3 Representational Tradeoffs

In this chapter, we will review a variety of simple representations of properties in finite, propositional domains, and we will develop our understanding of the relative expressive power of these various kinds of representations. We first introduce some simple knowledge representations in Section 3.1, and then some increasingly powerful compact representations in Sections 3.2–3.3. We will refine the notion of "relative expressive power" to take into account the representation size in Section 3.4; we will justify identifying "efficiently expressible" and "efficiently computable," respectively, with "polynomially related size" and "polynomial running time" in Section 3.5. Our discussion will focus on how the choice of representation features an inherent trade-off between the expressive power or succinctness of the representation, and the computational complexity of working with the representations. Specifically, in Section 3.6, we review the theory of *NP-completeness*, which is at heart a notion of expressive power for computational problems (corresponding to the representations) that is at the same time widely identified with computational intractability. In the remainder of the book, we will frequently see that NP-completeness poses limits to the richness of the representations we can effectively work with.

3.1 Naïve representations: truth tables and databases.

To store knowledge or compute on it, we need to represent it somehow. At the most basic level, we need the following:

Definition 3.1 A *knowledge representation* over an alphabet of symbols Σ is a partial mapping from finite strings of these symbols (denoted Σ^*) to properties (subsets of the set of possible worlds X), i.e., that assigns at most one property to each string.

There are many dimensions along which we may compare different kinds of knowledge representations: how wide a variety of properties they can represent, how compact they are, how efficiently operations on them can be supported, and so on. In this chapter we will examine these different aspects in more detail. But, for the moment, it will be helpful to have some simple, concrete examples of knowledge representations.

We have defined properties in terms of the sets of possible worlds that possess the property. So, one natural representation is to simply list encodings of the possible worlds that possess the property:

in_room_1	in_room_2	in_hallway
0	0	1
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Table 3.1 An example database table representation for worlds on attributes `in_room_1`, `in_room_2`, and `in_hallway`, where `in_room_1` or `in_hallway` are true. The representation is given by concatenating the rows in order.

in_room_1	in_room_2	in_hallway	in_room_1 or in_hallway
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 3.2 An example truth table representation of the same property as in Table 3.1, given by reading the final column from top to bottom.

Example 3.1: If there are n Boolean attributes, then each possible world may naturally be represented by a string of length n, in which the ith symbol is 0 or 1 if, respectively, the ith attribute is false or true in that world. Then the property may be encoded by simply listing the possible worlds one after another in lexicographic order – so the first n symbols encode the first possible world, symbols $n+1$ through $2n$ encode the second, and so on. We will refer to this as the *database (table) representation*.

Another simple representation arises by viewing the property as a Boolean function, that indicates whether or not an input possible world satisfies the property:

Example 3.2: If the possible worlds are represented by n Boolean attributes, then we may represent the property by a Boolean string of length 2^n in which the ith symbol is a 1 if the ith possible world (in lexicographic order) has the property, and otherwise the ith symbol is a 0. We will refer to this as the *truth table representation*.

We can start to compare these representations to each other in various respects, such as their length. The truth table representation always has length 2^n, whereas if there are N possible worlds satisfying the property, the database representation has length $N \times n$. If only a small number of possible worlds satisfy the property, then the database representation is a good choice. In general, however, N may be as large as 2^n, in which case the database representation would be much worse than the truth

table representation—indeed, n times larger. But, in such a case, neither of these representations is actually very practical: note that for even moderate values of n, say $n = 100$, both representations have sizes in the billions, and both quickly grow to be astronomically large.

We will next examine some representations that are small for a wider variety of natural properties. But, database tables and truth tables are still useful representations for conceptual purposes: they are relatively simple representations that capture all possible functions.

Aside: Although truth tables are large, requiring 2^n bits to represent functions on n bits, representations of *probability distributions* are generally larger. Representing arbitrary probability distributions on our n-bit possible worlds requires specifying *probability* values, not just Boolean values, for each of the possible worlds. Each of these are often smaller than $1/2^n$, and hence require n bits or more to represent. Probabilistic modeling approaches to learning and inference define families of probability distributions that have more compact representations, but these are much more challenging to learn, both statistically and computationally.

3.2 Formulas: conjunctions and disjunctions.

The most commonly used scheme for representing properties is Boolean formulas. Specifically, the family of formulas using De Morgan connectives:

Definition 3.2 *Boolean (De Morgan) formulas* are defined using a set of attributes, that we'll denote here by $X_1, \ldots, X_n$, and the *connectives* $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT), with brackets or parentheses "[ ]" and "( )" to indicate scopes as follows:

- A single attribute X_i is a De Morgan formula.
- The negation $\neg\varphi$ of a De Morgan formula φ is a De Morgan formula.
- The AND $[\varphi \wedge \psi]$ of two De Morgan formulas φ and ψ is a De Morgan formula.
- The OR $[\varphi \vee \psi]$ of two De Morgan formulas φ and ψ is a De Morgan formula.

A formula defines a property as follows. Given a possible world, assigning Boolean values to the attributes, the property contains that possible world if and only if the formula *evaluates* to 1 (true). Evaluation is determined recursively, based on the structure of the formula as follows:

- The value of a formula consisting of a single attribute X_i is the truth value of that attribute.
- The value of a formula $\neg\varphi$ is the negation of the value of φ: if φ evaluates to 1, $\neg\varphi$ evaluates to 0 and vice-versa.
- The value of a formula $[\varphi \wedge \psi]$ is 1 if both φ and ψ evaluate to 1. Otherwise, it evaluates to 0.
- The value of a formula $[\varphi \vee \psi]$ is 1 if either φ or ψ evaluate to 1. Otherwise, if both evaluate to 0, it also evaluates to 0.

There exist connectives that are not considered in De Morgan formulas. For example, *implication*, commonly denoted with $\supset$, is such a connective. The formula

$\varphi \supset \psi$ is taken to mean "if φ is true, then ψ must also be true" or in other words, the formula is false if φ is true and ψ is false, and otherwise it is true. This is surely a convenient connective, but it is actually not needed: the formula $[(\neg\varphi) \vee \psi]$ is easily seen to be equivalent. Likewise, the *exclusive-or (XOR)* connective $\varphi \oplus \psi$, which is true if exactly one of φ and ψ is true (i.e., not both), can be expressed as the De Morgan formula $[\varphi \wedge (\neg\psi)] \vee [(\neg\varphi) \wedge \psi]$. Indeed, we will see shortly that De Morgan formulas can express all properties, so in a sense we do not need to include these connectives.

We are usually interested in more limited families of formulas, for example *conjunctions* and *disjunctions*:

Definition 3.3 *Conjunctions* are the following family of Boolean formulas:

- An attribute X_i or the negation of an attribute $\neg X_i$ is a conjunction; these are also called *literals*.
- The AND $[\varphi \wedge \psi]$ of two conjunctions φ and ψ is a conjunction.

We refer to the number of attributes in the conjunction as the *width*.

Definition 3.4 *Disjunctions* (also frequently referred to as *clauses*) are the following family of Boolean formulas:

- A literal is a disjunction.
- The OR $[\varphi \vee \psi]$ of two disjunctions φ and ψ is a disjunction.

We refer to the number of attributes in the disjunction as the *width*.

We frequently omit the parentheses/brackets when writing conjunctions and disjunctions because the nesting does not matter: a conjunction is false if any of the constituent literals is false, and otherwise it is true. Similarly, a disjunction is true if any of its literals is true, and otherwise it is false.

Conjunctions and disjunctions are related by negation as follows:

Proposition 3.5 (De Morgan's Law) *For any formulas φ and ψ, $\neg[\varphi \wedge \psi] = [(\neg\varphi) \vee (\neg\psi)]$ and $\neg[\varphi \vee \psi] = [(\neg\varphi) \wedge (\neg\psi)]$.*

More generally, De Morgan's Law gives that the negation of a conjunction is the disjunction obtained by negating each of the constituent literals, and vice-versa.

An important consequence of De Morgan's Law is that many "if-then" rules can be written as disjunctions.

Example 3.3: Let's return to the setting of the box-moving robot from Sections 2.1 and 2.4. A *plan* for the robot is a sequence of *actions*. In this simple setting, the actions would be "move from hallway to room 1", "move from hallway to room 2", "return to hallway from room 1", "return to hallway from room 2", "pick up box", and "put down box". We can add attributes indicating which of the actions occurred at each step across time, in addition to the attributes indicating the states of each fluent. The states of the environment before and after each of the actions must satisfy certain conditions that are given by such rules: in order for the robot to leave the

hallway, it must have been in the hallway to begin with, and afterwards it must be in the indicated room. So, for each tth step, we have that

- "move from hallway to room 1"$_t$ $\supset$ "the robot is in the hallway"$_t$,
- "move from hallway to room 1"$_t$ $\supset$ "the robot is in room 1"$_{t+1}$, and
- "move from hallway to room 1"$_t$ $\supset$ $\neg$"the robot is in the hallway"$_{t+1}$.

As we noted earlier, the $\supset$ connective can be rewritten as an OR. Now, if the robot is holding the box, the box should also move; this is called a "conditional effect" of the action:

- ["move from hallway to room 1"$_t$ $\wedge$ "the robot is holding the box"$_t$] $\supset$ "the box is in room 1"$_{t+1}$, and
- ["move from hallway to room 1"$_t$ $\wedge$ "the robot is holding the box"$_t$] $\supset$ $\neg$"the box is in the hallway"$_{t+1}$.

Now, due to De Morgan's Law, these can also be rewritten as disjunctions/clauses:

- $\neg$"move from hallway to room 1"$_t$ $\vee$ $\neg$"the robot is holding the box"$_t$ $\vee$ "the box is in room 1"$_{t+1}$, and
- $\neg$"move from hallway to room 1"$_t$ $\vee$ $\neg$"the robot is holding the box"$_t$ $\vee$ $\neg$"the box is in the hallway"$_{t+1}$.

Similar rules can be written to characterize all of the necessary preconditions and effects of all of the actions in this domain. These rules almost fully characterize the dynamics of the domain. The only missing piece is that we need a rule that says that the attributes do not change unless an action changes them; these are called "frame axioms." For example, if the robot is in the hallway and the robot does not move to room 1 or room 2, the robot should still be in the hallway:

- ["the robot is in the hallway"$_t$ $\wedge$ $\neg$"move from hallway to room 1"$_t$ $\wedge$ $\neg$"move from hallway to room 2"$_t$] $\supset$"the robot is in the hallway"$_{t+1}$

which again may be rewritten using De Morgan's Law as the disjunction

- $\neg$"the robot is in the hallway"$_t$ $\vee$ "move from hallway to room 1"$_t$ $\vee$ "move from hallway to room 2"$_t$ $\vee$ "the robot is in the hallway"$_{t+1}$.

So even the seemingly-simple class of disjunctions can express each of the individual rules governing the dynamics of this environment.

3.3 Decision trees versus conjunctions and disjunctions: absolute expressive power.

Intuitively, it still appears that conjunctions and disjunctions are weak representations. We can establish this as a fact by showing that there are simple things they cannot express: indeed, they cannot express each other.

Theorem 3.6 *There is no conjunction representing* $[X_1 \vee X_2]$*, and no disjunction representing* $[X_1 \wedge X_2]$.

Proof Observe that $[X_1 \vee X_2]$ is true if X_2 is true, whether X_1 is true or false. If a conjunction included X_1 or $\neg X_1$, it would be false for one of the two settings of X_1. So no conjunction containing X_1 can represent $[X_1 \vee X_2]$; but now, the conjunction must give the same truth value, independent of the setting of X_1. In particular, if X_2 is false, it must give the same truth value for both values of X_1. But in this case, $[X_1 \vee X_2]$ takes different truth values, so the conjunction cannot represent $[X_1 \vee X_2]$.

We analogously find that $[X_1 \wedge X_2]$ is false if X_2 is false, whereas a disjunction that includes X_1 is true if X_1 is true, and one that includes $\neg X_1$ is true if X_1 is false. Therefore, the disjunction cannot include X_1 or $\neg X_1$, but now if X_2 is true, the value of $[X_1 \wedge X_2]$ is the same as the value of X_1, but the disjunction cannot depend on X_1. So no disjunction representing $[X_1 \wedge X_2]$ exists. □

Of course, both conjunctions and disjunctions are examples of De Morgan formulas, so the family of all formulas includes both kinds of representations, in addition to many others. Indeed, the family of *all* De Morgan formulas includes formulas that are quite complicated (see Exercise 3.8).

There exist other families of representations that cannot be as complicated as arbitrary De Morgan formulas, but capture both conjunctions and disjunctions. An important example, that does not have the form of a Boolean formula, is the family of *decision trees*.

Definition 3.7 Recall that a *graph* is given by a set of *nodes* (also called *vertices*) and *edges*, which are pairs of nodes. If the pairs are ordered, we say that the edges are *directed*, and otherwise (if the pairs are simply sets), the edges are *undirected*. We say that the edge is *outgoing* from the first node in the ordered pair, and *incoming* to the second node.

A *path* in a graph is a sequence of nodes in the graph such that each pair of consecutive nodes is included in an edge of the graph (in the given order, if the edges are directed). A graph is *connnected* if there is a path between every pair of nodes when the edges are interpreted as undirected.

A *cycle* is a path in which the first and last nodes are identical. If each other vertex appears at most once in the cycle, we say that the cycle is *simple*. We refer to directed graphs with no cycles as *Directed Acyclic Graphs (DAGs)*. The nodes of a DAG with no incoming edges are called *sources* and the nodes with no outgoing edges are called *sinks*.

A *tree* is a connected graph that has no simple cycles when the edges are interpreted as undirected. A *rooted* tree is a tree in a directed graph, in which there is a source node, called the *root*, that can reach each other node by a path. In a rooted tree, the sinks (i.e., that are only endpoints of paths) are called *leaves*. A *binary* tree is one in which every node except for the leaves have two outgoing edges.

Definition 3.8 *Decision trees* are binary rooted trees in which the leaves are labeled with truth values, the other nodes are labeled with attributes, and the edges are labeled

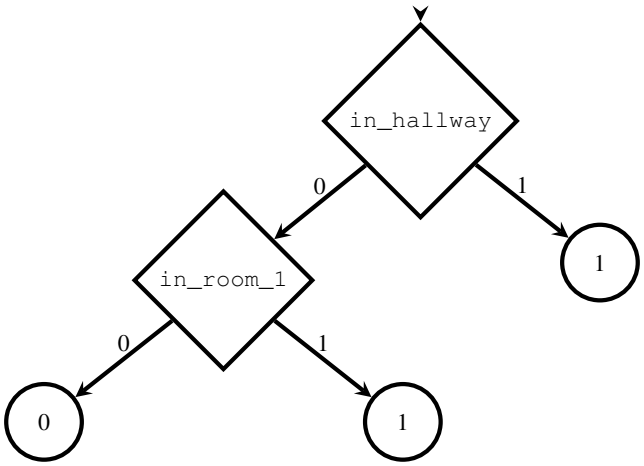


Figure 3.1 Example of a decision tree representing the same property as in Table 3.1 and Table 3.2.

with truth values such that each node that is not a leaf has one outgoing edge labeled true and one labeled false.

A decision tree represents the property consisting of the possible worlds such that there is a path from the root to a true leaf for which for each attribute labeling a node on the path, the subsequent edge is labeled with the truth value of the attribute in that possible world. Note that since each node has exactly one edge with a consistent truth value, this path is uniquely determined.

Proposition 3.9 *Without loss of generality, each attribute appears at most once on any path from root to leaf in the decision tree.*

Proof Observe that if an attribute appears twice on any path from the root to a leaf, the value of the attribute was already determined by the branch taken the first time the attribute was encountered. Therefore, we can omit the second node, and replace it with the subsequent node that is reached by the edge with the same truth value as the one on the edge taken previously on the path. □

Theorem 3.10 *For every conjunction there is a decision tree representing that conjunction. Similarly, every disjunction may be represented by a decision tree. If the conjunction or disjunction contains k literals, then in particular there is a decision tree with $2k + 1$ nodes.*

Proof Let any conjunction $\ell_1 \wedge \cdots \wedge \ell_k$ be given. We consider a decision tree with $k + 1$ leaves and k other nodes, which are labeled by the attributes $X_{i_1}, \ldots, X_{i_k}$ in the literals $\ell_1, \ldots, \ell_k$. The node labeled X_{i_1} is the root. For each $j < k$, the node labeled X_{i_j} has one edge to a false leaf and one edge to $X_{i_{j+1}}$; if $\ell_j = X_{i_j}$, the edge to the false leaf is labeled false and the edge to $X_{i_{j+1}}$ is labeled true. Otherwise, the edge to the false leaf is labeled true and the other edge is labeled false. Finally, the node labeled X_{i_k} has

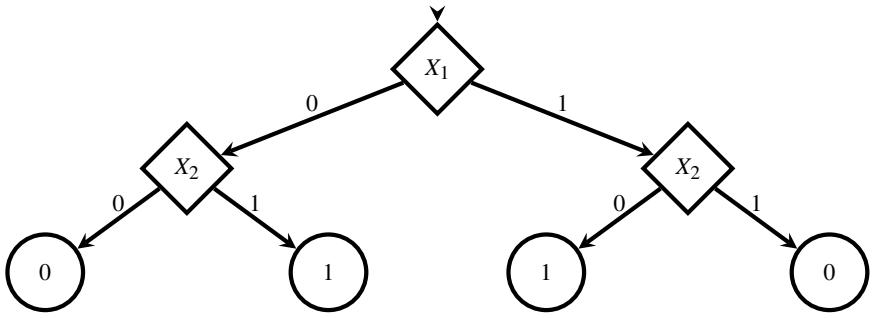


Figure 3.2 Decision tree for $[X_1 \oplus X_2]$.

edges to leaves labeled true and false; if $\ell_k = X_{i_k}$, then the edge to the false leaf is labeled false and the edge to the true leaf is labeled true, and otherwise (if $\ell_k = \neg X_{i_k}$) the labels are reversed.

To see that this represents the conjunction, observe that the worlds reaching the single true leaf must have set each ℓ_j to true for $j = 1, \ldots, k$, and therefore are in the property defined by the conjunction. Vice-versa, if the world is in the conjunction property, each ℓ_j is set to true, and we observe that for each node in the tree, the path must take the edge to $X_{i_{j+1}}$ for $j < k$, and from X_{i_k} to the final true leaf, so the world is in the property defined by the decision tree.

Analogously, given a disjunction $\ell_1 \vee \cdots \vee \ell_k$, we have a decision tree with k true leaves and a single false leaf, with the other k nodes labeled by $X_{i_1}, \ldots, X_{i_k}$ for the respective attributes in the k literals. X_{i_1} is again the root, and for $j < k$, X_{i_j} has the true edge to a true leaf and the false edge to $X_{i_{j+1}}$ if $\ell_j = X_{i_j}$, and otherwise vice-versa. X_{i_k} has its true edge to a true leaf and its false edge to a false leaf if $\ell_k = X_{i_k}$, and otherwise vice-versa. Now, the only worlds *not* in the property are those that reach the false leaf, which are those for which $\ell_1, \ldots, \ell_k$ are all false, and hence not included in the property defined by the disjunction. Conversely, if a world is not in the property defined by the disjunction, it must reach the false leaf and hence is not in the property defined by the decision tree. □

In fact, we observe that decision trees are *strictly* more expressive than conjunctions and disjunctions put together: the exclusive-or function cannot be represented by either conjunctions or disjunctions, but it is representable using a decision tree:

Theorem 3.11 *The property* $[X_1 \oplus X_2]$ *can be represented by a decision tree, but not by a conjunction or disjunction.*

Proof The decision tree is given in Figure 3.2. We observe that it indeed represents $[X_1 \oplus X_2]$ as needed.

To see that conjunctions cannot compute $[X_1 \oplus X_2]$, suppose for contradiction that such a representation exists. We first observe that the property is true for both $(0, 1)$ and $(1, 0)$. Therefore, since the inclusion of X_1 or $\neg X_1$ would make the conjunction false in one or the other of these possible worlds, neither literal can appear in the conjunction. But then, the property could not depend on the setting of X_1 and hence either both

$(0,0)$ and $(1,0)$ would have the property or neither would; since $(0,1)$ evaluates to true and $(0,0)$ evaluates to false, neither option can be correct, so indeed, the conjunction cannot represent $[X_1 \oplus X_2]$.

The argument for disjunctions is similar: the property is false for both $(0,0)$ and $(1,1)$, so again neither X_1 nor $\neg X_1$ can appear in any hypothetical disjunctive representation. But then the value cannot depend on X_1, and so the disjunction would give the same value to $(0,0)$ and $(1,0)$, which would be incorrect. □

3.4 DNF versus decision trees: complexity and expressive power.

So far, we have compared representations based on the variety of properties they can represent. We have seen that some representations, such as truth tables and databases, can represent all possible properties, whereas others, such as conjunctions and disjunctions, cannot. In this section, we'll develop a more refined view of expressive power in terms of the lengths of the representations, as opposed to simply whether or not a property can be represented at all.

Decision trees are actually another representation that can represent all possible properties, just like truth tables and database tables.

Theorem 3.12 *Every property on n attributes can be represented by a decision tree of size* $2^{n+1} - 1$.

Proof By induction on n. We take as the base case $n = 0$, which is only the constant properties, true or false; these can be represented by a single leaf, which is a decision tree of size 1.

So, supposing the theorem is proved for properties of $n - 1$ attributes, consider any property $c(X_1, \ldots, X_n)$ on n attributes. Let $c|_{X_n=0}(X_1, \ldots, X_{n-1}) = c(X_1, \ldots, X_{n-1}, 0)$ and $c|_{X_n=1}(X_1, \ldots, X_{n-1}) = c(X_1, \ldots, X_{n-1}, 1)$. By the induction hypothesis, $c|_{X_n=0}$ and $c|_{X_n=1}$, being properties defined on $n - 1$ attributes, can each be represented by decision trees of size $2^n - 1$. So, consider the following decision tree: the root is a node labeled with X_n, with the true edge directed to the tree for $c|_{X_n=1}$ and the false edge directed to the tree for $c|_{X_n=0}$. This is a decision tree of size $1 + 2^n - 1 + 2^n - 1 = 2^{n+1} - 1$. We observe that for any possible world $(x_1, \ldots, x_n)$, if $x_n = 1$, by hypothesis the tree gives the truth value of $c|_{X_n=1}(x_1, \ldots, x_{n-1}) = c(x_1, \ldots, x_{n-1}, 1)$, and similarly if $x_n = 0$, it gives the truth value of $c|_{X_n=0}(x_1, \ldots, x_{n-1}) = c(x_1, \ldots, x_{n-1}, 0)$. □

Unpacking the construction in the proof, we see that we obtain a complete binary tree of height n with a unique leaf for each possible world: the leaves simply give a truth table for the property. Again, the tree is therefore impractically large except for very small domains. Nevertheless, a tree representation can be given in principle.

Another important family of representations that, like decision trees, captures both conjunctions and disjunctions – and indeed, again, all possible properties – is the family of Disjunctive Normal Form (DNF) formulas:

Definition 3.13 *Disjunctive Normal Form (DNF) formulas* are the following family of Boolean formulas:

- A conjunction is a DNF (called a *term*).
- The OR $[\varphi \vee \psi]$ of two DNF formulas is a DNF.

I.e., a DNF is an OR of ANDs of literals, where we refer to the ANDs of literals as the *terms* of the DNF.

Theorem 3.14 *For every property c on n attributes, there is a DNF representation for c with a number of terms of width n equal to the number of possible worlds in c.*

Proof Consider each possible world $(x_1, \ldots, x_n)$ in c, and consider the following conjunction: if $x_i = 1$, the literal X_i appears in the conjunction, and otherwise ($x_i = 0$) $\neg X_i$ appears in the conjunction. Observe that this conjunction has width n and the only possible world for which this conjunction is true is precisely $(x_1, \ldots, x_n)$. Now, consider the DNF that has terms equal to these conjunctions. If the DNF is true, one of the terms must be true, and hence the possible world must be one of the possible worlds in c. Conversely, for each of the possible worlds in c, the corresponding term is true, and hence the DNF is true. Therefore the DNF indeed represents c. □

Just as the decision trees we constructed for arbitrary properties had a close connection to the truth table representation for a property, observe that the DNF formulas we constructed are very closely related to the database representation, and are only slightly larger. Unfortunately, this also means that when the database representation is very large, the DNF formula we construct will also be very large. It is still therefore not practical for such properties.

Taking stock, we have thus seen four different families of representations that can represent all properties. Naïvely, it may appear that the four representations are therefore equivalent in expressive power, unlike the strictly weaker conjunctions and disjunctions.

Unfortunately, for all four of these families, we have used representations of size at least 2^n to represent properties of n attribute worlds, rendering them impractical. For truth tables and databases, it is clear that the representation of some or all properties is necessarily large. But for decision trees and DNFs, we have so far only presented one way of representing arbitrary properties. It is clear that sometimes a property can be represented by a decision tree or DNF that is much smaller than the ones constructed by Theorems 3.12 and 3.14—indeed, consider the property given by the set of all possible worlds. Both the truth table and database representations have size at least 2^n for this property. Yet, a decision tree with a root equal to a single "true" leaf represents this property, and the DNF $X_1 \vee \neg X_1$ is also a small representation of this property. Do such small representations exist for arbitrary properties? Unfortunately not, and indeed all representations must be inefficient for most properties.

Theorem 3.15 *For every representation for properties on n attributes using an alphabet Σ, for every k at least a $1 - 1/2^k$ fraction of all properties require representations of length at least $\frac{1}{\log_2 |\Sigma|}(2^n - k)$.*

Proof First, observe that there are 2^n possible worlds for our n attributes, and there is a property for each subset of the possible worlds—hence, 2^{2^n} possible properties. There are $|\Sigma|^{s+1}$ strings of length at most s using the alphabet Σ. Since a knowledge representation can represent at most one property with each string, in order to represent a $1/2^k$ fraction of the possible properties, we need

$$|\Sigma|^{s+1} \geq 2^{2^n-k}$$

and up to length $s = \frac{1}{\log_2 |\Sigma|}(2^n - k) - 1$, we see that we have only 2^{2^n-k} possible strings. Therefore, the remaining $(1 - 1/2^k)2^{2^n}$ properties must use strings of length at least $\frac{1}{\log_2 |\Sigma|}(2^n - k)$. □

We therefore see that *no* knowledge representation scheme can give practically useful representations of anything but a vanishingly small fraction of the possible properties. In other words: the choice of knowledge representation scheme necessarily imposes a kind of "bias" towards some small family of properties, that are allocated the short representations.

We have already seen an example of a simple property, the set of all possible worlds, that does not have a small database representation (or truth table representation) but does have small decision tree and DNF representations. But by contrast, whenever the database representation is small, Theorem 3.14 shows that there exists a DNF representation of essentially the same size. So, for all properties, DNF formulas are essentially always preferable to databases, even though databases sometimes give compact representations of some properties. In this quantitative sense now, we see that DNFs are more expressive than databases.

A similar relationship holds between DNFs and decision trees: whenever a small decision tree exists, a reasonably small DNF exists as well.

Theorem 3.16 *If a property on n attributes is represented by a decision tree with s nodes, it may also be represented by a DNF with s terms of size* $\min\{s, n\}$.

Proof We note that the decision tree consists of the set of possible worlds that reach a true leaf via a path from the root. Since the decision tree has s nodes in total, it has at most s leaves. Consider, for each true leaf, the path that reaches the leaf. Again, since the decision tree has s nodes in total, that path is labeled by at most s attributes; likewise since by Proposition 3.9, the attributes labeling a path may be assumed to be distinct, there are also at most n nodes on the path. Consider the following conjunction for the path: if the path takes the true edge from a node labeled X_i, we include X_i in the conjunction; if it takes the false edge, we include $\neg X_i$; and if it does not visit a node labeled with X_i then there is no literal with X_i in the conjunction. Now we consider the DNF obtained by the OR of such conjunctions.

We observe that on the one hand, if the decision tree labels a possible world true, it must reach one of the true leaves, by the unique path in the tree. By construction, a possible world reaching the true leaf will make the conjunction we constructed for that path true. Therefore, our DNF is also true for such worlds. Conversely, if the DNF is true, one of the conjunctions must have been satisfied. Since in satisfying the

conjunction ensures that the possible world will take the edges on the path that lead to the corresponding true leaf, we see that this world will be included in the decision tree's property as well. Therefore, the DNF indeed expresses the same property as the decision tree. □

Again, by contrast, Exercise 3.2 establishes that there are properties with compact DNF representations that do not have compact decision tree representations. Thus, DNF formulas are considered to be more expressive than decision trees.

3.5 Polynomial sizes and "efficient" algorithms—efficient evaluatibility.

The DNF formulas constructed by Theorem 3.16 is only guaranteed to have total size at most roughly s^2 when the decision tree has size s. That is, the DNF may indeed be larger than the decision tree—yet, we suggested we were content to view this as "reasonable," whereas an exponential increase in size is not reasonable, and this was the reason we claimed DNFs to be strictly more expressive than decision trees. In this section we'll elaborate on why we are willing to tolerate, in general, polynomial increases in the representation size (and only polynomially bounded increases) as far as expressiveness is concerned.

We have been a bit cavalier about the precise encoding of our more complex representations, such as decision trees or Boolean formulas, as strings. For example: how do we encode the attributes? Do we suppose that we have a symbol for each attribute? Or, is there an "attribute" symbol with some encoding of the index of the attribute? And, how is this index encoded – do we again have a separate symbol for each index, is it a binary encoding, or is it something entirely different? Do we view conjunctions as encoded as simply a list of the constituent literals, or do we require writing the AND connective (with parentheses)? And so on. The different choices may yield encodings that differ significantly in their total size. In particular, if we regard attributes as needing to be indexed in binary, representing n different outcomes requires strings of length up to $\log_2 n - 1$. This may introduce an overhead in the encoding that grows with n, in contrast to if the alphabet simply has a symbol for each attribute. In practice, we do have to fix such details when we make use of these representations on a computer.

Indeed, the native representations used by the computer's instruction set may make some choices more natural than others. In particular, computers often operate on strings of fixed lengths greater than a single symbol, and it is natural to regard these as the alphabet. Unfortunately the lengths change across architectures – this is the distinction between "8-bit," "32-bit," "64-bit," and so on. It is true that asymptotically, even a 64-bit integer cannot take more than 2^{64} values, and hence properly we would still in general need to use $\log_{2^{64}} n$ such values to represent an index n. On the other hand, it is rare that we will actually seek to represent an index that needs more than at most, say, a couple of 64-bit values to represent: even 2^{64} is an astronomically large number. So

in practice at present it is natural to regard the index as being represented by a single 64-bit symbol.

Nevertheless, we abstain from committing to such a choice of representation. We do not want to develop a theory that is tied to the particular computing architectures in use at a particular point in time. Indeed, vastly different architectures are already in use at present (mobile vs. server vs. "cloud") and new ones continue to be introduced. There would be little value to developing a theory around the considerations of architectures that rapidly enter obsolescence. We desire a theory that captures the problems faced by large corporations as much as by individuals with consumer-grade hardware. We therefore seek to abstract away such details.

At the same time, a benefit of this decision is that the actual details of how the knowledge representation is encoded in a string on a particular computing architecture is often tedious and overwhelming. It is easy to get lost in the details and lose sight of the larger-scale properties of that representation. Indeed, the kind of theorems we have stated already do not depend on the particular details of precisely how the formulas or decision trees are represented as strings. If we only reasoned in terms of the actual representation size in symbols, the big picture would be obscured in the details of complex expressions.

The same considerations apply even more strongly when we start to discuss algorithms for operating on our representations. There are many different languages for programming, and different computer architectures execute various operations more or less quickly. The actual running time may not even be the same when the algorithm is executed twice on the same input using the same computer. It is essentially hopeless to talk about the exact running time of an algorithm in practice. But, there are limits to how much the running times may vary; across all reasonable, currently viable architectures[1] the variation can be bounded by a polynomial function of the running time and input size: specifically, one architecture can *simulate* another (and programs can be translated from one language to another) with at most a polynomial overhead. We will not go into the details of these simulations here. In general, any such polynomial overhead may be required between some pair of architectures, but not more. This claim about the equivalence of running time across architectures is known as the *Extended Church-Turing Thesis*.

In the case of algorithms, we define *efficient* algorithms to be those whose running time (on some arbitrary architecture) can be bounded by a polynomial function of the length of the input, represented as a string; this proposal is known as the *Cobham-Edmonds Thesis*. Observe that we assert a corresponding willingness to ignore polynomial increases in both the running time and the encoding length of the input representation.

There are two additional justifications for our focus on polynomial running time as our definition of efficient computation. The first is mathematical convenience, in that we want a notion that allows the use of other, arbitrary "efficient" algorithms as

[1] At the time of this writing, quantum computing has not yet reached a scale that we would regard as "viable," but quantum computing would be the one exception to this claim.

subroutines in the design of an "efficient" algorithm. Cobham proved that any family of algorithms

- that includes all algorithms that run in linear time, and
- that is closed under such use of subroutines as basic operations

must contain all polynomial-time algorithms. Thus the family of polynomial-time algorithms is the most conservative choice of "efficient" algorithms that permits us to assume that we can use subroutines freely, while also regarding linear-time as "efficient." Actually, we will make extensive use of these properties, especially in the next section.

The other justification is an observation by Edmonds, that a polynomial-time algorithm cannot use "brute force" enumerate-and-check methods, but rather must identify and utilize some kind of special property of the task. In practice, refined use of this property that enables the design of a polynomial-time algorithm has almost always led to algorithms that are truly efficient in practice.

Following standard practice in the Theory of Computation, we will focus on asymptotic running time bounds that are parameterized in terms of the length of the input:

Definition 3.17 Recall that we say that a function $f : \mathbb{N} \to \mathbb{N}$ is $O(g(n))$ for a monotone bound function $g : \mathbb{N} \to \mathbb{N}$ if there exists $C, N \geq 0$ such that for all lengths $n \geq N$, $f(n) \leq C \cdot g(n)$. We say that f is $o(g(n))$ if for all constants $\epsilon > 0$ there exists $N_\epsilon \geq 0$ such that for $n \geq N_\epsilon$ $f(n) \leq \epsilon \cdot g(n)$. Similarly, we say that f is $\Omega(g(n)$ if there exists $C, N \geq 0$ such that for all $n \geq N$, $f(n) \geq C \cdot g(n)$, and f is $\omega(g(n))$ if for all $C \geq 0$ there exists $N_C \geq 0$ such that for $n \geq N_C$, $f(n) \geq C \cdot g(n)$.

Definition 3.18 The (worst-case) *running time bound* of a deterministic algorithm is a function $T : \mathbb{N} \to \mathbb{N}$ such that $T(n)$ is the maximum running time of inputs of length n; for randomized algorithms for which each input has a finite bound with probability 1, then the running time $T(n)$ is defined analogously as the supremum of the running times for each length n. We say that an algorithm runs in *polynomial time* if $T(n)$ is $O(n^k)$ for some integer k.

We care not only that our knowledge representations give compact representations of properties of interest, but also that we can efficiently solve some basic computational tasks involving them. For example, the most basic task is, given a possible world, does it satisfy the given property or not? This is the evaluation problem:

Definition 3.19 For a class of representations C, the *evaluation problem* for C is, given a pair (c, x) where $c \in C$ and x is a possible world, return true if $x \in c$ and return false otherwise. If there is a polynomial time algorithm for the evaluation problem, we say that the class is *efficiently evaluatable*

For our basic representations, the evaluation problem has straightforward linear-time algorithms; for example, a linear-time recursive algorithm for evaluating decision trees is essentially given in the definition of the semantics of the decision tree. There are

similarly very simple linear-time algorithms for evaluating conjunctions, disjunctions, and DNFs.

3.6 NP search problems, NP completeness, and Boolean circuits.

So far we have developed a theory of the "expressive power" of knowledge representations. Expressive power is desirable: more expressive power means that we can capture more properties with reasonable-size representations. But in this section we will see that expressive power usually comes at a cost in terms of what kinds of *operations* we can support on the representations. The more expressive a representation is, the smaller the variety of operations we can efficiently perform on it. Thus, expressive power is not unequivocally desirable; for any application, we must find a class of representations that balances our need to represent properties against our need to operate on those representations.

Another such basic operation is to find an example of a possible world satisfying the property if one exists, or else report that none exist. This is the satisfiability problem:

Definition 3.20 For a class of representations C, the *satisfiability problem* for C is, given a representation $c \in C$, if c is nonempty, return a possible world $x \in c$ represented as a Boolean vector. Otherwise, return $\perp$, representing none exist.

For example, we can solve the satisfiability problem for decision trees efficiently:

Theorem 3.21 *There is an algorithm for the satisfiability problem for decision trees with s nodes on n attributes (for which each attribute appears at most once on each root-to-leaf path) that runs in time $O(s+n)$.*

Proof If the tree representation does not initially indicate in each node which node is its "parent," we can scan the tree and create a version with these recorded in each node in time $O(s)$. Then, we can search for a leaf node labeled "true," returning $\perp$ if none exist, in time $O(s)$. Finally, supposing we have found such a node, we can initialize a vector of length n to all-false, and follow the parent edges from the leaf to the parents, back to the root, filling in a new assignment to X_i if X_i labeled the parent of a node reached via a "true" edge. Initializing the vector is time $O(n)$, and walking back up the tree is again $O(s)$. We then return the vector as our possible world satisfying the tree. Thus the overall time is $O(s+n)$. The algorithm is correct because the possible world contains true for each X_i for which the true edge appears on the path to the chosen leaf, and otherwise it contains false for each X_i; hence, the possible world will indeed reach the chosen leaf and is therefore included in the property represented by the tree. □

There is also a straightforward linear-time algorithm for solving satisfiability for DNFs. By contrast, as we will see, if we were instead looking for possible worlds that *do not* satisfy the property, essentially the same algorithm (starting from a "false" leaf) solves the problem for decision trees, but no efficient algorithm is believed to exist for DNFs. Thus, it turns out that the expressiveness of DNFs comes at a price: the

additional richness of the representation prevents us from efficiently supporting this basic operation on the representation.

To explain why we believe that such efficient algorithms cannot exist, we will need to be more formal about the kinds of computational problems we study. The definition of polynomial-time is important because it gives a precise criteria for what counts as an efficient algorithm, and hence allows us to ask whether or not any exist. Similarly, towards presenting this "evidence," we define some families of computational problems we consider:

Definition 3.22 A *relation problem* (or *search problem*) is the following. Let X be a set of inputs and Y be a set of outputs. Then for the relation $R \subseteq X \times Y$, the relation problem is given $x \in X$, return $y \in Y$ such that $(x, y) \in R$ or else return $\perp$, indicating that none exist.

Observe that the satisfiability problem is an example of a relation problem, in which $X = C$, Y is the set of possible worlds, and R is the set of pairs (c, x) such that $x \in c$. Likewise, the evaluation problem is a relation problem with $X = C \times \{0,1\}^n$ and $Y = \{0,1\}$ where $R = \{((c,x), b) : c(x) = b\}$.

Definition 3.23 A *NP search problem* is a relation problem for a relation R with the following properties:

1. R is *polynomially balanced*: there is a polynomial $p(n)$ for every $x \in X$ such that there exists some $y_0 \in Y$ for which $(x, y_0) \in R$, there also exists $y \in Y$ with length at most $p(|x|)$ (where $|x|$ denotes the length of the representation of x) such that $(x, y) \in R$
2. R is *efficiently evaluatable*: i.e., there is a polynomial-time algorithm that given $(x, y) \in X \times Y$, returns true if $(x, y) \in R$ and returns false otherwise.

As the representations we have considered so far are efficiently evaluatable, it is also straightforward to see that the evaluation and satisfiability problems for them are indeed examples of NP search problems.

But, the class of NP search problems also contains many problems for which we do not know efficient algorithms. For example, integer factoring: $\{(n, (p, q)) \in \mathbb{N} \times (\mathbb{N} \times \mathbb{N}) : n = p \cdot q; p, q \neq 1\}$. It is a centuries-old open problem to give a fast algorithm for factoring, and apart from quantum algorithms, none have been found. More broadly, most engineering problems can be cast as finding a design that meets some given verifiable requirements. Such problems qualify as NP search problems as well. Given the breadth and apparent difficulty of such problems, it seems unlikely that *all* such problems would yield to efficient algorithms.

It turns out that there are *specific* problems that, if solved by a polynomial-time algorithm, would enable the design of polynomial-time algorithms for *all* NP search problems. These are called "NP-complete" problems:

Definition 3.24 We say that a relation problem P *polynomial-time reduces to* a relation problem Q if there is a polynomial-time algorithm for P, that is allowed to use an instruction for Q that is assumed to execute in a single step.

We say that an NP search problem R is *NP-complete* if every NP search problem P polynomial-time reduces to R.

Note that if P reduces to Q, then given a polynomial-time algorithm for Q we can obtain a polynomial-time algorithm for P by taking the reduction algorithm and replacing the instruction for Q with the algorithm for Q. Since the reduction algorithm for P ran in some polynomial time $t_P(n)$ assuming it could solve Q in a single step, the inputs to Q have size at most $t_P(n)$, and the instruction for Q was invoked at most $t_P(n)$ times. So if the algorithm for Q ran in time $t_Q(n)$, the overall running time for the modified algorithm is at most $t_P(n) \cdot t_Q(t_P(n))$. Assuming these are both polynomials, the overall running time is still clearly polynomial. So, a polynomial-time algorithm for an NP-complete problem would yield polynomial-time algorithms for *all* NP search problems. As we discussed above, this seems unlikely; NP-completeness is thus our "evidence" that a problem is computationally intractable.

Similarly, given an NP-complete problem P, if P polynomial-time reduces to Q, then Q is also NP-complete: for any NP search problem R, R polynomial-time reduces to P. We can replace the instruction for P with the reduction algorithm of P to Q, to obtain instead an algorithm that solves R using an instruction for Q rather than P. Letting $t_R(n)$ now denote the (polynomial) running time of the reduction of R to P, we can similarly bound the running time of the algorithm for R using an instruction for Q instead by $t_R(n) \cdot t_P(t_R(n))$, which is also a polynomial. So, given one example of an NP-complete problem, it is easy to obtain others by giving a single polynomial-time reduction from that known NP-complete problem to the other problem. In this way, it is relatively easy to establish the intractability of many problems.

Unfortunately, the satisfiability problem for most expressive knowledge representations is usually NP-complete, and thus presumably intractable. The trouble is that NP-completeness exploits the expressive power of a relation problem. A sufficiently versatile representation can be used to capture a vast variety of other problems. In turn, this means that any difficulty that arises in solving the other problems also arises as a special case of the expressive problem. In this way, expressive power poses a liability.

It is thus easiest to start with particularly rich representations to obtain a first NP-complete problem. We thus introduce the Boolean circuit representation:

Definition 3.25 A *Boolean circuit* representation of a function $f : \{0,1\}^n \to \{0,1\}^k$ is given by a Directed Acyclic Graph with k sinks, in which the sources are labeled by either attributes $X_1, \ldots, X_n$ or the Boolean constants 0 or 1, the k sinks are distinctly labeled $Y_1, \ldots, Y_k$, and the remaining nodes are labeled with a De Morgan connective, AND, OR, or NOT. The nodes labeled by AND and OR must have two incoming edges, and the nodes labeled by NOT have one incoming edge.

The circuit represents the following function: For a given setting $x_1, \ldots, x_n$ of the n attributes, we first label each source node X_i with the Boolean value x_i. We next fix an ordering of the nodes such that if node i appears before j, then there is no path from j to i (this is called a *topological sort* of the nodes). Now, in this order, we label each node with the value obtained by evaluating the connective labeling that node on the input values given by the labels of the other endpoints of the incoming edges; these must be

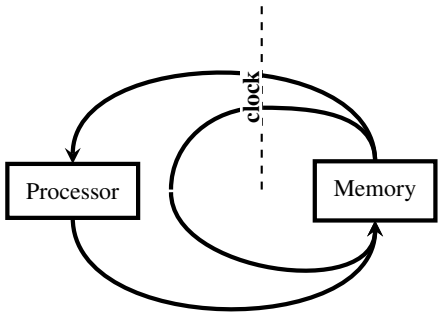


Figure 3.3 Computers are implemented by circuits with loops, broken at one point by a "clock."

labeled with Boolean values since they cannot appear later in the topological sort, so this value is indeed well-defined. The output is thus the values $y_1, \ldots, y_k$ labeling the nodes $Y_1, \ldots, Y_k$.

We note that the evaluation problem for Boolean circuits can be solved in polynomial time, once again by essentially following the definition. We consider Boolean circuits to be expressive because they can easily capture the execution of an arbitrary algorithm:

Proposition 3.26 *For every polynomial-time computable function f such that on inputs of n Boolean attributes, f produces $k(n)$ Boolean attributes as output, there is an algorithm that takes as input n, runs in time bounded by a polynomial in n, and outputs a Boolean circuit that computes f on inputs of length n.*

Unfortunately, the proof of Proposition 3.26 relies on the specification of a programming model for capturing all algorithms. In turn, we must justify that the programming model is indeed sufficiently expressive to capture all algorithms. Establishing such a programming model is beyond the scope of this book (see the chapter notes for references). We offer only the following sketch that captures the spirit of the construction, while glossing over the details of an actual programming model.

Sketch of proof: As informal justification, observe that real general-purpose computers certainly seem to be capable of executing all algorithms. We recall the following high-level structure of real computers. Real computers typically are constructed as circuits that, unlike those of Definition 3.25, have loops. At one point along the loops, there is a "clock" that prevents the values on one side from propagating to the other until a fixed amount of time has elapsed. The construction of the computer guarantees that all values from the previous "tick" of the clock have propagated fully through the circuit, before the next "tick" occurs, allowing those new values to propagate beyond the clock. These values specify the new contents of the computer's memory, and the values that were fetched from the memory on the previous cycle. The fetched values propagate through a processor that specifies new values to be stored in memory, and specifies values to be fetched from memory on the next cycle. (See Figure 3.3.) Thus, on each cycle, a single step of computation is performed. An algorithm's running time corresponds to the number of such steps.

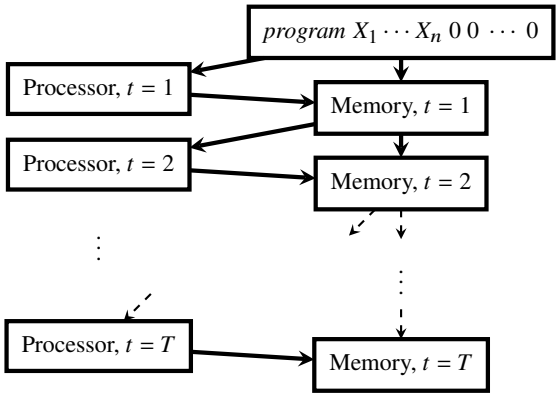


Figure 3.4 Unrolling the loop T times. The input is represented on the wires $X_1, X_2, \ldots, X_n$ with the program and 0s "hard-wired" in the rest of memory prior to the first step. This executes the program on input $X_1, \ldots, X_n$ for T steps. The output wires $Y_1, \ldots, Y_k$ carry appropriate positions of the memory at time T

So, we suppose that there is a program for computing the function f that, on inputs of length n, runs for at most $t(n)$ steps. $t(n)$ iterations of the loop are thus sufficient to obtain the $k(n)$-bit output. The input and output are written in specific locations in the memory. The program is also stored in a specific region of memory. We can thus obtain an acyclic circuit (per Definition 3.25) for computing f on inputs of length n as follows. We "unroll" the loop $t(n)$ times. That is: we make $t(n)$ copies of the circuit, but instead of placing the clock along a loop, we join the wires on one side of the clock in the ith copy to the wires on the other side in the $i+1$st copy (without a clock in between). We fix the Boolean values encoding the program incoming to the memory in the first copy (we will refer to this as *"hard-wiring"* the values), label the wires encoding the input with the corresponding attributes $X_1, \ldots, X_n$, and hard-wire the rest of wires incoming to memory to value 0. We will label the wires encoding the output in the final, $t(n)$th copy as $Y_1, \ldots, Y_{k(n)}$, and discard any gates from the circuit that do not have a path to one of these wires. (See Figure 3.4 for an illustration.)

Using the fact that the circuit can only read and write from $O(1)$ locations per cycle, we observe that we only need $O(t(n))$ distinct locations in memory to execute the program. We can execute the program using a memory of size only $O(t(n))$ by storing a value along with the intended address; to store or fetch a value, the circuit checks all currently-represented locations for a match, and if it finds none, it creates a new location with the needed address, containing value 0.

Thus, we only need to produce $t(n)$ copies of circuits of size $O(t(n))$. As we have by hypothesis that $t(n)$ is a polynomial in n, it is easy to print out a representation of this circuit in time polynomial in n as well. By construction, on any input $x_1, \ldots, x_n$, the values that propagate through the ith copy of the circuit match those that would propagate through the original circuit on the ith cycle. The output wires will therefore contain $f(x_1, \ldots, x_n)$ as needed. □

So we see that circuits are powerful indeed. It is now relatively straightfoward to show that the satisfiability problem for circuits is NP-complete, and hence presumed to be intractable.

Theorem 3.27 (Cook-Levin) *The satisfiability problem for Boolean circuits is NP-complete.*

Proof We first note that the satisfiability problem for circuits is indeed an NP search problem: since the circuit itself represents the input attributes, the output for the satisfiability problem is at most linear in the input size, and is hence polynomially bounded. Furthermore, we note that the relation corresponds to the evaluation problem for Boolean circuits, and is hence solved by a polynomial-time algorithm.

We now consider an arbitrary NP search problem R, for which the relation is likewise polynomially balanced and efficiently evaluatable. We note that we can convert R into R' for which the evaluation algorithm allows "padding" of 0s to be added to the output y, e.g., by indicating how many of the symbols are actually part of the output. the length of y. Thus, for the bound $p(|x|)$ in the polynomial balance condition for R, there is a polynomial $p'(|x|)$ such that if $(x, y) \in R$, there is a y' such that (x, y') has length *exactly* $p'(|x|)$ in R', and the output is a single bit encoding true or false.

The reduction is now as follows. On input $x \in X$ of length n, by Proposition 3.26 now, there is an algorithm running in time polynomial in $p'(n)$ (thus also, polynomial in n) that outputs a circuit computing this modified R' on inputs of length $p'(n)$. We hard-wire the first part of the input to x (i.e., replacing the labels for $X_1, \ldots, X_n$ with the Boolean values $x_1, \ldots, x_n$), and invoke our instruction for solving the satisfiability problem for circuits on this circuit. Supposing there is an input that makes the circuit output 1, then we obtain a representation of some y' such that $(x, y') \in R'$; we can discard the padding in y' and output y such that $(x, y) \in R$. Otherwise, the circuit satisfiability instruction returns $\perp$, and we can return this as well, since then there is no y such that $(x, y) \in R$. The reduction is clearly polynomial time and solves R, as needed. Thus, as R was an arbitrary NP search problem, we see that the satisfiability problem for Boolean circuits is indeed NP-complete. □

Again, in light of Proposition 3.26, it may seem unsurprising that the satisfiability problem for circuits is NP-complete. What may come as a surprise is that NP-completeness extends to representations that are much less expressive. We earlier mentioned the problem of finding a possible world where a given DNF property does not hold; this corresponds to a satisfiability problem for the *negation* of a DNF, which by De Morgan's law, can be written as a Conjunctive Normal Form formula (CNF):

Definition 3.28 *Conjunctive Normal Form (CNF) formulas* are the following family of Boolean formulas:

- A disjunction is a CNF (called a *clause*).
- The AND $[\varphi \wedge \psi]$ of two CNF formulas is a CNF.

If the clauses all have width at most k, we refer to the formula as a *k-CNF*.

I.e., a CNF is an AND of ORs of literals, where we refer to the ORs of literals as the *Clauses* of the CNF.

CNFs arise naturally as descriptions of the possible worlds that simultaneously obey a collection of if-then rules, as described in Section 3.3. As such collections of rules naturally describe many domains, it is particularly unfortunate that such basic problems involving CNFs are intractable.

Theorem 3.29 *The satisfiability problem for 3-CNFs is NP-complete.*

The satisfiability problem for 3-CNFs is conventionally known as 3SAT.

Proof As with the satisfiability problem for circuits, the attributes are represented as part of the 3-CNF, and hence the polynomial balance condition is satisfied. Likewise, since Boolean formulas can be evaluated in polynomial time, the relation is efficiently evaluatable. 3SAT is thus an NP search problem.

We now show how to reduce the satisfiability problem for Boolean circuits to 3SAT. The main idea is that we introduce new attributes that indicate the values that gates of the circuit output. Given proposed values for the inputs and output of a gate, we'll be able to check that these are correct using a clause of width at most three as follows. The AND and OR gates take two inputs and produce a single output; the NOT gates take a single input and produce a single output. For each of the $2^2 = 4$ possible pairs of input values to an AND/OR gate (and two possible input values for a NOT gate), we can write a rule indicating what value the gate should output. For example, for an AND gate with output represented by the attribute G_k taking inputs from gates represented by attributes G_i and G_j, we have the following rules:

$$\begin{aligned}
[G_i \wedge G_j] &\Rightarrow G_k \\
[G_i \wedge \neg G_j] &\Rightarrow \neg G_k \\
[\neg G_i \wedge G_j] &\Rightarrow \neg G_k \\
[\neg G_i \wedge \neg G_j] &\Rightarrow \neg G_k
\end{aligned}$$

Note that for each possible pair of values that G_i and G_j output, exactly one of these rules may be applied to infer the value that G_k must output. As we noted in Section 3.3, De Morgan's law allows us to rewrite each of these as a clause, giving us the set of clauses

$$\neg G_i \vee \neg G_j \vee G_k,\ \neg G_i \vee G_j \vee \neg G_k,\ G_i \vee \neg G_j \vee \neg G_k,\ G_i \vee G_j \vee \neg G_k.$$

Similar sets of clauses can be written for the OR and NOT gates.

The reduction is thus as follows. Given as input a Boolean circuit, we construct a 3-CNF with one variable for each input and output, and one variable for each gate. We start with an "empty" CNF (with no clauses). We scan the Boolean circuit and for each gate, we add the corresponding set of clauses for that gate as described above, that capture the rules specifying the value that the gate must output given the values of the input gates. Finally, for the sole output Y_1, which is produced by some gate G_s, we include the clause consisting of the single literal, G_s, which is thus only satisfied if the gate outputs 1. We invoke the instruction for 3SAT on this formula. If there is a set

of values $x_1, \ldots, x_n, g_1, \ldots, g_s$ for the attributes that evaluates to true for our 3-CNF, we obtain such a setting and otherwise we obtain $\perp$. In the former case, the reduction returns $x_1, \ldots, x_n$ and in the latter case it returns $\perp$. It is easy to see that the running time is indeed polynomial in the representation size of the circuit.

We observe that for any set of values for $X_1, \ldots, X_n$, there is a unique set of values for $G_1, \ldots, G_s$ that satisfies the set of constraints we have constructed for the clauses: the values that the gates would output when the circuit is evaluated on input $x_1, \ldots, x_n$. Supposing that we have ordered the gates so that if there is a path from gate i to gate j, then i appears before j in the ordering, we argue by induction on the gates. The base case is trivial – we are simply given a set of values $x_1, \ldots, x_n$ for $X_1, \ldots, X_n$. So now, supposing that $G_1, \ldots, G_i$ are uniquely determined to take values $g_1, \ldots, g_i$ for $x_1, \ldots, x_n$, we see that in particular the gates j and k that are inputs to $i + 1$ must, by construction, appear before $i + 1$ in the ordering and hence their outputs are uniquely determined to be the values g_j and g_k that these gates would output when the circuit is evaluated on input $x_1, \ldots, x_n$. Thus, for the rule that determines G_{i+1} when $(G_j, G_k) = (g_j, g_k)$, we have a clause that is only satisfied when G_{i+1} takes a specific value g_{i+1} that is obtained by evaluating gate $i + 1$ on the values that gates j and k would take when the circuit is evaluated on input $x_1, \ldots, x_n$. This is indeed the unique value, in turn, that gate $i + 1$ would output on that input.

It is now clear that if the reduction returns $x_1, \ldots, x_n$, then this must be an input to the circuit such that the gate s producing the output is given the value $g_s = 1$ to satisfy the clause G_s; it is thus as needed for the satisfiability problem for circuits. Conversely, if there is a set of inputs $x_1, \ldots, x_n$ for which the circuit evaluates to 1, the assignment $g_1, \ldots, g_s$ to $G_1, \ldots, G_s$ corresponding to the values that the gates output on that input indeed yields a satisfying assignment to the constructed 3-CNF, and so the instruction for 3SAT must return some values for $X_1, \ldots, X_n$ (and not $\perp$) in this case; again, as we just argued, whatever values are returned indeed cause the circuit to output 1. The reduction thus solves the satisfiability problem for circuits as needed. As the Cook-Levin Theorem gave us that the satisfiability problem for circuits was NP-complete, by our earlier observation that polynomial-time reductions confer NP-completeness, we therefore may conclude that 3SAT is NP-complete as well. □

Chapter notes

The Boolean approach to logic as functions of truth values was developed by Boole (1847; 1854) and De Morgan (1847), which in particular formally introduced De Morgan's Law. These initial systems were not quite mature: in particular, Boole used the arithmetic + instead of the logical or connective, equations rather than implication, and had an understanding of negation that was closer to arithmetic. These issues were addressed by Peirce (1865) and independently by Jevons (1874). Disjunctive Normal Form appears in these early systems as the natural analogue of the expansion of a polynomial into a sum of products. Truth tables were proposed by Wittgenstein (1922) and Post (1921) (though Post duly remarks that this is essentially disjunctive

normal form, which was used earlier). Our "database representation" is a strawman simplification of Codd's model (1970), which was of course based on these earlier systems of logic. Decision trees were first proposed by Belson (1959).

Our modern understanding of the general-purpose, programmable computer is essentially due to Turing (1936). The application of Boolean logic to digital circuits was pioneered by Shannon in his Master's thesis (Shannon, 1938), and independently by Nakashima (1935) and Sheskatov (1938); we now know that Peirce had the idea much earlier, but did not publish it. The basic principles of computational complexity of algorithms were developed first by Hartmanis and Stearns (1965). The identification of "efficient algorithm" with "polynomial time" was independently proposed by Edmonds (1965) and Cobham (1965); although Cobham's motivation is more principled, Edmonds also suggested that some combinatorial problems are inherently intractable, anticipating the development of NP-completeness a few years later by Cook (1971) and Levin (1973). In particular, Cook proved that satisfiability for 3-CNF formulas is NP-complete, and we use Levin's formulation in terms of search problems. But, the transformation of circuits into CNF formulas was first proposed by Tseitin (1968). Exercise 3.8 was first proved by Khrapchenko and independently by Spira (1971).

We did not present the details of the proof of Proposition 3.26 because we did not develop a formal programming model, and thus our construction of the first NP-complete problem is not complete. Many standard textbooks for the theory of computation provide the ground work by developing such a programming model and rigorously establish the NP completeness of CNF satisfiability. We recommend Sipser (2012) as a starting point, using the Turing machine model. Other authors, e.g. Papadimitriou (1994), furthermore present the simulation of a RAM computer by a Turing machine, which thus provides a complete compilation of a standard, familiar programming model down to a Boolean circuit. Readers familiar with digital circuits may find the roundabout path via Turing machines unsatisfying. The virtue of the Turing machine is its mathematical simplicity, rendering such arguments cleaner.

We have only provided a minimal introduction to the topic of the relative expressive power of Boolean function families. Jukna (2012) provides a much more complete overview of the expressive power of different families of Boolean functions and their relationships.

Exercises

3.1 *Writing down encodings of rules.* Consider a simplified smart thermostat system with the following Boolean attributes:

H = "the home is occupied",	C = "the current temperature is cold",
W = "a window is open",	M = "manual override is enabled",
P = "energy-saving mode is enabled",	F = "the furnace is on".

For each of the following informal rules, write an equivalent propositional formula. When possible, rewrite the rule as a disjunction of literals.

(a) If the home is occupied and the temperature is cold, then the furnace is on.

(b) The furnace is on only if the temperature is cold or manual override is enabled.
(c) The furnace is on if and only if the temperature is cold, the home is occupied, and no window is open.
(d) For each formula above, say whether it is a conjunction, a disjunction, a DNF formula, a CNF formula, or none of these. If the formula is not already in CNF, rewrite it as an equivalent CNF formula.

3.2 *Separating the expressive power of DNFs, CNFs, and Decision Trees.* Consider the following linear-size DNF formula defining the function $\text{Tribes}_{2,n/2}(x_1, \ldots, x_n)$

$$\text{Tribes}_{2,n/2}(x_1, \ldots, x_n) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{n-1} \wedge x_n)$$

(a) Describe a CNF formula for $\text{Tribes}_{2,n/2}$. (***Hint.*** *It is helpful to consider the extremal cases of inputs where as many of the attributes are true as possible but* $\text{Tribes}_{2,n/2} = 0$ *and where as many of the attributes are false as possible, but* $\text{Tribes}_{2,n/2} = 1$.)
(b) Prove that any CNF formula for $\text{Tribes}_{2,n/2}$ requires size $2^{\Omega(n)}$. (***Hint.*** *Consider your solution to (a). What features of your solution were forced, and why?*)
(c) Prove that (b) implies that any Decision Tree for $\text{Tribes}_{2,n/2}$ must also have size $2^{\Omega(n)}$.

3.3 *Exclusive-or cannot represent conjunction or disjunction.* An *exclusive-or formula* is a formula in which the only connectives are the exclusive-or ($\oplus$). We allow the inputs to be attributes $X_1, \ldots, X_n$ or the Boolean constants 0 and 1.
(a) Prove that there is no exclusive-or formula representing the conjunction $X_1 \wedge X_2$.
(b) Prove that there is no exclusive-or formula representing the disjunction $X_1 \vee X_2$.

3.4 *DNF for parity has exponential size.* Prove that any family of DNF formulas representing the formulas $X_1 \oplus X_2 \oplus \cdots \oplus X_n$ (i.e., with one DNF formula representing the exclusive-or formula for each number of inputs n) must have at least $2^{\Omega(n)}$ terms.

3.5 *Majority can represent disjunction but not DNF.* The *majority function* $\text{Majority}(X_1, \ldots, X_n)$ is the Boolean function that evaluates to 1 if at least $n/2$ of its n inputs are 1. Otherwise, it takes value 0. A *majority circuit* representation of a function $f : \{0,1\}^n \to \{0,1\}^k$ is given by a Directed Acyclic Graph with k sinks, in which the sources are labeled by either attributes $X_1, \ldots, X_n$ or the Boolean constants 0 or 1, and k sinks are distinctly labeled $Y_1, \ldots, Y_k$. (We do not limit the in-degree of the nodes here.) Again, for a possible world $\vec{x} \in \{0,1\}^n$, the value of the gates of the circuit are determined in topological order: each gate computes the majority function of the values of the nodes with edges to it.
(a) Prove that there is a majority circuit representing the disjunction $X_1 \vee X_2$.
(b) Prove that there is a DNF formula representing a function that cannot be represented by any majority circuit.

3.6 *DNF for majority has exponential size.* Prove that any family of DNF formulas representing the functions $\text{Majority}(X_1, \ldots, X_n)$ (defined in the previous exercise) must have $2^{\Omega(n)}$ terms.

3.7 *OBDD vs. Decision tree.* An *Ordered Binary Decision Diagram (OBDD)* is a widely-used representation of Boolean functions. An OBDD consists of a Directed Acyclic Graph with a single source and two sinks, labeled 0 and 1. Its non-sink nodes are labeled by attributes, and each have out-degree 2. For each of these nodes, one edge is labeled 0 and the other edge is labeled 1. An OBDD has an associated *ordering* of the attributes, given by a permutation π of the indices $\{1, \ldots, n\}$, and on any path in the OBDD, edges may only proceed from nodes with label $X_{\pi(i)}$ to nodes with label $X_{\pi(j)}$ for $i < j$—i.e., the attributes are read in the order specified by π on every path. The computation of an OBDD is analogous to a decision tree: on any possible world $\vec{x} \in \{0,1\}^n$, starting at the source node of the OBDD, if the current node is labeled by X_i, then if $x_i = 0$ the computation proceeds by traversing the edge labeled 0 exiting that node, and otherwise (if $x_i = 1$), the computation follows the edge labeled 1. When a sink is reached, the Boolean value is the output value of the OBDD.
(a) Prove that there is a family of OBDDs of size $O(n)$ such that every family of decision trees representing the same functions must have $2^{\Omega(n)}$ nodes. (***Hint.*** *Use the result of Exercise 3.2 part (c).)*
(b) Prove that there is a family of decision trees of size $O(n)$ such that every family of OBDDs representing the same function has $2^{\Omega(n)}$ nodes.

3.8 *Formulas are log-depth circuits.* The *depth* of a Boolean circuit is the longest path from a source to a sink in the circuit.
(a) Prove that in any undirected graph G such that every node has degree at most 3, there is a vertex v^* in G such that removing v^* from G breaks G into components each containing at most $2/3$ of the nodes of G.
(b) Using (a), prove that every family of Boolean formulas $F_1, F_2, \ldots, F_n, \ldots$ can be represented by a corresponding family of Boolean circuits (where F_n is represented by C_n) such that each C_n has depth $O(\log n)$.
(c) Prove that any Boolean circuit with s nodes and depth d can be represented by a Boolean formula of size bounded by a polynomial in s and 2^d. (Thus: if s is a polynomial and n and $d = O(\log n)$, the circuit has size polynomial in n.)

3.9 *Tractable operations versus expressive power I.* Suppose that $\mathcal{A}$ and $\mathcal{B}$ are efficiently evaluatable knowledge representations such that $\mathcal{B}$ is more expressive than $\mathcal{A}$, the satisfiability problem for $\mathcal{A}$ is NP-complete, and there is a polynomial-time algorithm that, given a representation in $\mathcal{A}$, returns an equivalent representation in $\mathcal{B}$. Prove that the satisfiability problem for $\mathcal{B}$ is NP-complete as well. (Thus: for any representation that is more expressive than 3-CNF in the above sense, the satisfiability problem is intractable.)

3.10 *Tractable operations versus expressive power II.* For any class of efficiently evaluatable representations C, the NP search problem Difference-C is: given two representations C_1 and C_2 in C, find a possible world $\vec{x}$ such that $C_1(\vec{x}) \neq C_2(\vec{x})$ (equivalently, $\vec{x}$ is in the symmetric difference of the sets defined by C_1 and C_2).
(a) Prove that there is a polynomial time algorithm for Difference-DecisionTree.
(b) Prove that Difference-DNF is NP-complete.

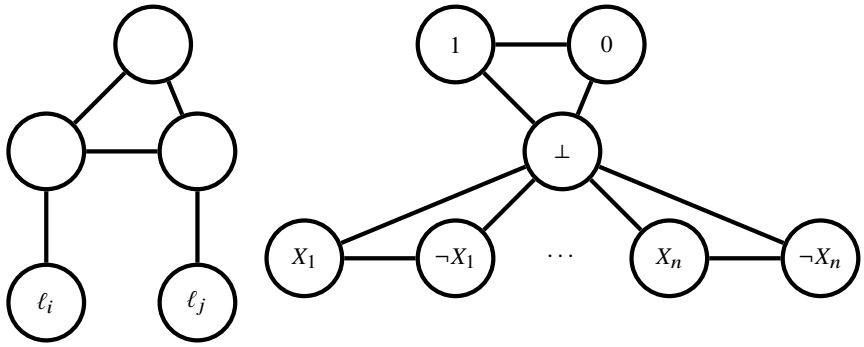


Figure 3.5 Example graph structures for Exercise 3.12.

3.11 *Independent Set is NP-complete.* Given an undirected graph $G = (V, E)$, an *independent set* in G is a subset of vertices $S \subseteq V$ such that no pair is joined by an edge. The Independent Set search problem is, given a pair (G, k) where G is an undirected graph and k is an integer, return an independent set S in G with k vertices.
(a) Prove that Independent Set is an NP search problem.
(a) Prove that Independent Set is NP-complete.

3.12 *3-coloring is NP-complete.* Given an undirected graph $G = (V, E)$, a *proper 3-coloring* is an assignment of one of three different *colors* to each of the vertices of G such that every pair of vertices u and v that are joined by an edge in G receive different colors. The 3-coloring search problem is, given an undirected graph G, return a proper 3-coloring of G.
(a) Prove that 3-coloring is an NP search problem.
(a) Prove that 3-coloring is NP-complete. *(**Hint:** identify a mapping from parts of an input 3CNF φ to the kinds of graph structures illustrated in Figure 3.5.)*