to a truth value and we continue this branch by adding a node representing $f_{ab\neg c}$.



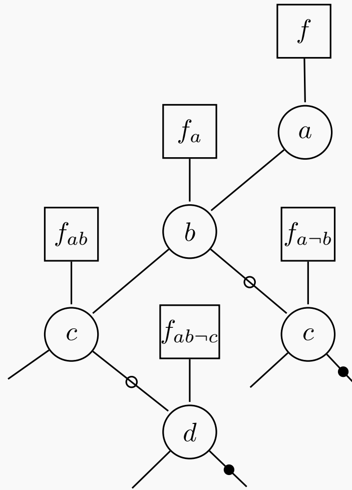
The cofactors for $f_{ab\neg cd}$ and $f_{ab\neg c\neg d}$ evaluate to **T** and **F** respectively. We therefore add two dangling edges and negate the else-edge.



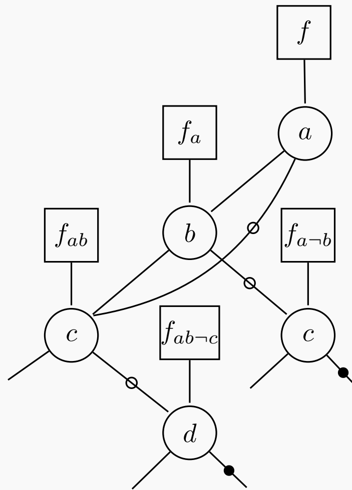
We backtrack and consider the cofactor $f_{a\neg b}$. The cofactor does not evaluate to a truth value or to any cofactors we have seen before. We therefore add another node representing this cofactor.



The cofactors $f_{a\neg bc}$ and $f_{a\neg b\neg c}$ evaluate to **T** and **F** respectively and draw the according dangling edges.



Finally, we consider the negative cofactor $f_{\neg a}$ which is equal to f_{ab}. Since this cofactor is already represented in the BDD we simply connect the else edge of the node a to the node representing f_{ab}.



As there are not complemented then-edges, we do not need to perform step 3 and are therefore done.

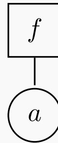
Exercise 3.6

Consider the following formula $f = (a \land \neg c) \lor \big(\neg a \land (b \lor (\neg b \land c))\big)$ and the variable order $a < b < c$. Construct the BDD that represents f.

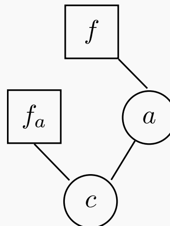
Solution. We start by recursively computing all the cofactors and taking note of any equivalent cofactors.

$$
\begin{aligned}
f &= (a \land \neg c) \lor \big(\neg a \land (b \lor (\neg b \land c))\big)\\
f_a &= \neg c\\
f_{ac} &= \bot\\
f_{a\neg c} &= \top\\
f_{\neg a} &= b \lor (\neg b \land c)\\
f_{\neg ab} &= \top\\
f_{\neg a\neg b} &= c = \neg f_a
\end{aligned}
$$

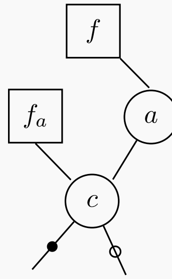
Note that $f_a = \neg c$. The variable b does not appear in f_a and assigning b either truth value does not have an effect in this branch. Therefore, we skip the cofactor of b and immediately compute the f_{ac} and $f_{a\neg c}$.
We can now continue to construct the BDD from the computed cofactors. We start by introducing the function node f and connect the first-level node representing a.



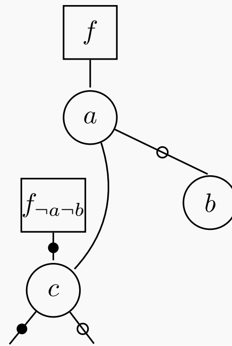
Since the positive cofactor f_a does not evaluate a truth value, we draw the node representing the next variable c.



The cofactors f_{ac} and $f_{a\neg c}$ evaluate to **F** and **T**. We draw the dangling edges and negate the then-edge.

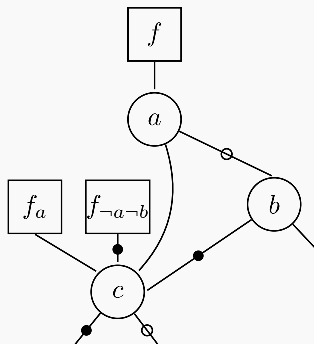


We continue with the negative cofactor $f_{\neg a}$ by drawing the node representing the next variable b.

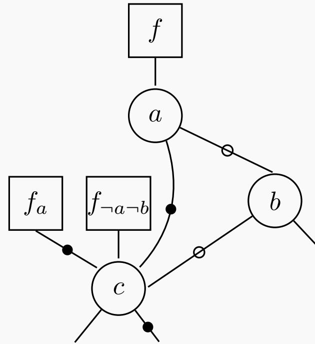


We have computed that $f_{\neg ab}$ evaluates to **T** and therefore add a dangling edge.

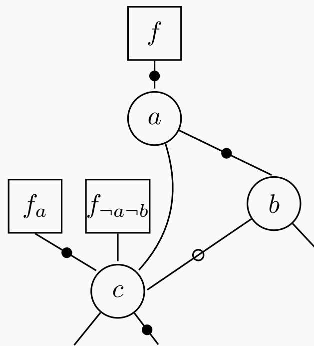
The cofactor $f_{\neg a \neg b}$ is equal to the negation of f_a, we, therefore, connect the else-edge of the node representing the cofactor $f_{\neg a}$ to the node representing f_a and complement the newly added edge. We also indicate that the node labeled with c represents two cofactors and negates the edge connecting the function node for cofactor $f_{\neg a \neg b}$.



This BDD has a complemented then-edge at the node representing c. Therefore, we distribute the complement to all edges of the c node and remove the complement from the then-edge.



By doing so we have created a negation at the then-edge outgoing from the node representing a. We repeat step 3 and distribute the complement to all adjacent edges.



There are no negated then-edges in this representation and we are therefore done.

List of Definitions

3.1 Directed Acyclic Graph 3
3.2 Binary Decision Diagram. 3
3.3 Disjunctive Normal Form 4
3.4 Ordered BDDs 8
3.5 Cofactor. 13

Bibliography

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

[2] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Springer, 2016.