Automated Deduction with Shannon Graphs

Joachim Posegga & Peter H. Schmitt

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
76128 Karlsruhe, Germany
posegga@ira.uka.de, pschmitt@ira.uka.de

Abstract

Binary Decision Diagrams (BDDs) are a well-known tool for representing Boolean functions. We show how BDDs can be extended to full first-order logic by integrating means for representing quantifiers. The resulting structures are called Shannon graphs. A calculus based on these Shannon graphs is set up, and its soundness and completeness proofs are outlined.

A comparison of deduction with first-order BDDs and semantic tableaux shows that both calculi are closely related. From a practical perspective, however, BDDs have advantages over tableaux: they provide a more compact representation, since BDDs can be understood as a linear, graphical representation of a fully expanded tableaux. Furthermore, BDDs represent not only the models of a formula, but also its counter models: this offers a very efficient way to represent lemmata during the proof search.

The last part of the paper introduces a compilation-based approach to implementing deduction systems based on Shannon graphs. The idea is to compile the graphs into programs that carry out the proof search at run time.

1 Introduction

If a computer scientist is asked to suggest an efficient formalism for representing and manipulating boolean functions, the answer will probably be *Binary Decision Diagrams* (BDDs). Their underlying idea is to use a graph representation of the *if-then-else* normal form for propositional formulæ. The success of the method has been so convincing[1], that it became very tempting to try an extension of this approach to full first-order predicate logic. This is done in this paper, which is a condensed version of [22]. It was not clear from the start what changes would become necessary and whether efficiency could still be obtained. We hope to convince the reader that our extension of the BDD method results in a very efficient proof procedure for first-order logic.

[1] BDDs proved to be useful in applications like processing database queries [17], pattern recognition [4], and diagnosis [11]. The most successful application, however, lies surely within the design of digital systems (see Bryant [7, 8] for introductory papers, or [6, 10] for papers describing applications).

Attempts to use BDDs in automated deduction are sparse. This may be attributed to the fact that BDDs have been developed in a different scientific community[2]. The only contributions we know of are:

- Ehrenfeucht and Orłowska [14] describes a propositional proof procedure that works on structures similar to BDDs; this method has been extended to a decidable subset of first-order logic [19]. Although the propositional version of the method was actually implemented, it is chiefly a theoretical contribution that is not well suited for an implementation.

- Billon [5] describes an approach to representing quantifier-free formulæ of first-order logic in so-called *Typed Decision Graphs*, which are an extension of ordered BDDs. Billon's approach requires a formula to be in prenex normal form; furthermore, it is not possible to have free variables in the graphs.

Independently of Billon, a method for first-order deduction with non-ordered BDDs was developed by the author [23, 22]. This method also requires a prenex normal form, but allows free variables in the graphs. In this paper, the method is carried forward to general first-order formulæ which need not be in prenex normal form; this is achieved by explicitly representing the scope of quantifiers in the graphs. The shift from quantifier-free to general first-order formulæ avoids much redundancy in the proof search.

One of the distinctive features of BDDs is the use of an ordering on the propositional variables. In the first-order case this would be an ordering on atomic formulæ, which is incompatible with the use of free variables during proof search: each application of a substitution can destroy the ordering and iterated re-ordering can be of exponential cost. Our experience with semantic tableaux proof procedures [2] has shown that the use of free variables is crucial. So we decided to drop orderings. This is less painful when we observe that unique normal forms, which are the main benefit from using orderings in propositional logic, cannot be obtained in first-order logic, since it is only semi-decidable[3]. The other alternative, to retain orderings and give up free variables, is certainly less suited for proof search in general, but may have its merits under special circumstances; the interested reader is referred to [22] for a detailed account.

Since the structures we will introduce below differ not only in the lack of an ordering from "standard" BDDs but also by other peculiarities, which allow e.g. the representation of the scope of quantifiers, we decided to name them differently: *Shannon graphs*.

At first glance BDDs seem rather different from other approaches used in automated deduction. However, we will reveal close connections between Shannon graphs and semantic tableaux (see [15] for a good introduction): a Shannon graph can be seen as an efficient representation of a fully expanded tableau.

The basic idea of tableaux is to prove the inconsistency of a formula by failure of a model-construction process: it is tried to satisfy a formula by stepwise refinement of potential models, until a contradiction in a potential model is detected; if all models

[2]Only few textbooks on logic treat special normal forms that are close to BDDs (see e.g. chapters 10, 12.4, and 12.5 in [1, pp. 80ff and 99ff] or §24 in [12, pp. 129ff]).

[3]Even for propositional logic it is still subject to research whether relaxing or removing ordering restrictions on "standard" BDDs is advantageous in some cases [9]

are ruled out, it is shown that the formula is inconsistent. This refutation-orientedness suggests that it is sufficient to consider only models. Shannon graphs on the other hand, being derived from a formalism for boolean functions, offer a representation of both, models and counter-models of a formula in linear time and space[4]. We will see that this has also advantages for refutation-oriented deduction, especially if lemmata are used in the proof search.

The idea of a lemma in tableau-based calculi is to provide the information that certain models have already been ruled out The same effect can be achieved by adding information about counter-models of a lemmatized formula to the branch where the lemma is to be used. This is best seen by an example:

Assume, that we treated a disjunction $\phi \vee \psi$ during the proof search and had already ruled out the models satisfying ϕ. The goal is now to show that ψ is contradictory as well. It is done by ruling out the models of ψ. As ϕ is already known to be false, we can safely assume that $\neg\phi$ is true; thus, this information can be added to the model(s) of ψ we will be investigating. This additional assumption can lead to earlier contradictions in those models. As $\neg\phi$ will in general be a compound formula, we will usually have to decompose it for deriving a contradiction. Semantically, this means that the counter-models of ϕ need to be examined – together with the models of ψ. The exploration of these counter-models is usually done independently of the earlier investigation of the models of ϕ, thus duplicating effort. This overhead can be avoided with Shannon graphs, since counter models are inherently present at no extra cost.

Shannon graphs lend themselves for efficient implementations. We pursue here a compilation-based approach guided by the analogy between tableau-based deduction and an interpreter for a non-deterministic programming language: the tableau is seen as a program containing statements to be executed (i.e.: formulæ to be expanded), until certain conditions are met and the process terminates (i.e.: all branches are closed). Non-determinism is usually resolved by storing the current tableau in an explicit data structure and modifying it during runtime according to fixed control strategies. This paper proposes a method for moving those control decisions into a preprocessing phase, such that the actual proof search can be carried out faster. Control of the proof search is thus moved from the meta to the object level. The principle of *compiling* proof search with Shannon graphs can be translated via the correspondence mentioned above to tableaux-based proof procedures.[5].

More precisely the proof procedure first transforms formulæ into a Shannon graph, and then compiles this graph into a program which shows the formulæ's inconsistency if it is executed. See Figure 1. The compilation process will be described for target language Prolog, but any other general-purpose language can be chosen. Prolog is just *convenient*, because it offers many primitives for dealing with first-order logic that must be implemented in other languages.

Our method differs essentially from other approaches to deduction by Horn clause generation (e.g. [25]), in that the generated clauses have no logical relation to the formula that is to be proven (i.e. they are not a logically equivalent variant of the formula), but that they are *procedurally* equivalent to the search for a model. This makes it also possible to compile the graphs to other programming languages, without

[4] W.r.t. the length of the negation normal form of a formula.

[5] [21] describes how the proof search of standard tableau can be compiled directly.

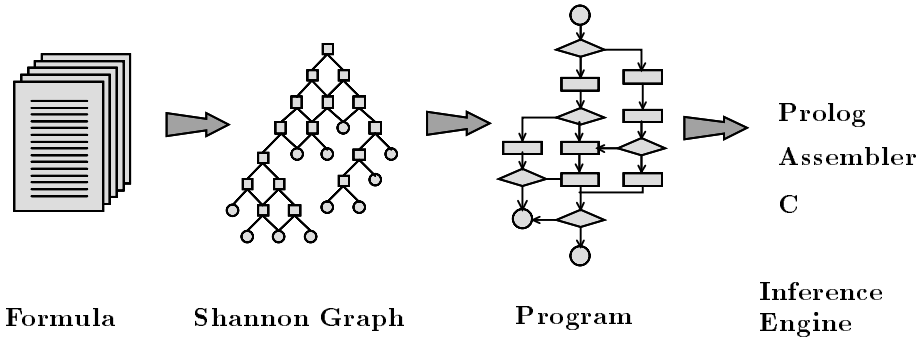


Figure 1: An Overview of the Proof Procedure

changing the principles described in this paper. Furthermore, the length of proofs can often be decreased compared with PTTP-like approaches, since our method does not require clausal form and quantifiers can remain at their original position.

The paper is written from a practical point of view and intends to show how methods based on Shannon graphs can be applied to deduction. We do not concentrate on theoretical issues, but provide the theoretical background that is necessary to understand the proposed method. When arguing on the implementation level, clearness and readability is preferred over showing how to achieve efficient code. A comprehensive treatment of implementation issues can be found in [18].

1.1 Paper Outline

The paper starts by setting up a formal framework for Shannon graphs in section 2 and 3. In the following section 4 we review a deduction procedure based on Shannon graphs, that works for universal (prenex) formulæ, while section 5 contains a full treatment the general first-order case. Here we introduce Shannon graphs that allow an explicit representation for quantifiers; We set up a refutation-oriented calculus, show its soundness and completeness in section 6. Section 7 investigates and formalizes the relation between Shannon graphs and semantic tableaux.

In section 8 an approach for implementing a deduction system based on Shannon graphs by compiling the graphs into programs is presented; performance figures of an implemented prototype are given and some extra-logical features for a practical application of the method are discussed. We draw conclusions from our research in section 9.

The text assumes the reader to be familiar with first-order predicate logic and the basics of free variable tableau (see e.g. Fitting [15]). Familiarity with BDDs is not required, but helpful; Bryant [8] gives a short introduction.

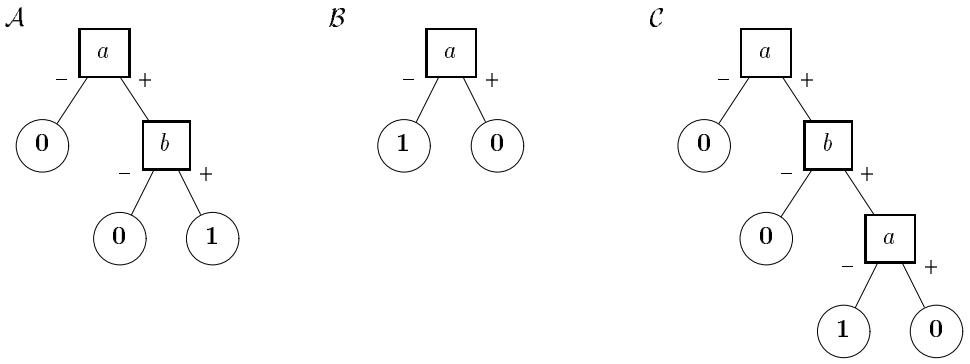


Figure 2: Shannon expressions for $a \wedge b$, $\neg a$, $a \wedge b \wedge \neg a$

R

2 Basic Definitions

Let $\mathcal{L}$ be the language of first-order logic and $\mathcal{L}_{At}$ the atomic formulæ of $\mathcal{L}$. The propositional case is subsumed by allowing 0-ary predicate symbols in $\mathcal{L}_{At}$ acting as propositional variables. Assume further, that the language $\mathcal{L}$ does not contain the atomic truth values "**1**" (*true*) and "**0**" (*false*).

Definition 1 (*sh*-connective) "*sh*" is a new logical connective of arity three, defined as: $sh(A,B,C) \stackrel{\text{def}}{=} ((\neg A \wedge B) \vee (A \wedge C))$.

$sh(A,B,C)$ can be read as as "**unless** A **then** B **else** C".

It is often argued that an *if-then-else*–form is more natural than the reversed notation of *unless-then-else*; indeed, the whole framework can be equally well set up in both ways, and it is mainly a matter of taste which is preferred. We will stick the *unless-then-else* notation, simply because it is more common.

The "*sh*"-connective can be used to form nested expression, which constitute the set of all Shannon expressions; we will use letters of the calligraphic alphabet "$\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$" to denote members of this set:

Definition 2 (Shannon expressions) The set of **Shannon expressions** is denoted by SH and defined as the smallest set such that

1. $\mathbf{1}, \mathbf{0} \in \mathrm{SH}$
2. *if* $\mathcal{A}, \mathcal{B} \in SH$ *and* $\phi \in \mathcal{L}_{At}$ *then* $sh(\phi, \mathcal{A}, \mathcal{B})$ *is in SH.*

If $\mathcal{L}_{At}$ is restricted to contain only propositional variables, we speak of propositional Shannon expressions, otherwise of first-order Shannon expressions. $\mathbf{1}, \mathbf{0}$ denote the truth values "true" (**1**) and "false" (**0**).

The semantics of Shannon expressions is defined by extending the standard evaluation function $val_{\mathcal{D},\beta} : \mathcal{L} \rightarrow \{\mathbf{0}, \mathbf{1}\}$ to a mapping $\mathcal{L} \cup \mathrm{SH} \rightarrow \{\mathbf{0}, \mathbf{1}\}$, according to Definition 1.

We can visualize formulæ of SH as binary trees whose nodes are labeled with the first argument of a *sh*-expression, and whose leaves are labeled with "**1**" or "

0". Consider, for example, the formula $a \wedge b$: a logically equivalent Shannon graph is $sh(a, \mathbf{0}, (sh(b, \mathbf{0}, \mathbf{1})))$, the corresponding tree is shown in Figure 2 (tree $\mathcal{A}$). All leaves are represented as circles, and all other nodes as squares. The edges labeled with "–" and "+" lead to the nodes representing the second and third argument of a "sh"-expression; the nodes are labeled with the first argument. We will say that an edge labeled with "+" is the **positive edge** of a node, the other one will be called the **negative edge**. Instead of writing "the node labeled with ϕ", we will simply write "the node ϕ". We will switch between referring to a "sh"-expression and to its corresponding tree without further notice.

Semantically, a propositional Shannon expression can be regarded as a case-analysis over the truth values of the atoms in a formula; consider the example above: if a is false, then $a \wedge b$ will be false, regardless of the truth-value of b. This is represented by the "–"-edge from the root to **0** in $\mathcal{A}$ of Figure 2. Otherwise, if a is true, the truth value depends on the value of b, i.e., $a \wedge b$ is true if b is true, and false, otherwise.

We will need to refer to the arguments of a sh-expression; this can conveniently be achieved with three projections denoted by $[\ldots]_{\mathsf{nd}}$, $[\ldots]_{-}$, and $[\ldots]_{+}$:

Definition 3 (Accessing Arguments of sh-expressions)

$$[sh(A, \mathcal{B}, \mathcal{C})]_{\mathsf{nd}} \stackrel{\text{def}}{=} A, \quad [sh(A, \mathcal{B}, \mathcal{C})]_{-} \stackrel{\text{def}}{=} \mathcal{B}, \quad \text{and} \quad [sh(A, \mathcal{B}, \mathcal{C})]_{+} \stackrel{\text{def}}{=} \mathcal{C}$$

(In the first case, nd stands for "node", $\mathcal{B}$ and $\mathcal{C}$ are also called **cofactors**.) Furthermore, we define: $[\mathbf{0}]_{\mathsf{nd}} = \mathbf{0}$ and $[\mathbf{1}]_{\mathsf{nd}} = \mathbf{1}$

The following logical properties of Shannon expressions form the basis for further development:

Proposition 4 (Simplifying sh-expressions)

1. $\neg sh(P, \mathcal{Q}, \mathcal{R}) \Leftrightarrow sh(P, \neg\mathcal{Q}, \neg\mathcal{R})$
2. If $sh(P, \mathcal{Q}, \mathcal{R}) \in \mathrm{SH}, F \in (\mathcal{L} \cup \mathrm{SH})$ and "$\circ$" is any binary logical operator, then: $(sh(P, \mathcal{Q}, \mathcal{R})) \circ F \Leftrightarrow sh(P, \mathcal{Q} \circ F, \mathcal{R} \circ F)$

Proof Part 1 follows directly from the definition of the sh-operator (1):

$$\begin{aligned}
\neg(sh(P, Q, R)) &\Leftrightarrow \neg((\neg P \wedge Q) \vee (P \wedge R)) && \textit{(Definition 1)} \\
&\Leftrightarrow (P \vee \neg Q) \wedge (\neg P \vee \neg R) \\
&\Leftrightarrow \underbrace{(P \wedge \neg R)}_{A} \vee \underbrace{(\neg Q \wedge \neg P)}_{B} \vee \underbrace{(\neg Q \wedge \neg R)}_{C} \vee \underbrace{(P \wedge \neg P)}_{\mathit{false}} \\
&\Leftrightarrow (P \wedge \neg R) \vee (\neg Q \wedge \neg P) && \textit{(since } C \rightarrow (A \vee B)\textit{)} \\
&\Leftrightarrow sh(P, \neg Q, \neg R)
\end{aligned}$$

Part 2:

Case 1: $val_{\mathcal{D},\beta}(P) = 0$				
then	$val_{\mathcal{D},\beta}(sh(P, Q, R)) \circ F$	$=$	$val_{\mathcal{D},\beta}(Q \circ F)$	*(Def. 1)*
and also	$val_{\mathcal{D},\beta}(sh(P, Q \circ F, R \circ F))$	$=$	$val_{\mathcal{D},\beta}(Q \circ F)$	*(Def. 1)*
Case 2: $val_{\mathcal{D},\beta}(P) = 1$				
then	$val_{\mathcal{D},\beta}(sh(P, Q, R)) \circ F$	$=$	$val_{\mathcal{D},\beta}(R \circ F)$	*(Def. 1)*
and also	$val_{\mathcal{D},\beta}(sh(P, Q \circ F, R \circ F))$	$=$	$val_{\mathcal{D},\beta}(R \circ F)$	*(Def. 1)*

The basic mechanism for manipulating Shannon expressions is the replacement of leaves:

Definition 5 (Replacement of Leaves) Let $\mathcal{A}, \mathcal{C} \in \mathrm{SH}, \mathcal{B} \in \{\mathbf{1}, \mathbf{0}\}$; the replacement of all leaves $\mathcal{B}$ in $\mathcal{A}$ by $\mathcal{C}$ is recursively defined as follows:

$$\mathcal{A}\left[\frac{\mathcal{B}}{\mathcal{C}}\right] \stackrel{\mathrm{def}}{=} \begin{cases} sh(A, B\left[\frac{\mathcal{B}}{\mathcal{C}}\right], C\left[\frac{\mathcal{B}}{\mathcal{C}}\right]) & \text{if } \mathcal{A} = sh(A,B,C) \\ \mathcal{C} & \text{if } \mathcal{A} = \mathcal{B} \\ \mathcal{A} & \text{otherwise} \end{cases}$$

Similarly to the standard way of dealing with substitutions in first-order logic, we assume that $\mathcal{G}\left[\frac{\mathcal{A}}{\mathcal{A}'}, \frac{\mathcal{B}}{\mathcal{B}'}\right]$ (where $\mathcal{A} \neq \mathcal{B}$) is performed simultaneously, so that an expression like $\mathcal{G}\left[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathbf{0}}\right]$ makes sense

Later on we will consider Shannon expressions that may contain at the first position of the Shannon operator not only atomic formulæ but more complex expressions that may themselves contain the constants $\mathbf{0}$ and $\mathbf{1}$. These "interior" constants should not be affected by replacement operations. Notice, that this guaranteed by the above definition.

The following proposition bridges the gap between replacing leaves and logical operations:

Proposition 6 Let $\mathcal{A}, \mathcal{B} \in \mathrm{SH}$; then

$$(REP_\neg)\ \neg\mathcal{A} \Leftrightarrow \mathcal{A}\left[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathbf{0}}\right] \qquad (REP_\vee)\ \mathcal{A}\vee\mathcal{B} \Leftrightarrow \mathcal{A}\left[\frac{\mathbf{0}}{\mathcal{B}}\right] \qquad (REP_\wedge)\ \mathcal{A}\wedge\mathcal{B} \Leftrightarrow \mathcal{A}\left[\frac{\mathbf{1}}{\mathcal{B}}\right]$$

The proofs are again quite straightforward; here is an example:

Proof Case $REP_\vee$ of Proposition 6(induction over the structure of a Shannon graph $\mathcal{A}$)

1. $\mathcal{A} \in \{\mathbf{1}, \mathbf{0}\}$**:** clear.

2. $\mathcal{A} = sh(P,Q,R)$**:**

$$\begin{array}{lcll} \mathcal{A}\left[\frac{\mathbf{0}}{\mathcal{B}}\right] & \Leftrightarrow & sh(P, Q\left[\frac{\mathbf{0}}{\mathcal{B}}\right], R\left[\frac{\mathbf{0}}{\mathcal{B}}\right]) & \text{Def. 5} \\ & \Leftrightarrow & sh(P, Q\vee\mathcal{B}, R\vee\mathcal{B}) & \text{Ind. Hyp.} \\ & \Leftrightarrow & sh(P,Q,R)\vee\mathcal{B} & \text{Def. 1} \end{array}$$

The following proposition shows that Shannon expressions are a normal form for formulæ.[6]:

Proposition 7 (Shannon Expressions are a Logical Basis) For every quantifier-free formula $\phi \in \mathcal{L}$, there is a Shannon expression $\mathcal{G} \in \mathrm{SH}$, such that $\phi \Leftrightarrow \mathcal{G}$.

Proof (Proposition 7) We will define a function *f2Sh* that maps quantifier-free formulas in Shannon expressions such that $F \Leftrightarrow f2Sh(F)$.

[6] This property of the *sh*-operator is discussed in detail by Church [12, §24, p. 129ff].

Definition 8 If $F(\bar{X})$ is a quantifier-free first-order formula (the free variables in F are denoted by "$\bar{X}$"), then *f2Sh* maps $F(\bar{X})$ to a Shannon graph $\mathcal{F}(\bar{X})$:

$$f2Sh(F) \stackrel{\text{def}}{=} \begin{cases} sh(F, \mathbf{0}, \mathbf{1}) & \textit{iff } F \in \mathcal{L}_{At} \\ sh(A, \mathbf{1}, \mathbf{0}) & \textit{iff } F = \neg A, A \in \mathcal{L}_{At} \\ f2Sh(A)\left[\frac{\mathbf{1}}{f2Sh(B)}\right] & \textit{iff } F = A \wedge B \\ f2Sh(A)\left[\frac{\mathbf{0}}{f2Sh(B)}\right] & \textit{iff } F = A \vee B \\ f2Sh(\neg A \vee \neg B) & \textit{iff } F = \neg(A \wedge B) \\ f2Sh(\neg A \wedge \neg B) & \textit{iff } F = \neg(A \vee B) \\ f2Sh(A) & \textit{iff } F = \neg\neg A \end{cases}$$

For a formula $F \in \mathcal{L}$, we will refer to the graph $f2Sh(F)$ as the **initial graph for** F. This will often be abbreviated by simply using a calligraphic symbol: $\mathcal{F}$ denotes the initial graph for F. Figure 2 on page 5 shows the initial graphs for $a \wedge b$, $\neg a$, and $a \wedge b \wedge \neg a$.

Corollary 9 or every quantifier-free formula F: $F \Leftrightarrow f2Sh(F)$
(This follows from the definition of *f2Sh* and Proposition 6)

It is straightforward how *f2Sh* can be extended to handle other logical connectives.We will come back to this in a later section and assume for the rest of this section that $\mathcal{L}$ only contains "$\neg, \wedge$, and $\vee$" as propositional connectives.

2.1 Trees vs Graphs

Up to this point it was unclear why we speak of Shannon graphs and not of Shannon trees; the function *f2Sh* defined above actually maps a formula to a tree because it simply returns a *sh*-expression. This often results in very huge trees, even for smallish formulæ. Replacing **1**/**0**-leaves will result in multiple occurrences of the same subtree if more than one leaf had been replaced. We can get a much more compact representation if we use *structure sharing* and keep only one copy of the subtrees that is inserted. In this case, the resulting Shannon graphs are directed, acyclic graphs (DAGs).

Technically, this can be achieved as follows: we start by having exactly one node for **1**-leaves, and exactly one node for **0**-leaves; when *f2Sh* processes a literal and creates a new node (i.e. builds a new *sh*-expression) with one of those nodes as successors, an edge to them is introduced instead of using a copy. When replacing **1**/**0**-leaves for combining graphs, the *edges* to the **1**/**0**-leaves are altered instead of replacing the nodes themselves. The resulting graphs are reduced in the sense that they do not contain different copies of isomorphic subgraphs. This may easily be proved by induction. Assume we know already that the claim is true for graphs $\mathcal{A}$ and $\mathcal{B}$ and $\mathcal{G}$ results from $\mathcal{A}$ by replacing its **1**-leave by $\mathcal{B}$. If $\mathcal{A}_1, \mathcal{A}_2$ are different copies of isomorphic graphs in $\mathcal{G}$ then by induction hypothesis the roots $\mathbf{n}_1$ and $\mathbf{n}_2$ of these graphs will not lie in the same original graph. We have e.g. $\mathbf{n}_1$ in $\mathcal{A}$ and $\mathbf{n}_2$ in $\mathcal{B}$.

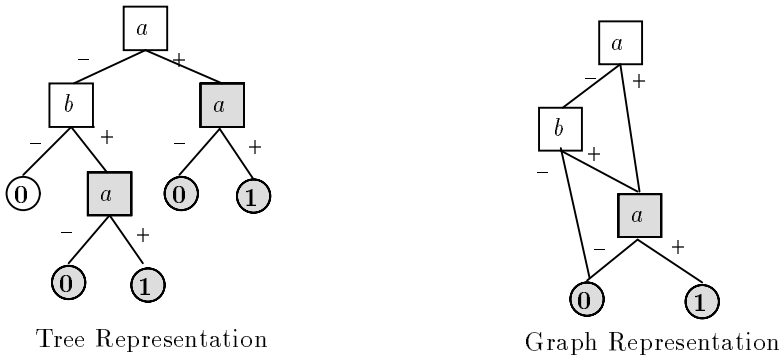


Figure 3: Tree vs. Graph Representation of Shannon graphs

By construction of $\mathcal{G}$ the length of the maximal **1**-path in $\mathcal{A}_1$ will be strictly greater than the length of the maximal **1**-path in $\mathcal{A}_2$ contradicting isomorphism between $\mathcal{A}_1$ and $\mathcal{A}_2$.

Another nice effect of the above way to implement *f2Sh* is that the size of the resulting graph, i.e. number of nodes + number of edges, is linear in the size of the input formula in negation normal form[7]. For each occurrence of an atom in the input formula, there is exactly one non-terminal node in the graph.

Figure 3 depicts a graph for $(a \vee b) \wedge a$ computed in this way. The two marked subtrees in the tree representation appear only once in the graph representation. This results from the different treatment of the conjunction in the formula: if the **1**-leaves of $f2Sh(a \vee b)$ are replaced with $f2Sh(a)$ we get a tree, and if the *edges* are replaced we get a graph.

In the sequel we assume that *f2Sh* builds a graph. For the following theoretical considerations it does not make any difference whether we think of trees or graphs, so we will use whatever is more convenient. But remember, that in the implementation *f2Sh* really builds a graph.

3 Paths in Shannon Expressions

We have seen how to transform an arbitrary formula into a logically equivalent Shannon expression and now take up the task to determine whether a given Shannon expression is consistent. The key to such a test is the consideration of paths in the tree corresponding to the expression: a path consists of signed nodes, where the sign is determined by the node's outgoing edge. A path can be understood as a (partial) interpretation of the atoms occurring in a graph. We start with a definition that might seem slightly at first sight:

[7] The pigeon-hole formulæ for seven pigeons and six holes, for instance, yield a tree with about 10^{43} nodes. If *f2Sh* builds a graph, the number of nodes is reduced to less than 300.

Definition 10 (Paths) A **path** π in a Shannon expression $\mathcal{G}$ is a sequence of signed subformulæ of $\mathcal{G}$ and denoted by an expression of the form $[v_1\phi_1, \ldots, v_n\phi_n]$, such that there is a sequence of nodes $[\mathbf{n}_1, \ldots, \mathbf{n}_n]$ in $\mathcal{G}$ satisfying:

1. $\forall i \in \{1, \ldots, n\}:\ v_i \in \{-, +\}$, ϕ_i is the label of $\mathbf{n}_i$,
2. $\forall j \in \{1, \ldots, n-1\}:$ there is an edge from $\mathbf{n}_i$ to $\mathbf{n}_{i+1}$ labeled by v_i.

A path is said to **start** at formula ϕ_1 (or: node $\mathbf{n}_1$), and said to **end at** formula ϕ_n (or: node $\mathbf{n}_n$). We use $\bot(\pi)$ to denote the formula where the path π ends.

As a consequence of this definition there is no path in the expressions $\mathbf{1}, \mathbf{0}$.

As an example, consider the graph $\mathcal{A}$ for "$a \wedge b$" in Figure 2: the path $[+a, +b]$ leads from the root to the "$\mathbf{1}$"-leaf of $\mathcal{A}$. When dealing with propositional logic, the components of a path will always be signed propositional variables.

Definition 11 (Truth value of paths) A path $[v_1\phi_1, \ldots, v_n\phi_n]$ is regarded as the conjunction of its signed formulas; i.e.

$$val_{\mathcal{D},\beta}\left([v_1\phi_1, \ldots, v_n\phi_n]\right) \stackrel{\text{def}}{=} val_{\mathcal{D},\beta}\left(\bigwedge_{i\in\{1,\ldots,n\}} v_i\phi_i\right), \text{ where } v\phi \stackrel{\text{def}}{=} \begin{cases} \phi & \text{if } v = + \\ \neg\phi & \text{if } v = - \end{cases}$$

An empty path –as an empty conjunction– has the truth value $\mathbf{1}$. A path π is said to be satisfiable, if there is an interpretation, such that: for all variable assignments β, $val_{\mathcal{D},\beta}(\pi) = \mathbf{1}$. A path is unsatisfiable, if it is not satisfiable.

The following definition introduces a notation for referring to paths that lead to the leaves of Shannon expressions.

Definition 12 (0-paths and 1-paths) A **1-path** in a Shannon expression $\mathcal{G}$ is a path of maximal length in $\mathcal{G}$ with $\bot(\pi) = \mathbf{1}$; thus, it starts at the root and ends in a $\mathbf{1}$–leaf. **0–paths** are defined analogously.
We use the notation:

$$\Pi^{\mathbf{1}}_{\mathcal{G}} \stackrel{\text{def}}{=} \{\pi \mid \pi \text{ is a } \mathbf{1}\text{-path in } \mathcal{G}\} \quad \text{and} \quad \Pi^{\mathbf{0}}_{\mathcal{G}} \stackrel{\text{def}}{=} \{\pi \mid \pi \text{ is a } \mathbf{0}\text{-path in } \mathcal{G}\}$$

Note that for one-node graphs this yields:

$$\Pi^{\mathbf{1}}_{\mathbf{1}} \stackrel{\text{def}}{=} \{\emptyset\} \quad \Pi^{\mathbf{1}}_{\mathbf{0}} \stackrel{\text{def}}{=} \emptyset \quad \Pi^{\mathbf{0}}_{\mathbf{1}} \stackrel{\text{def}}{=} \emptyset \quad \Pi^{\mathbf{0}}_{\mathbf{0}} \stackrel{\text{def}}{=} \{\emptyset\}$$

The reader might have noticed that paths are an analogue of branches in free-variable semantic tableau; likewise to tableaux, we define

Definition 13 (Closed Paths) A path π is said to be **closed** iff there is a substitution σ such that $\pi\sigma$ contains at least two identical elements with complementary signs. Such a σ is called **closing substitution** for π.

A path that is not closed is said to be **open**.

Example: the path $[+a, +b, -a]$ of $\mathcal{C}$ in Figure 2 is closed.

If we recall the definition of the truth value of paths, it is clear that the following holds:

$$\text{a path } \pi \text{ is closed} \quad \Rightarrow \quad \pi \text{ is unsatisfiable}$$

The observation can be strengthened for propositional logic: the implication then is an equivalence, since the first argument of *sh*-expressions, and therefore the components of paths, are always atomic[8]. We stick to the weaker form because the subsequent treatment of first-order logic requires also non-atomic nodes; in this case, the equivalence is not valid any more, but the implication still is.

Definition 14 (Closed Shannon expressions) A first-order Shannon expression $\mathcal{G}$ is called **closed** iff

1. it consists of only one node labeled with **0**, or
2. there is a single closing substitution σ for all **1**-paths of $\mathcal{G}$. σ is called a **closing substitution** for $\mathcal{G}$.

An expression is said to be **open** if it is not closed.

Example: in Figure 2, $\mathcal{A}, \mathcal{B}$ are open, $\mathcal{C}$ is closed.

Analogously to paths, a satisfiable expression is open and a closed expression is unsatisfiable. However, there is actually a closer connection between Shannon expressions and their paths than this. The following definitions helps to formalize it:

Definition 15 (Truth value of Sets of Paths) To compute the truth value of a set of paths we regard the set as a disjunction: Let $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, then

$$val_{\mathcal{D},\beta}(\Pi) \stackrel{\text{def}}{=} val_{\mathcal{D},\beta}\left(\bigvee_{i \in \{1,\ldots,n\}} \pi_i\right)$$

The empty disjunction is defined to be false, as usual.

So, sets of paths are regarded as a disjunction of conjunctions. Here is the central proposition of this section that relates sets of **1**-paths and **0**-paths in a Shannon expression to the expression itself:

Proposition 16 Let $\mathcal{G}$ be a Shannon expression, then $\quad \Pi^{\mathbf{1}}_{\mathcal{G}} \Leftrightarrow \mathcal{G} \quad$ and $\quad \Pi^{\mathbf{0}}_{\mathcal{G}} \Leftrightarrow \neg\mathcal{G}$

To facilitate the proof we use the following notation:
Let $\pi_1 = [k_1, \ldots, k_n]$ and $\pi_2 = [l_1, \ldots, l_m]$ be paths, then

$$\pi_1 \odot \pi_2 \stackrel{\text{def}}{=} [k_1, \ldots, k_n, l_1, \ldots, l_m]$$

"$\odot$" is carried forward to sets of paths Π_1 and Π_2, by defining:

$$\Pi_1 \odot \Pi_2 \stackrel{\text{def}}{=} \{\pi_1 \odot \pi_2 \mid \pi_1 \in \Pi_1 \text{ and } \pi_2 \in \Pi_2\}$$

For proving Proposition 16 it helps to know that

[8] This is not true for non-atomic nodes: consider a path $[+(a \wedge b), -b, -a]$; it is inconsistent but not closed.

Corollary 17 Let Π be a path set and π be a path, then

1. $\Pi \odot \emptyset = \emptyset \odot \Pi = \emptyset$ 2. $val_{\mathcal{D},\beta}(\{\pi\} \odot \Pi) = val_{\mathcal{D},\beta}(\pi \wedge \Pi)$.

Proposition 16 can now be shown conveniently by induction over the structure of a Shannon expression $\mathcal{G}$:

Proof (Proposition 16)
Basis.

$\mathcal{G} = \mathbf{0}$: $val_{\mathcal{D},\beta}(\Pi^{\mathbf{1}}_{\mathbf{0}}) = \mathbf{0} = val_{\mathcal{D},\beta}(\mathbf{0})$, since $\Pi^{\mathbf{1}}_{\mathbf{0}} = \emptyset$, which is defined to be false.
$val_{\mathcal{D},\beta}(\Pi^{\mathbf{0}}_{\mathbf{0}}) = val_{\mathcal{D},\beta}(\{[]\}) = \mathbf{1} = val_{\mathcal{D},\beta}(\neg\mathbf{0})$, since $val_{\mathcal{D},\beta}([]) = \mathbf{1}$ by Definition 11.

$\mathcal{G} = \mathbf{1}$: $val_{\mathcal{D},\beta}(\Pi^{\mathbf{1}}_{\mathbf{1}}) = val_{\mathcal{D},\beta}(\{[]\}) = \mathbf{1} = val_{\mathcal{D},\beta}(\mathbf{1})$.
$val_{\mathcal{D},\beta}(\Pi^{\mathbf{0}}_{\mathbf{1}}) = \mathbf{0} = val_{\mathcal{D},\beta}(\neg\mathbf{1})$.

Recursion.

$$\begin{aligned}
val_{\mathcal{D},\beta}\left(\Pi^{\mathbf{1}}_{sh\,(A,B,C)}\right) &= val_{\mathcal{D},\beta}\left(\left(\{[-A]\} \odot \Pi^{\mathbf{1}}_{B}\right) \cup \left(\{[+A]\} \odot \Pi^{\mathbf{1}}_{C}\right)\right) \\
&= val_{\mathcal{D},\beta}\left(\left(\neg A \wedge \Pi^{\mathbf{1}}_{B}\right) \vee \left(A \wedge \Pi^{\mathbf{1}}_{C}\right)\right) \quad \text{(Cor. 17 and Def. 11)} \\
&= val_{\mathcal{D},\beta}\left((\neg A \wedge B) \vee (A \wedge C)\right) \quad \text{(induction basis)} \\
&= val_{\mathcal{D},\beta}\left(sh(A,B,C)\right).
\end{aligned}$$

$$\begin{aligned}
val_{\mathcal{D},\beta}\left(\Pi^{\mathbf{0}}_{sh\,(A,B,C)}\right) &= val_{\mathcal{D},\beta}\left(\left(\{[-A]\} \odot \Pi^{\mathbf{0}}_{B}\right) \cup \left(\{[+A]\} \odot \Pi^{\mathbf{0}}_{C}\right)\right) \\
&= val_{\mathcal{D},\beta}\left(\left(\neg A \wedge \Pi^{\mathbf{0}}_{B}\right) \vee \left(A \wedge \Pi^{\mathbf{0}}_{C}\right)\right) \\
&= val_{\mathcal{D},\beta}\left((\neg A \wedge \neg B) \vee (A \wedge \neg C)\right) \quad \text{(induction basis)} \\
&= val_{\mathcal{D},\beta}\left(\neg sh(A,B,C)\right) \quad \text{(Proposition 4)}.
\end{aligned}$$

To summarize, we have seen that

1. **1**-paths in a Shannon expressions are a disjunctive normal form for the expression, and
2. **0**-paths are a disjunctive normal form for the *negated* Shannon expression.

Thus, **1**-paths represent models, and **0**-paths counter models of a Shannon expression.

The result of this section can easily be used to check whether a propositional formula F is inconsistent or a tautology:

Proposition 18

1. If there is no open **1**-path in $f2Sh(F)$ then F is inconsistent.
2. F is a tautology if there is no open **0**-path in $f2Sh(F)$.

4 Shannon Graphs with Free Variables

The presented formalism can also be applied to state a complete calculus for first-order formulæ in prenex normal form. This is discussed in detail elsewhere [22] and we will only give a brief account here.

Proposition 19 Assume, that $\bar{x}$ stands for a finite list of variables. If $\forall \bar{x} F(\bar{x})$ is unsatisfiable, then there exists some $k \in I\!N$ and a grounding substitution σ, such that the conjunction $(F(\bar{x}_0) \wedge \ldots \wedge F(\bar{x}_k))\sigma$ is unsatisfiable.
(σ maps all variables to ground terms, i.e., to terms from the Herbrand-universe $\mathcal{U}_F$. Each $\bar{x}_i$ is a *new* list of variables $x_{i,1}, \ldots, x_{i,n}$ distinct from all $\bar{x}_j$ with $j < i$.)

It is possible to apply this well-known proposition to get a straightforward first-order proof procedure for Shannon graphs: we start by constructing a logically equivalent initial graph $f2Sh(F(\bar{x})) = \mathcal{F}(\bar{x})$. Then, an iteration is used to subsequently generate the above conjunction by introducing variants of $f2Sh(F(\bar{x}))$.

Such an iteration step will be called an **extension** in the sequel. More formally, the process is described in:

Definition 20 (Maximal Extensions of Shannon Graphs) Let $F(\bar{x})$ be a quantifier-free formula; we recursively define a sequence $\mathcal{F}_0, \mathcal{F}_1, \ldots$ of Shannon graphs:

$$\mathcal{F}_0(\bar{x}_0) := f2Sh(F(\bar{x}_0)) \quad (1)$$

$$\mathcal{F}_{i+1}(\bar{x}_0, \ldots, \bar{x}_{i+1}) := \mathcal{F}_i(\bar{x}_0, \ldots, \bar{x}_i) \left[\frac{\mathbf{1}}{\mathcal{F}_0(\bar{x}_{i+1})} \right] \quad (2)$$

($\mathcal{F}_0(\bar{x}_{i+1})$ differs from $\mathcal{F}_0$ in that all free variables in $\mathcal{F}_0$ have been renamed.)

$\mathcal{F}_i$ is called the i^{th} extension of $f2Sh(F(\bar{x}_0))$

Clearly, the above iteration "implements" the derivation of the conjunction of Proposition 19; we can therefore guarantee that at some point the iteration actually delivers a proof if there is one:

Proposition 21 If $F(\bar{x})$ is a quantifier-free first-order formula and $\forall \bar{x} F(\bar{x})$ is unsatisfiable, then there is some $i \in I\!N$ such that $\mathcal{F}_i$ (as defined in 20) is closed.

In definition 20 every **1**-node is replaced by $\mathcal{F}_0(\bar{x}_{i+1})$ with the same tuple of fresh variables. This is more restrictive than necessary as the next proposition will show.

Definition 22 (Disjoint Extensions of Shannon Expressions) Let a Shannon expression $\mathcal{F}(\bar{x})$ be given. The Shannon expression $\mathcal{G}$ obtained from $\mathcal{F}(\bar{x})$ by replacing each **1**-node by a variant $\mathcal{F}_0(\bar{Y}_j)$ of $\mathcal{F}_0(\bar{x})$ is called a disjoint extension of $\mathcal{F}$ by $\mathcal{F}_0$. The variables $\bar{Y}_j$ are different for each **1**-node to be replaced and also different from all variables that occur in $\mathcal{F}$. Again we recursively define a sequence $\mathcal{F}_0^d, \mathcal{F}_1^d, \ldots$ of Shannon expressions:

$$\mathcal{F}_0^d(\bar{x}) := f2Sh(F(\bar{x})) \quad (3)$$

$$\mathcal{F}_{i+1}^d \quad \text{is a disjoint extension of } \mathcal{F}_i^d \text{ by } \mathcal{F}_0^d. \quad (4)$$

Proposition 23 If $F(\bar{x})$ is a quantifier-free first-order formula and $\forall \bar{x} F(\bar{x})$ is unsatisfiable, then there is some $i \in I\!N$ such that $\mathcal{F}_i^d$ (as defined in 22) is closed.

Proof (Proposition 23) It suffices to show that a disjoint extension $\mathcal{G}$ of $\mathcal{F}$ by $\mathcal{F}_0(\bar{x})$ satisfies:

$$\mathcal{G} \Leftrightarrow \mathcal{F} \wedge \forall \bar{x} \mathcal{F}_0(\bar{x})$$

By proposition 16 we have $\mathcal{F} \Leftrightarrow \pi_1 \vee \ldots \vee \pi_k$, where $\pi_1, \ldots, \pi_k$ are all **1**-paths in $\mathcal{F}$ and as before a path is identified with the conjunction of its formulas. By the same proposition and since $\mathcal{G}$ is a disjoint extension of $\mathcal{F}$ be $\mathcal{F}_0$ we have

$$\mathcal{G} \Leftrightarrow (\pi_1 \wedge \mathcal{F}_0(\bar{Y}_1)) \vee \ldots \vee (\pi_k \wedge \mathcal{F}_0(\bar{Y}_k))$$

This equivalence is implicitly universally quantified. Since the Y-variables do not occur on the left hand side we obtain:

$$\mathcal{G} \Leftrightarrow \forall(\bar{Y}_1) \ldots (\bar{Y}_k)((\pi_1 \wedge \mathcal{F}_0(\bar{Y}_1)) \vee \ldots \vee (\pi_k \wedge \mathcal{F}_0(\bar{Y}_k)))$$

Since the tuples $(\bar{Y}_i)$, $(\bar{Y}_j)$ are disjoint for $i \neq j$ and do not occur in any π_i we get furthermore

$$\mathcal{G} \Leftrightarrow \forall(\bar{Y}_1)(\pi_1 \wedge \mathcal{F}_0(\bar{Y}_1)) \vee \ldots \vee \forall(\bar{Y}_k)(\pi_k \wedge \mathcal{F}_0(\bar{Y}_k))$$

This may require a little thought. The reader may start be noticing that under the above conditions regarding the occurrence of variables the formula

$$(\forall Y_1 \forall Y_2 (\pi_1 \wedge F(Y_1)) \vee (\pi_2 \wedge F(Y_2))) \Leftrightarrow (\forall(Y_1 \pi_1 \wedge F(Y_1))) \vee (\forall Y_2 (\pi_2 \wedge F(Y_2)))$$

is a tautology. Back to the main argument: renaming of bound variables and the distributive law now yields:

$$\mathcal{G} \Leftrightarrow (\pi_1 \vee \ldots \vee \pi_k) \wedge \forall \mathcal{F}_0(\bar{x}), \text{ and thus also } \mathcal{G} \Leftrightarrow \mathcal{F} \wedge \forall \mathcal{F}_0(\bar{x})$$

Performing disjoint extensions makes it easier to close a graph, since we may have different variable bindings in different copies of the extended graphs.

We will shortly consider an example:

Problem 24

Axm 24.1: $\forall x \neg p(x)$

Axm 24.2: $p(a) \vee p(b) \vee p(c)$

Figure 4 shows the initial graph and its first disjoint extension; the latter is closed with $\sigma = [x/a, x_1/c, x_2/b]$. An additional extension would have been required with the former way of extending, since x_2 and x_3 would have been equal to x_1.

Figure 4 is also a good example for the disadvantage of maximal extensions: using a variant of the complete initial graph causes much redundancy. The nodes $p(a), p(b)$ and $p(c)$ in the extension do not contribute to the proof. In this case it is not very relevant because the nodes actually need not be visited for closing all **1**-paths. In general, however, such extensions tend to considerably complicate the proof search.

The method of disjoint extensions can be further optimized if the formula we wish to test for inconsistency is a conjunction, as it is in the example above: in such cases, it is sufficient to use elements of the conjunction for extensions. Correctness and completeness of such optimized disjoint extensions is easily obtained by adapting the proofs above. In our example, this would mean that it is sufficient to use variants of a Shannon graphs for $p(x)$ in extensions. This would remove all redundancy in this case, but in general it does not: quantifiers can appear deeply nested in formulæ, and the best way to avoid introducing redundancy is to avoid extending the scope of quantifiers. The next section shows how this can be achieved.

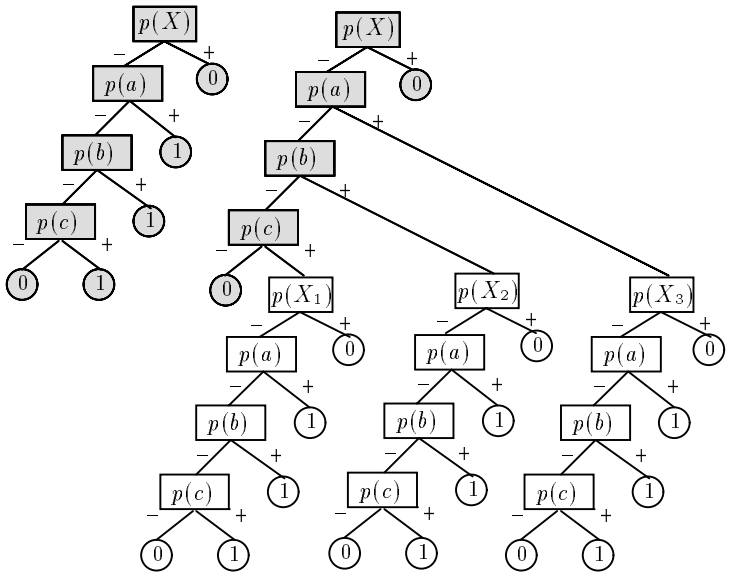


Figure 4: Shannon Graphs for Problem 24

5 Quantified Shannon Graphs

We will extend the definition of Shannon expressions by allowing universal quantifiers guided by the use of γ-formulæ in tableau calculus. We could have equally well allowed existential quantifiers, but chosen not to do so, since existential quantifiers are best handled by Skolemization during preprocessing.

Problem 25 Axm 25: $(\neg p(a) \vee \neg p(b)) \wedge (\forall x\, p(x))$

We will briefly outline the approach at an example before formalizing it. Figure 5 shows on the left-hand side of its upper half a closed tableau for the inconsistent formula of Problem 25; the proof is easily obtained by applying a γ-formula $\forall x\, p(x)$ twice, after decomposing the original formula. The bottom left Shannon graph uses a similar principle for treating quantified formulæ: the idea is to switch from "flat" graphs to "nested" graphs by allowing that the nodes contain another Shannon graph inside. Such graphs inside nodes represent a formula within the scope of a quantifier of the input formula.

The proposed procedure for carrying out the proof search is the following: when trying to close a graph, we do not immediately descend into the quantified subgraphs, but just note in the path which exit of the subgraph was chosen. (From the tableau-perspective, this means that we remember having seen a γ-formula on a branch, which can be expanded if desired.) If we arrive with a consistent path at a **1**-leaf, we choose an arbitrary quantified subgraph whose node we left through its *positive* edge and insert if for the **1**-leave.

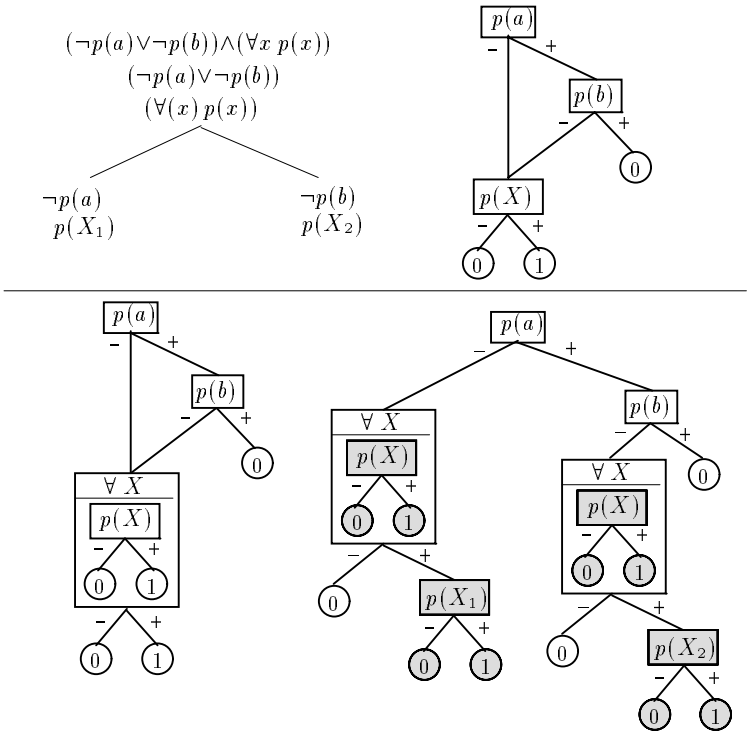


Figure 5: Closed Tableau and Shannon Graphs for Problem 25 ($\sigma = [x_1/a, x_2/b]$)

Coming back to our example shown in Figure 5, we note that there are two open paths to **1**-leaves in the Shannon graph, namely

$$[+p(a), -p(b), +(\forall x\ sh(p(x), 0, 1))] \quad and \quad -p(a), +(\forall x\ sh(p(x), 0, 1))].$$

They represent two potential models, wherein $\forall x\, p(x)$ is assumed to be true. $p(x)$ must therefore also be true for all values of x. Similarly to free-variable tableaux, we add $p(x)$ to the paths, where x is a free variable that may be instantiated, later. Technically, this is achieved by replacing the **1**-leaf by the quantified subgraph — after renaming the quantified variable and dropping the surrounding quantifier (see the bottom-right graph in the figure). Now, both paths become contradictory if we substitute a and b for x_1 and x_2, respectively[9]. The substitution can be found by searching the path for unifiable and complementary atoms.

Two important points must be considered for quantified Shannon graphs:

1. *we must be allowed to re-use the same universally quantified subgraph arbitrarily (but finitely) often*

[9] This contradicts a bit to the idea of replacing the **1**-leaf by only one subgraph, since we use different substitutions for both paths. In practice, however, we will perform extensions relative to paths, as will be seen later.

This is the counterpart to the fact that γ-formulæ in tableaux can be expanded as often as desired. In general it is not possible to predict an upper bound for this, unless the underlying domain is finite.

2. *we are not allowed to perform extensions with quantified subgraphs that appear with a negative prefix in a path.*
   To see this, assume we had a path to a **1**-leaf containing $-(\forall x\ sh(p(x),0,1))$; semantically, this means that $\forall x\ sh(p(x),0,1)$ is false in the present partial model; it is therefore not correct to extend the **1**-leaf with $p(x)$!

Shannon graphs possibly containing $\forall$-subgraphs are called **quantified Shannon graphs**.

5.1 Shannon Graphs with Selective Extensions

The definition of Shannon graphs (Definition 2, page 5) is extended for representing quantifiers in the following way:

Definition 26 (quantified Shannon graphs) The set of *quantified Shannon graphs* is denoted by $\mathrm{SH}_{\boxed{\forall}}$ and defined as the smallest superset of SH such that

1. $\mathrm{SH} \subset \mathrm{SH}_{\boxed{\forall}}$
2. *if* $\mathcal{A},\mathcal{B},\mathcal{C} \in SH_{\boxed{\forall}}$ *then* $sh((\forall x_1,\ldots,x_n\,\mathcal{A}),\mathcal{B},\mathcal{C})$ *is in SH, where* $x_1,\ldots,x_n$ *are free variables occurring in* $\mathcal{A}$

The semantics of a universal quantifier preceding a graph is handled in the standard way of first-order logic by mapping the variable assignment at position x to all elements of the domain. For constructing an initial graph for a formula F, Definition 8 of *f2Sh* on page 8 is altered such that the new function (denoted by $f2Sh_{\boxed{\forall}}$) maps an arbitrary first-order formula into a quantified Shannon graph. Readers familiar with tableau calculus will notice that our translation of existential quantifiers corresponds to the "liberalized δ-rule" [3]. This has the advantage that the Skolem terms become very compact. We will assume that $\bar{x}$ stands for a finite list of variables $x_1,\ldots,x_m$.

Definition 27

$$f2Sh_{\boxed{\forall}}(F) = \begin{cases} \textit{[include all rules from f2Sh (Def. 8)]} & \\ sh(\forall\bar{x}\ f2Sh_{\boxed{\forall}}(A),\mathbf{0},\mathbf{1}) & \textit{if } F = \forall\bar{x}\ A \\ f2Sh_{\boxed{\forall}}(A\left[\frac{x_1,\ldots,x_n}{f_1(\bar{y}),\ldots,f_n(\bar{y})}\right]) & \text{if } F = \exists\bar{x}\ A, \text{ where } \bar{y} \text{ are free variables in } F \text{ and } f_1,\ldots,f_n \text{ are new function symbols} \\ f2Sh_{\boxed{\forall}}(\exists\bar{x}\ \neg A) & \textit{if } F = \neg(\forall\bar{x}\ A) \\ f2Sh_{\boxed{\forall}}(\forall\bar{x}\ \neg A) & \textit{if } F = \neg(\exists\bar{x}\ A) \end{cases}$$

Next we define selective extensions; these will be carried out w.r.t. **1**-paths: we do not extend all **1**-paths, but allow more focussed extension which apply to a single

path, only. We must therefore define this operation on Shannon expressions, rather than on Shannon graphs[10].

Definition 28 (Selective Extension) Let $\mathcal{G} \in \mathrm{SH}_{\boxed{\forall}}$, π a path in $\mathcal{G}$ with $\perp(\pi) = \mathbf{1}$ and $\forall \bar{x}\, \mathcal{A}$ a universal Shannon expression occurring positively on π. Then a selective extension of $\mathcal{G}$ is obtained by replacing the leaf node $\perp(\pi)$ with $\mathcal{A}\left[\frac{\bar{x}}{\bar{x}'}\right]$ where $\bar{x}'$ are new variables.
We sometimes use the notation: $\mathcal{G}\left[\frac{\perp(\pi)}{\mathcal{A}\left[\frac{\bar{x}}{\bar{x}'}\right]}\right]$

Proposition 29 (Correctness of selective extensions) Selective extensions, as defined in Definition 28 preserve satisfiability.

Proof (Proposition 29) We show: if $\mathcal{G} \in SH_{\boxed{\forall}}$ is satisfiable then a selective extension $\mathcal{F} = \mathcal{G}\left[\frac{\perp(\pi)}{\mathcal{A}\left[\frac{\bar{x}}{\bar{x}'}\right]}\right]$ where $\forall \bar{x}\, \mathcal{A}$ occurs positively on the path π, is also satisfiable. More precisely we show that $val_{\mathcal{D},\beta}(\mathcal{G}) = val_{\mathcal{D},\beta}(\mathcal{F})$

By proposition 16 $\mathcal{G}$ is logically equivalent to the disjunction of its $\mathbf{1}$-paths. If $val_{\mathcal{D},\beta}(\mathcal{G}) = 0$ then $val_{\mathcal{D},\beta}(\nu) = 0$ for all path ν of $\mathcal{G}$. Since any path ν' in $\mathcal{F}$ contains a path ν from $\mathcal{G}$ as an initial segment we have also $val_{\mathcal{D},\beta}(\nu') = 0$ for all ν' in $\mathcal{F}$ and thus $val_{\mathcal{D},\beta}(\mathcal{F}) = 0$.
If on the other hand $val_{\mathcal{D},\beta}(\mathcal{G}) = 1$ then we have $val_{\mathcal{D},\beta}(\nu) = 0$ for at least one path ν. If $\nu \neq \pi$, then we immediately get $val_{\mathcal{D},\beta}(\mathcal{F}) = 1$. In case $\nu = \pi$ the assumption entails $val_{\mathcal{D},\beta}(\forall \bar{x}\, \mathcal{A}) = 1$. Thus no matter what elements the function β assigns to the variables $\bar{x}'$ we have $val_{\mathcal{D},\beta}(\mathcal{A}\left[\frac{\bar{x}}{\bar{x}'}\right]) = 1$ and thus $val_{\mathcal{D},\beta}(\pi \cup \{\mathcal{A}\left[\frac{\bar{x}}{\bar{x}'}\right]\}) = 1$ and $val_{\mathcal{D},\beta}(\mathcal{F}) = 1$.

Proposition 30 (Completeness of selective extensions) For every inconsistent $f2Sh_{\boxed{\forall}}(A) = \mathcal{G} \in \mathrm{SH}_{\boxed{\forall}}$ there is a finite sequence of selective extensions ending in $\mathcal{B}$ and a substitution σ such that every $\mathbf{1}$-path in $\mathcal{B}$ is closed.

Proof (Proposition 30) Assume $\mathcal{G} \in \mathrm{SH}_{\boxed{\forall}}$ is inconsistent and the claim of the proposition is not true. We obtain thus an infinite sequence of elements in $\mathrm{SH}_{\boxed{\forall}}$ $\mathcal{F}_0, \ldots, \mathcal{F}_n, \ldots$ with $\mathcal{F}_0 = \mathcal{G}$ and each $\mathcal{F}_{i+1}$ a selective extension of $\mathcal{F}_i$, such that there is no closing substitution for any $\mathcal{F}_n$. We may arrange this sequence in such a way that any formula $\forall \bar{x}\, \mathcal{A}$ has been used infinitely often for a selective extension to any path π on which $\forall \bar{x}\, \mathcal{A}$ occurs positively. Let $\mathcal{F}_\infty$ be the union of all $\mathcal{F}_n$. Of course $\mathcal{F}_\infty$ is infinite and for any path π that contains $\forall \bar{x}\, \mathcal{A}$ positively we also have $\mathcal{A}\left[\frac{\bar{x}}{\bar{x}_j}\right]$ for infinitely many different new variables $\bar{x}_j$. Let σ be a substitution such that for any branch and formula the σ-image of all the new variable tuples is the set of all r-tuples of ground term (r = length of $\bar{x}$). By our assumption no $\sigma(\mathcal{F}_n)$ is closed. By a standard argument, often called König's Lemma, we get an infinite open path π in $\sigma(\mathcal{F}_\infty)$. We use π to built a Herbrand model $\mathcal{D}$: A ground atomic formula A is true in $\mathcal{D}$ iff it occurs positively on π.

$$\text{If } A \text{ occurs positively on } \pi \text{ then } \mathcal{D} \vdash A$$

[10] We will see later that this does not require to switch from graphs to trees for an implementation. The key observation to avoid this is that substitutions can be considered relative to paths

and

If A occurs negatively on π then $\mathcal{D} \vdash \neg A$

For ground atoms the first claim is true by definition and the second follows from the fact that π is open.

Now assume that $\forall \bar{x}\, \mathcal{A}$ that occurs positively on π where $\mathcal{A}$ does not contain quantifiers. For any r-tuple $\bar{t}$ of ground terms we have $\sigma(\bar{x}_j) = \bar{t}$ for some j. A selective extension along the path π has added $\mathcal{A}\left[\frac{\bar{x}}{\bar{x}_j}\right]$ and π contains one path π_j of $\sigma(\mathcal{A}\left[\frac{\bar{x}}{\bar{x}_j}\right])$ as a subset. By induction hypothesis $\mathcal{D} \vdash \pi_j$ and therefore $\mathcal{D} \vdash \mathcal{A}\left[\frac{\bar{x}}{\bar{t}}\right]$. Since this is true for every r-tuple of ground terms and $\mathcal{D}$ is a Herbrand structure we have get $\mathcal{D} \vdash \forall \bar{x}\, \mathcal{A}$. By construction of $f2Sh_{\boxed{\vee}}(A)$ a **1**-path never contains a universal formula negatively. Thus the second claim above is vacuously satisfied.

In implementing the selective extension calculus we must consider all possible choices for extensions to guarantee completeness. One way to do this is to construct a sequence of sets of Shannon expressions, each set containing all possible choices:

Definition 31 Let F be an arbitrary first-order formula and $\mathcal{F} = f2Sh_{\boxed{\vee}}(F)$, then we recursively define a sequence $\mathcal{F}_0, \mathcal{F}_1, \ldots$ of subsets of $2^{\mathrm{SH}_{\boxed{\vee}}}$:

$$\mathcal{F}_0 := \{\hat{\mathcal{F}}\}$$

$$\mathcal{F}_{i+1} := \left\{ \mathcal{G}\left[\frac{\bot(\pi)}{\mathcal{G}\left[\frac{\bar{x}}{\bar{x}'}\right]}\right] \;\middle|\; \begin{array}{l} \mathcal{G} \in \mathcal{F}_i \text{ and } \pi \text{ is a path in } \mathcal{G} \text{ such that:} \\ (1)\ \pi \text{ starts at the root of } \mathcal{G}, \\ (2)\ \langle \forall \bar{x}\, \mathcal{G} \rangle \mathcal{BC} \text{ is an element of } \pi, \text{ and} \\ (3)\ \bot(\pi) = \mathbf{1}. \end{array} \right\}$$

Note, that all elements of $\mathcal{F}_0 \cup \mathcal{F}_1 \cup \ldots$ are logically equivalent.

Proposition 32 (Complete search for selective extensions) If F is unsatisfiable, there will be an $i \in I\!N$, such that an element of $\mathcal{F}_i$ is closed, i.e. $\mathcal{F}_i$ is inconsistent.

Problem 33 is used to give an example for a proof with selective extensions:

Problem 33

Axm 33.1: $p(a)$

Axm 33.2: $\neg p(f(f(a))) \wedge \neg p(g(g(a)))$

Axm 33.3: $(\forall x\, p(x) \rightarrow p(f(x))) \vee (\forall y\, p(y) \rightarrow p(g(y)))$

The left graph in Figure 6 on page 20 shows the initial Shannon graph. When we search for a proof and reach a **1**-leaf with a consistent path during the proof search, we can select one of the quantified subgraphs that we crossed positively with this path, and descend into the subgraph again with new variable bindings. This means that we can selectively choose the universally quantified subformula we need. The right graph in Figure 6 depicts this; we will shortly discuss how the graph closes to explain the situation: at the positive edge of the first quantified subgraph a **1**-leaf is reached. We need to extend and choose the only positive appearance of a quantified subgraph in the path we constructed: "$\forall Y$" . This graph is inserted for the **1**-leaf we reached[11]. We close the path at the negative edge of $p(Y_1)$ with $[Y_1/a]$ and extend again at the **1**-leaf of $p(g(Y_1))$. This extension closes with $[Y_2/g(a)]$.

[11] Note, that the explanation is made in terms of the search space and *not* in terms of representation.

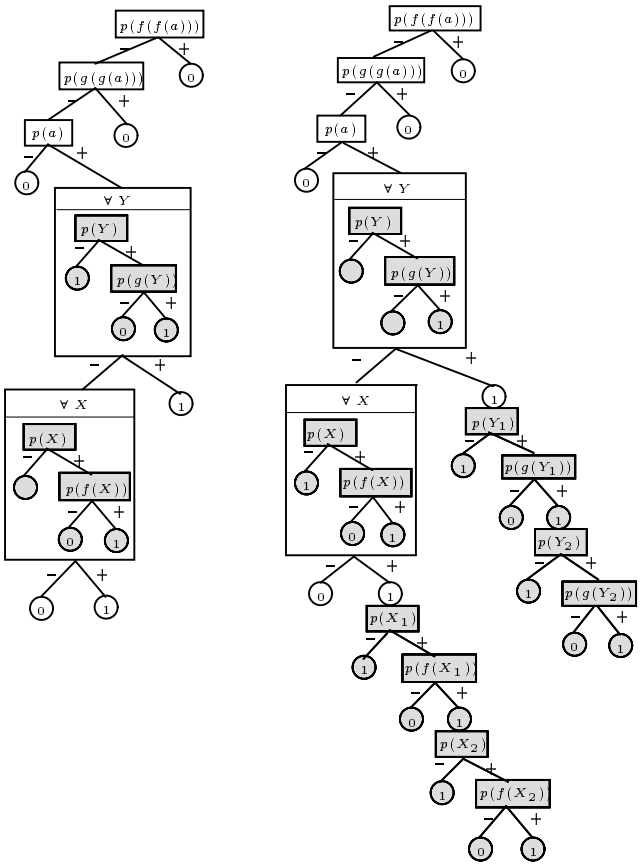


Figure 6: Using quantified subgraphs to model extensions

It remains to close the subgraph rooted at the negative edge of $p(g(Y))$, which is done analogously to the former. Note, that we must not use the "$\forall Y$"-subgraph for extending, because it was traversed through its negative edge.

6 Improvements

Whilst deduction with Shannon graphs works already rather efficient, there are still a number of ways for improvement. This applies to propositional logic, as well as to first-order logic. We will now consider some optimizations; those are not of a technical nature, but deal with with principal peculiarities of the representation of formulæ as Shannon Graphs.

6.1 CUT-nodes

The philosophy behind Shannon graphs is to represent models and counter models of a formula together: models as **1**-paths and counter models as **0**-paths. These counter models are then automatically considered in the proof search. To show that a disjunction $A \vee B$ is inconsistent, we must look at the **1**-paths in the Shannon graph for the disjunction. As this graph is built by inserting $\mathcal{B}$ for $\mathcal{A}$, all the **1**-paths going through $\mathcal{B}$ will be "prefixed" by the **0**-paths of $\mathcal{A}$. This is superfluous: it would suffice to consider all **1**-paths in $\mathcal{A}$ and all **1**-paths in $\mathcal{B}$ without combining them into longer paths.

In terms of the proof search this means: when showing that B is inconsistent, we assume $\neg A$, because either A has already been ruled out, or will be ruled out, later.

It may be observed in general that lemmata have advantages and disadvantages: they possibly allow shorter proofs. On the other hand, this does not mean that shorter proofs are also easier to find: lemmata might increase the search space. When considering first-order logic an additional problem arises: closing one path can prevent others from closing since the unification of complementary atoms can bind free variables. Since lemmata potentially increase the number of closing substitutions for paths, this is problematic if those additional substitutions are useless for the proof. Note, that this affects just *proof search*, and not the length of the shortest proof.

It can therefore be useful to have an alternative at hand that suppresses this lemma-generation[12]. To achieve this we introduce special nodes, $sh(!, \mathcal{A}, \mathcal{B})$, during the construction of a Shannon graph. These can simply be regarded as $sh(L, \mathcal{A}, \mathcal{B})$, where L is a new atom. From a semantical point of view, $sh(L, \mathcal{A}, \mathcal{B})$ is closed if and only if both $\mathcal{A}$ and $\mathcal{B}$ are closed; this will be the case if $\mathcal{A} \vee \mathcal{B}$ is not satisfiable. Adding the new literal can be seen as replacing a formula $A \vee B$ by $(A \wedge L) \vee (B \wedge \neg L)$. This correspondence is the reason why we call them CUT-nodes. Since the variable L is immaterial we suppress mentioning it and simply use ! as a label, see Problem 33; Figure 7 for an example. Note, that our use of CUT is somewhat dual to D'Agostino's usage [13]; he wants to add lemma-generation in a tableau-based calculus, we want to prevent it in our calculus.) Also note, that the use of CUT-nodes preserves satisfiability, but not logical equivalence.

Since both CUT-nodes and the standard nodes are just special cases of a general definition they may both be combined in one graph.

So, *f2Sh* can decide for each disjunction which representation to use. Here's a definition of *f2Sh* that uses CUT-nodes exclusively:

Definition 34 (Computing Initial Graphs with CUTs) The function $f2Sh_{\boxed{!}}$ is identical to *f2Sh* (Definition 8, page 8), besides that the rule for disjunction is replaced by

$$sh(!, f2Sh_{\boxed{!}}(A), f2Sh_{\boxed{!}}(B)) \quad \textit{iff } F = A \vee B$$

Analogously, a variant of $f2Sh_{\boxed{\vee}}$ with CUT-nodes can be defined.

Proposition 16 remains true for Shannon graphs with CUT-nodes, if we modify the notion of paths sets accordingly:

[12] We have already seen that a quantified subgraph that appears negatively on a path is never extended, although this would be sound and correspond to the principle of using lemmata. However, this would hardly be reasonable, because the additional Skolem-terms that appeared in lemmata do usually not contribute to a proof.

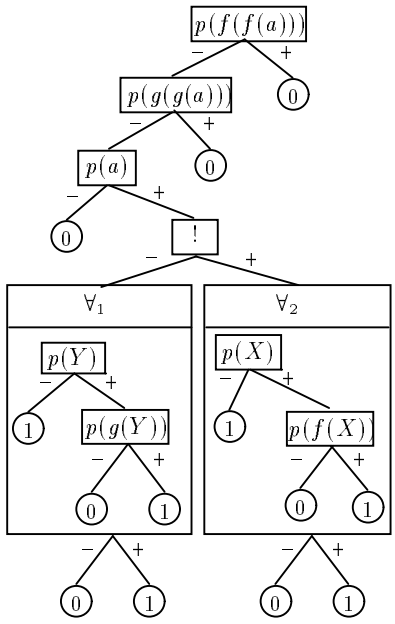


Figure 7: Introducing CUT to shorten the search space

Definition 35 (Paths in CUT-graphs)

$$\Pi^{\mathbf{1}}_{sh(!,A,B)} \stackrel{\text{def}}{=} \Pi^{\mathbf{1}}_{A} \cup \Pi^{\mathbf{1}}_{B} \quad \text{and} \quad \Pi^{\mathbf{0}}_{sh(!,A,B)} \stackrel{\text{def}}{=} \Pi^{\mathbf{0}}_{A} \cup \Pi^{\mathbf{0}}_{B}$$

7 Shannon Graphs and Semantic Tableaux

We have often drawn comparisons between Shannon graphs and semantic tableaux in the previous sections, but no formal relationship has been established, yet. Both calculi construct interpretations during the proof search that are potential models. Besides this common principle, there is actually a rather close relation in the models they investigate: it will be shown that there is a bijection from the paths in Shannon graphs to branches in propositional tableaux. It should be stressed that the results presented in this section apply to fully expanded tableaux. A comparison at the level of proof search is much more intricate and we do attempt it here.

We will first show the relation between CUT-Shannon graphs and standard tableaux, which is very close since CUTs have been introduced to mimic the treatment of disjunctions in tableaux. The second and more important result is that the bijection between paths in standard Shannon graphs and paths in semantic tableaux with lemma generation. The results are presented for propositional logic, but can be carried forward to the first order case.

7.1 Formalizing Tableaux

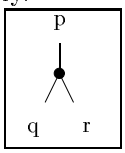
One principal problem when comparing tableaux and Shannon graphs is that tableaux are dynamic structures that are developed step by step during the search for a proof, whilst Shannon graphs are static objects. We could describe the Shannon graph calculus similarly to tableaux, which could conveniently be done. The idea is to start with a term $sh(F, 0, 1)$ for an inconsistent formula F, and then apply rules that will eventually rewrite this term into a Shannon graph with atomic nodes. In the propositional case, the graph will be closed if F is inconsistent. This would be closer to the classical understanding of a *calculus*. When having implementation in mind, however, efficiency beats classical aesthetics and our current framework seems more appropriate for being implemented. So, we will treat tableaux in an analogous way as at Shannon graphs and consider fully expanded tableaux only.

The search for a proof in a fully expanded tableaux is carried out by trying to close each branch[13] First, we define the notion of a fully expanded tableau.

Definition 36 (Set of Fully Expanded Tableaux) The set $\mathcal{TAB}$ of fully expanded tableaux is defined to be the smallest set such that

1. $\mathbf{1} \in \mathcal{TAB}$
2. if $\mathcal{T} \in \mathcal{TAB}$ and $A \in \mathcal{L}_{At}$, then $A \wedge \mathcal{T}$ and $\neg A \wedge \mathcal{T} \in \mathcal{TAB}$.
3. if $\mathcal{T}_1, \mathcal{T}_2 \in \mathcal{TAB}$, then $\mathcal{T}_1 \vee \mathcal{T}_2 \in \mathcal{TAB}$.

This notation is a bit exotic, but convenient. The intuition behind it is that the formulæ on a branch denote a conjunction, and that branching means disjunction. This means: tableaux are a disjunctive normal form. As a simple example, consider the formula "$p \wedge (q \vee r)$"; assume we start a tableau with this formula and expand it completely.

We get 

which can be written as $p \wedge ((q \wedge \mathbf{1}) \vee (r \wedge \mathbf{1}))$, an element of $\mathcal{TAB}$.

The atom "$\mathbf{1}$" is superfluous, but handy. Including it can be regarded as marking the end of a branch. Note, that it is possible to use a graph to represent a fully expanded tableau: as for Shannon graphs, we can build a directed, acyclic graph that exploits structure-sharing in a tableau[14].

On the left hand side of Figure 8 such a graph representing a fully expanded tableau is shown: each element of a branch is drawn in a frame and is consecutively numbered; the branching points of the tableaux are also included in this numbering. "$\Downarrow n \Downarrow$" stands for an edge to the node labeled with $\langle n \rangle$. This tableau corresponds to the following formula from Pelletier's problem set [20]:

[13] Note, that this results in a tableau proof procedure that never closes a branch by using compound formulæ for a contradiction, but only literals. From a theoretical point of view this might prevent finding a short proof; in practice, however, it very rarely happens that branches can indeed be closed with compound formulæ.

[14] Such a representation is actually more efficient than the standard approaches to implementing tableaux with the stepwise development of a tree[21].

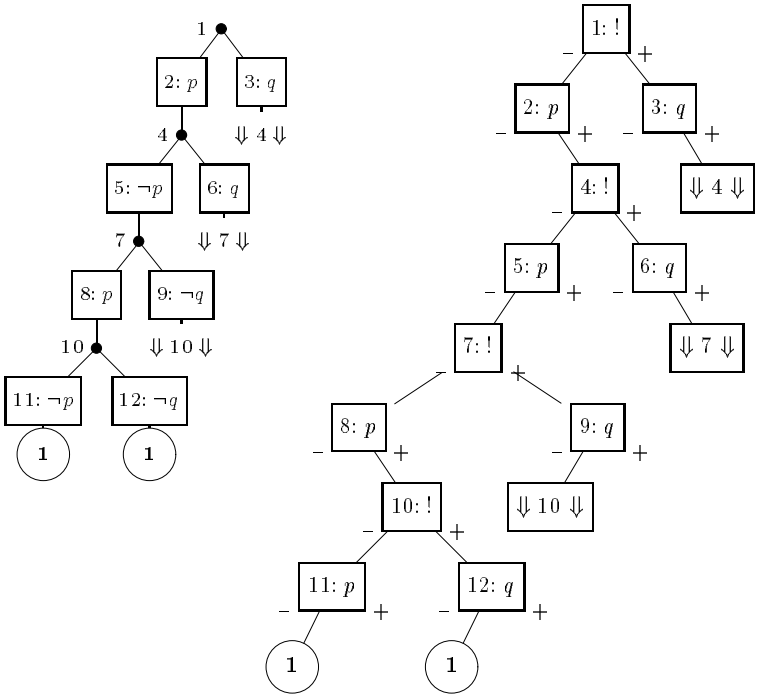


Figure 8: Tableau and Shannon Graph for Problem 37

Problem 37 Axm_Pel9: $\neg[((p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)) \rightarrow \neg(\neg p \vee \neg q)]$

7.2 CUT-Shannon graphs and Semantic Tableau

Consider again Figure 8: the Shannon graph $f2Sh_{\square}(Axm\_Pel9)$ is on the right hand side (all its edges to **0**-leaves have been omitted to reduce the size). A careful comparison with the tableau already shows that those two graphs are very closely related. Stating a formal relationship is indeed not difficult. Consider:

Proposition 38 (Structure of CUT-Shannon graphs) Let $\mathcal{G}$ be the result of applying $f2Sh_{\square}$ to an arbitrary formula. Then one of the two successors of each non-CUT node in $\mathcal{G}$ is always "**0**".

(The proof is straightforward: each non-CUT node is introduced by building a graph for a literal; $f2Sh_{\square}$ never replaces **0** nodes during the graph construction (cf. Definition 34 of $f2Sh_{\square}$ on page 21)

We can define a bijection between Shannon graphs and fully expanded tableaux:

Proposition 39 (Bijection Between CUT-Shannon Graphs and Tableaux) Let $\mathrm{SH}_{\square} = \{\mathcal{G} \mid \mathcal{G} = f2Sh_{\square}(F) \text{ for some } F \in \mathcal{L}\}$; there is a bijection "$\approx$" from $\mathrm{SH}_{\square}$ to $\mathcal{TAB}$, visualized in the following table:

Assume $A \in \mathcal{L}_{At}$.

$$\begin{array}{rcl} \mathcal{TAB} & \approx & \mathrm{SH}_! \\ \hline \mathbf{1} & \approx & \mathbf{1} \\ A \wedge \mathcal{T} & \approx & sh(A, \mathbf{0}, \mathcal{G}) \text{ where } \mathcal{T} \approx \mathcal{G} \\ (\neg A) \wedge \mathcal{T} & \approx & sh(A, \mathcal{G}, \mathbf{0}) \text{ where } \mathcal{T} \approx \mathcal{G} \\ \mathcal{T}_1 \vee \mathcal{T}_2 & \approx & sh(!, \mathcal{A}, \mathcal{B}), \text{ where } \mathcal{T}_1 \approx \mathcal{A}, \mathcal{T}_2 \approx \mathcal{B} \end{array}$$

This means that tableaux and CUT-Shannon graphs are just different notations for the same deduction principle. The above bijection can easily be carried forward to the first-order level: quantified subgraphs are regarded as a fully expanded tableau for γ-formulæ. Note that this difference in notation allied to a *graphical* representation of tableau (with structure sharing) and not to standard tableaux.

7.3 Standard Shannon Graphs and Semantic Tableau

The next question suggests itself: what about Shannon graphs without CUT nodes? The answer was already indicated when CUT-nodes were introduced in Section 6.1: they avoided the addition of lemmata into the models investigated.

We will use the same notation to denote branches in a tableau as we did to present paths in Shannon graphs, in particular we only record the appearing literals , not taking into account composed formulas and use

$$v\phi \stackrel{\text{def}}{=} \begin{cases} \phi & \text{if } v = + \\ \neg\phi & \text{if } v = - \end{cases}$$

Let $\mathrm{br}(\mathcal{T})$ denote the set of all branches of $\mathcal{T}$.

Proposition 40 Let $F \in \mathcal{L}, \mathcal{F} = f2Sh(F)$, and assume we use the following rule as the only β-rule for expanding a tableau:

$$\frac{A \vee B}{A \mid (\neg A) \wedge B}$$

Then, there is a fully expanded tableau, $\mathcal{T}_F$, such that

$$\mathrm{br}(\mathcal{T}_F) = \Pi^{\mathbf{1}}_{f2Sh(F)} \quad \text{and} \quad \mathrm{br}(\mathcal{T}_{\neg F}) = \Pi^{\mathbf{0}}_{f2Sh(F)}$$

Proof (Proposition 40 (outline)) The proof can be carried out by induction over the structure of F. The proposition is clearly valid for literals. Because of the β-rule with lemma generation a path in $\mathcal{T}_{A \vee B}$ either is a path in $\mathcal{T}_A$ or a path in $\mathcal{T}_{\neg A \wedge B}$. The path in the latter tableau are obtain by first following a path through $\mathcal{T}_{\neg A}$ and then continue along a path in $\mathcal{T}_B$. Thus

$$\mathrm{br}(\mathcal{T}_{A \vee B}) = \mathrm{br}(\mathcal{T}_A) \cup (\mathrm{br}(\mathcal{T}_{\neg A}) \odot \mathrm{br}(\mathcal{T}_B))$$

On the other hand $\Pi^{\mathbf{1}}_{f2Sh(A \vee B)}$ consists of $\Pi^{\mathbf{1}}_{f2Sh(A)}$ and since the leave node $\mathbf{0}$ of $f2Sh(A)$ is replaced by $f2Sh(B)$ we also have to consider $\Pi^{\mathbf{0}}_{f2Sh(A)} \odot \Pi^{\mathbf{1}}_{f2Sh(B)}$, i.e.

$$\Pi^{\mathbf{1}}_{f2Sh(A \vee B)} = \Pi^{\mathbf{1}}_{f2Sh(A)} \cup (\Pi^{\mathbf{0}}_{f2Sh(A)} \odot \Pi^{\mathbf{1}}_{f2Sh(B)})$$

. This proves the induction step for the formula $A \vee B$. The case of $A \wedge B$ is proved by the two equations:

$$\mathrm{br}(\mathcal{T}_{A \wedge B}) = \mathrm{br}(\mathcal{T}_A) \odot \mathrm{br}(\mathcal{T}_B) \text{ and } \Pi^{\mathbf{1}}_{f2Sh(A \wedge B)} = \Pi^{\mathbf{1}}_{f2Sh(A)} \odot \Pi^{\mathbf{1}}_{f2Sh(B)}$$

. It remains to consider the negated cases.

$$\begin{aligned} \mathrm{br}(\mathcal{T}_{\neg(A \wedge B)}) &= \mathrm{br}(\mathcal{T}_{\neg A \vee \neg B}) \\ &= \mathrm{br}(\mathcal{T}_{\neg A}) \cup (\mathrm{br}(\mathcal{T}_A) \odot \mathrm{br}(\mathcal{T}_{\neg B})) \\ &= \Pi^{\mathbf{0}}_{f2Sh(A)} \cup (\Pi^{\mathbf{1}}_{f2Sh(A)} \odot \Pi^{\mathbf{0}}_{f2Sh(B)}) \\ &= \Pi^{\mathbf{0}}_{f2Sh(\neg A \vee \neg B)} = \Pi^{\mathbf{0}}_{f2Sh(A \wedge B)} \end{aligned}$$

The remaining case for $\neg(A \vee B)$ is proved analogously.

The above Proposition does not directly relate the proof search in Shannon graphs and semantic tableaux with lemmata, but relates Shannon graphs to fully expanded tableaux. It could be paraphrased as "Shannon graphs can be understood as a representation of a fully expanded tableau (with lemmata)". This is a crucial point and the conclusion that non-CUT Shannon graphs relate to tableaux with lemmata in the same way as CUT Shannon graphs relate to standard tableaux might be tempting, but is, in a sense, misleading:

Recall that we are able to derive a Shannon graph in linear time and with linear space w.r.t. to the input formula. The structure of tableaux does not offer an equally efficient way to derive a representation of fully expanded tableau *with* lemmata, that is as compact as a Shannon graph. The reason is that Shannon graphs and tableaux represent different information: a tableau represents the models for a formula, whereas a Shannon graph represents models *and* counter-models, as paths to **1**-leaves and paths to **0**-leaves. In tableaux, only the paths to **1**-leaves are present, i.e. the interpretations satisfying a formula. Thus, extra effort is required for expanding and representing formulæ and their corresponding lemmata.

Another "conclusion", that Shannon graphs are in general superior to tableaux for this reason, is also not justified – at least in the first-order case: lemmata easily complicate things. But as there is a way to suppress lemmata in graphs, the overall assessment is clearly in favor of Shannon graphs. In the rest of this paper implementation techniques are described that shall support this rating.

8 Compilation Techniques for First-order Graphs

In this section we will show how the propositions 23 and 32 may be used as the basis for an implementation of a theorem prover. The methods set forth by these propositions do not place any limitations on the way to implement them. The approach we will follow in this section works by translating a Shannon graph of a formula into a program and then run this program in order to search for a proof. Such compilation-based approaches are well-known and have proven to be very efficient (see e.g. [25]). A major difference between these methods and our approach is that we do not rely on clausal form, but describe a compilation principle for general formulæ. The method

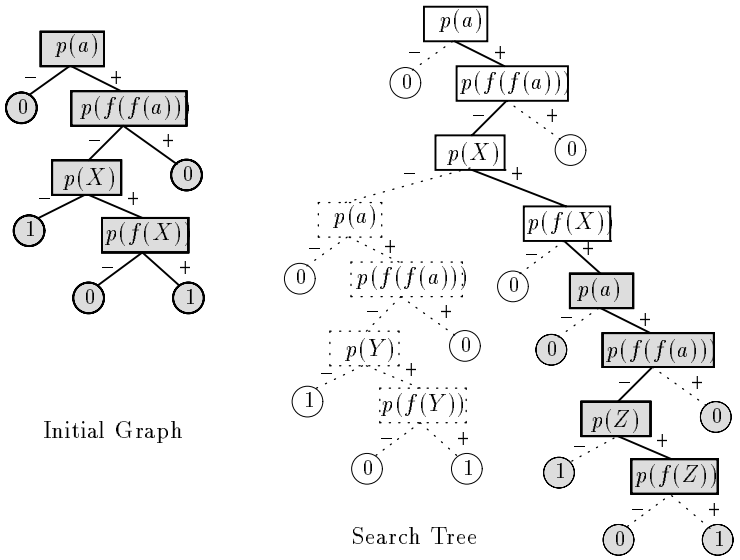


Figure 9: Proving Inconsistency of Problem 41

presented here therefore extends previously-known compilation-based approaches to non-clausal formulæ; furthermore, our compilation principle can easily be extended to other target languages, since we do not use Prolog as a Horn-clause language, but in a procedural way.

8.1 Maximal Extensions

The basic idea to show unsatisfiability of a formula of full first-order logic is the following: build an initial graph $\mathcal{F}_0(\bar{X}_0)$ for some Skolemized first-order formula $F(\bar{X})$, and try to find a closing substitution σ for $\mathcal{F}_0(\bar{X}_0)$. If this fails, extend $\mathcal{F}_0(\bar{X}_0)$ by substituting all of its **1**-leaves with a copy of $\mathcal{F}_0(\bar{X}_0)$ with fresh variables. The process then continues until eventually some graph is closed, if and only if $F(\bar{X})$ is inconsistent. Figure 9 on page 27 shows an example we have already considered:

Problem 41
Axm 41.1: $p(a)$
Axm 41.2: $\neg p(f(f(a)))$
Axm 41.3: $\forall x\ (p(x) \rightarrow p(f(x)))$

The left graph is the initial graph constructed by *f2Sh*; there is no closing substitution, so a disjoint extension is tried: we replace each **1**-leaf by a copy of the graphs itself after having renamed the free variables. The result is shown on the right hand side. Now, $\{a/X, f(a)/Z\}$ is a closing substitution. This cannot directly be represented as a graph, since we would have different free variables in the same subgraph, depending on the path we use to enter it. However, there is an elegant solution to this if we think in terms of paths: consider the above process again in terms of **1**-paths:

we can subsequently (say, from left to right) consider the **1**-paths in an initial graph and try to close them as early as possible. For this, we may apply a substitution that instantiates the path to become contradictory. If we "arrive" at a **1**-leaf and the current path cannot be closed, we continue this path at the root again and regard all free variables in the graph as being renamed. This corresponds to a disjoint extension at the current path.

One major problem we must face are dead loops: in the above example, these can arise if we would always instantiate the variables to a. This would always close one branch, but never all of them. So, we must either use a fair selection scheme for instantiations, or initiate backtracking at some point and enumerate alternatives. As we chose Prolog to demonstrate our implementation method, we will use the latter alternative and implement a bounded depth-first search strategy.

The key to achieve an efficient implementation is the efficient construction of paths; this can be left to the Prolog engine: we translate each non-terminal node in a Shannon graph into a Prolog clause and these clauses "call" each other if their corresponding nodes in the graph are connected by en edge. Thus, the calling hierarchy at run time resembles the tree represented by the Shannon graph. In order to close branches, we must know the previously "visited" nodes. This is achieved by passing this information through all calls of Prolog clauses. Furthermore, we need to represent bindings of variables in nodes, because Prolog clauses are by definition variable-disjoint. As the number of variables in a graph is known before, we can use a Prolog term with a fixed arity to represent this. Making variable bindings explicit is also the key to modeling extension properly: when calling the clause for the root from a **1**-leaf, we simply drop all variable bindings which corresponds to renaming all variables in the graph.

This outlines our approach to compiling Shannon graphs into Prolog programs. As an example, consider the clause generated for node labeled with $p(X)$ in the first graph in Figure 9. For compiling, we need a unique identifier for nodes; assume this node is called $n3$, its positive successor is $n4$, and the root is called $n1$.

```
node(n3,Path,bind(X)):-                                    (1)
      (closed(-p(X),Path)                                  (2)
      ; node(n1,[-p(X)|Path],_)),                          (3)
      (closed(+p(X),Path)                                  (4)
      ; node(n4,[+p(X)|Path],Binding)).                    (5)
```

The generated clauses will succeed if the Shannon graph they implement is closed, i.e. all paths can be made inconsistent. The head of the clause (1) has the name of the node, the path it was called with, and the variable binding as parameters. The letter is a unary term, because there is only one variable in the graph.
Line (2) tries to close the path via the negative edge of $n3$. The predicate succeeds if the list that is passed as the second argument contains an element that is unifiable and contradictory the first argument[15] and complementary literals.
Alternatively to closing the branch, we can call the clause for the negative successor. This is a **1**-leaf, so an extension must be implemented (line (3): it works by adding $-p(X)$ to the path and calling the root ($n1$). Renaming the variable is implemented

[15] Note that sound unification is required and that `closed` must enumerate all alternatives during backtracking.

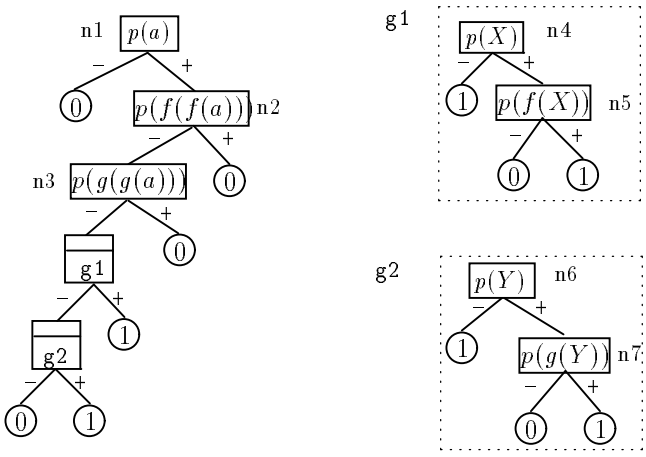


Figure 10: Shannon Graph for Problem 42

by dropping the variable binding[16].
Lines (4) and (5) implement taking the positive edge of $n3$: either the path can be closed, or the clause for $n4$ succeeds.

This surprisingly simple compilation scheme achieves a complete implementation for Shannon graphs with maximal extensions — provided that we limit the depth of the search and then start to explore alternatives for closing branches. This can, for instance, be done by limiting (better: iteratively deepening) the number of allowed extensions or the length of paths.

Compiling selective extensions works quite similar; some additional effort is required, since extensions are bound to quantified subgraphs, and not to the whole graph.

8.2 Selective Extensions

Although the theoretical treatment of selective extensions in Chapter 5.1 was more complicated than the treatment of maximal extensions, their compilation is only sightly more complicated. We describe the principle with the following example:

Problem 42

Axm 42.1: $p(a)$
Axm 42.2: $\neg p(f(f(a))) \wedge \neg p(g(g(a)))$
Axm 42.3: $(\forall x\ p(x) \rightarrow p(f(x))) \vee (\forall (y)\ p(y) \rightarrow p(g(y)))$

The initial graph consists of the interior nodes for $p(a)$, $\neg p(f(f(a)))$, $\neg p(g(g(a)))$, and the two conjunctively combined quantified subgraphs for Axm 42.3 (see Figure 10, the quantified graphs are drawn separately).

In terms of paths, we may extend selectively, if we crossed a node containing a quantified subgraph through the node's positive edge. Extension will only take

[16]Note that previously applied substitution will remain in the path.

```
memberunify(X,[Y|_]) :- unify(X,Y).
memberunify(X,[_|L]) :- !,memberunify(X, L).

closed(+Literal,Path) :- memberunify(-Literal, Path).
closed(-Literal,Path) :- memberunify(+Literal, Path).

extend(PATH,bind(X,Y)):- member(gamma(Node),PATH),node(Node,PATH,bind(X,Y)).

node(n1,PATH,bind(X,Y)):-(closed(+p(a),PATH)
                          ; node(n2,[+p(a)|PATH],bind(X,Y))).
node(n2,PATH,bind(X,Y)):-(closed(-p(f(f(a))),PATH)
                          ; node(n3,[-p(f(f(a)))|PATH],bind(X,Y))).
node(n3,PATH,bind(X,Y)):-(closed(-p(g(g(a))),PATH)
                          ; node(g1,[-p(g(g(a)))|PATH],bind(X,Y))).
node(g1,PATH,bind(X,Y)):- node(g2,PATH,bind(X,Y)),extend([gamma(n4)|PATH],bind(X,Y)).
node(g2,PATH,bind(X,Y)):-  extend([gamma(n6)|PATH],bind(X,Y)).
node(n4,PATH,bind(_,Y)):- (closed(-p(X),PATH) ; extend([-p(X)|PATH],bind(X,Y))),
                          (closed(+p(X),PATH) ; node(n5,[+p(X)|PATH],bind(X,Y))).
node(n5,PATH,bind(X,Y)):- (closed(+p(f(X)),PATH) ; extend([+p(f(X))|PATH],bind(X,Y))).
node(n6,PATH,bind(X,_)):- (closed(-p(Y),PATH) ; extend([-p(Y)|PATH],bind(X,Y))),
                          (closed(+p(Y),PATH) ; node(n7,[+p(Y)|PATH],bind(X,Y))).
node(n7,PATH,bind(X,Y)):- (closed(+p(g(Y)),PATH) ; extend([+p(g(Y))|PATH],bind(X,Y))).
```

Figure 11: Sample Prolog code for Problem 42

place at **1**-path, so paths we construct will just contain information about the nested nodes we visited. If a **1**-leaf is reached, we select an applicable universally quantified subgraph and call its root clause after dropping the corresponding slots in the variable binding.

Figure 11 shows the complete Prolog program for the above problem.

The first four lines define a predicate `closed`, as described above. `extend` handles extensions: when a quantified node is visited and left through the positive edge, a Prolog term `gamma(`$\langle n \rangle$`)` will be added to the path, where n is the name of the root of the quantified graph. `extend` selects one of those terms from the path and calls the corresponding Prolog clause (dropping variable bindings is done elsewhere).

The principle of the clauses for nodes is the same as with maximal extensions; recall again that a clause succeeds if the graph under it can be closed. The clauses for $n1$, $n2$, and $n3$ must "implement" only one edge, since only non-**0** edges must be considered.

A quantified node is implemented separately from its contents: g1 is an example: we do not try to close the branch (it is not necessary for completeness) and call the clause for g2. No information is added, since the quantified node was left through its negative edge[17]. The positive successor is **1**, so `extend` is called after adding a `gamma` term for selecting future extensions.

Due to lack of space, we cannot discuss the rest of the generated program in detail, but the other clauses work analogously to previously discussed cases. The reader might wish to verify that $n4$ and $n6$ actually implement a variable renaming for the quantified variables by dropping the binding at the according positions.

[17] We could add something like "`-gamma(...)`" to the path and also try to close paths using such components of paths, but it is not necessary for completeness.

As a concluding remark to this section, we would like to emphasize that we did not intend to present efficient, but readable Prolog code. There are numerous ways to optimize the generated code, but a discussion of this is beyond the scope of this paper[18] Last but not least, it should be stressed that it is also possible to compile to other target languages than Prolog: in a functional language, for instance, we can map nodes into functions. Machine-oriented languages require more effort, but can also speed up the proof search considerably: an experimental implementation of a propositional prover compiling to Intel Assembler showed a speedup in the order of 40 compared with Prolog.

8.3 Comparison with other theorem provers

We will briefly compares the performance of a prototypical Prolog-implementation of our method [18] with the ${}_3T^AP$ theorem prover [16] and PTTP [25]. Although one can doubt that it is actually possible to compare two different theorem provers in a reasonable way, we include the figures to give the reader a "feeling" on how Shannon graphs compare to existing approaches.

The idea of this section is to support two claims: the first one is to practically verify the theoretical result that Shannon graphs offer advantages over standard semantic tableaux, and, secondly, that our compilation principle for non-clausal formulæ results in a more efficient proof search than previously known approaches offer. We will therefore compare the implementation of the Shannon graph system with two provers: One is the ${}_3T^AP$-system, a tableaux-based prover which was originally intended to implement multi-valued logic. We used its two-valued variant, which implements a standard tableau-calculus with free variables. The second prover we consider is PTTP, a well-known approach to implementing first-order logic by generating Prolog programs. As a test set we are using Pelletier's problems $18 - 46$ [20], which are all first-order. All runtimes have been measured with Quintus Prolog Release 3.0 on a Sun Sparc Station II with 36 MBytes of physical main memory and a swap space of 150 MByte.

8.3.1 The Shannon Graph prover vs. ${}_3T^AP$

The ${}_3T^AP$ theorem prover is a system based on free-variable tableaux, and also implemented in Prolog.

The left part of Figure 12 on page 32 shows clearly the advantage of the principle of compiling to Prolog: the proof search with compiled Shannon graphs is much faster than in ${}_3T^AP$ (both figures measure the time for the actual proof search and do not include preprocessing/compilation). The number of inferences is the number of tableau rule applications in the case of ${}_3T^AP$, and the number of visited Shannon graph nodes. Both figures cannot be directly compared as they measure different things, but they do reflect the effort for carrying out the proof. A better measure is the number of paths or branches in Shannon graphs or tableaux, respectively, that have

[18] Some points to start with would be that `extend` should include heuristics for selecting γ-graphs, `closed` should not search a simple list, but a sophisticated datastructure. Other optimizations like "unfolding" parts of the Prolog clauses, statically analyzing the need for unification at nodes beforehand, and most other known techniques of automated deduction are also applicable.

Pel.	*Proof[msec.]*		*Inferences*		*Paths*		*Backtr.*		*Inferences*	*Proof[msec.]*
	${}_3T^AP$	*Sh*	${}_3T^AP$	*Sh*	${}_3T^AP$	*Sh*	${}_3T^AP$	*Sh*	*PTTP*	*PTTP*
18	34	0	3	3	1	1	0	0	1	0
19	83	0	6	25	2	7	0	3	2	0
20	133	0	13	6	3	3	0	0	3	0
21	83	0	13	15	9	7	0	0	18	0
22	67	0	7	10	4	4	0	0	21	0
23	100	0	9	8	4	4	0	0	253	5
24	300	0	51	20	22	14	0	0	137	17
25	100	0	16	14	7	7	0	0	54	33
26	250	0	39	44	12	23	0	0	>6 470 423	$\infty^\dagger$
27	317	17	53	59	23	36	0	19	17	0
28	133	17	18	35	5	10	0	0	>77 479	$\infty^\dagger$
29	267	17	46	69	18	32	0	8	>1 341 700	$\infty^\dagger$
30	67	0	10	11	3	5	0	0	6	0
31	116	17	11	9	4	4	0	0	4	17
32	117	0	12	13	6	7	0	0	8	0
33	167	0	25	42	9	27	0	3	4	0
34	1 583	150	219	259	76	77	0	0	667 296‡	$\infty^\ddagger$
35	50	0	5	2	1	1	0	0	1	0
36	100	0	13	14	3	7	0	0	3	0
37	166	16	21	13	5	5	0	0	8	0
38	9 733	500	726	1 115	358	474	102	422	>742 336	$\infty^\dagger$
39	67	0	6	4	2	2	0	0	4	17
40	117	0	13	10	5	4	1	0	26	17
41	50	0	5	7	5	3	1	0	6	0
42	67	0	15	10	5	5	0	0	1 135	2
43	1 700	0	158	36	97	25	21	0	>1 358 307	$\infty^\dagger$
44	117	0	15	16	5	7	1	0	15	0
45	700	0	111	27	31	13	9	0	1 783	35
46	183	17	26	26	10	17	0	0	45 039	935

†Proof search aborted after a couple of minutes

‡Run time error of Quintus-Prolog: "internal error: segmentation fault".

Figure 12: Performance of ${}_3T^AP$, Compiled Shannon Graphs, and PTTP

been closed for finding the proof. In most cases the Shannon graph proof procedure needs to consider more paths than ${}_3T^AP$ opens branches. This is chiefly due to the fact that the proof search in ${}_3T^AP$ is more focussed due to the use of static links for deciding which γ-formula to expand. The Shannon graph prover blindly selects a γ-node for expansion; this easily creates branches that are useless for the proof and thus generates unnecessary overhead. This results in more backtracking during the proof search than ${}_3T^AP$ requires. However, the overall time required for finding a proof is much shorter for compiled Shannon graphs, since the compilation results in a higher inference rate: pel34 and pel38, for example, suggest an inference rate of approximately 40–50 branches or 75 rule applications per second for ${}_3T^AP$, and of 500-1000 paths or 2000 visited nodes per second for compiled Shannon graphs. It is likely, that compiling to low-level machine code would again speed things up — at least by a factor of 10.

8.3.2 The Shannon Graph prover vs. PTTP

PTTP is a prover that compiles formulæ in conjunctive normal form into Prolog and uses these programs for the proof search. So, it's working principle is similar to our's. However, it requires CNF, which often causes problems in that the number of axioms explodes; this is especially the case for formulæ containing equivalences, like problems 34 and 38 of Pelletier's problem set. Those cannot be solved by PTTP as the number of required axioms explodes. Most other examples are more or less given close to CNF by Pelletier, so this should not reason for PTTP's failure to find proofs. However, the scope of quantifier can often be held narrower with Shannon graphs, which also increases performance.

Version "pttp-in-prolog-2e" has been used for deriving the results shown in the right part of Figure 12 on page 32; clearly, the Shannon Graph prover outperforms PTTP. The prover was stopped if it failed to deliver a proof after about 10 minutes CPU time. Proof time is the time used for the proof search.

9 Conclusion & Outlook

We described a new approach to automated deduction which is based on Shannon graphs, a variant of non-ordered BDDs. These BDDs have been extended to first-order logic by incorporating means for representing quantifiers. A calculus based on Shannon graphs was set up and its soundness and completeness was proven.

It was shown that the proof search in this calculus behaves similar to semantic tableaux with lemmata. In this sense the method resembles well-known theorem proving techniques and most research results can directly be carried forward. Besides issues relevant to Automated Deduction, the presented approach also contributes to research in BDD: Shannon graphs are a superset of ordered BDDs and the method for lifting Shannon graphs to first-order logic can be carried forward to them.

Besides these more theoretical issues, we also described a method for efficiently implementing a deduction system based on Shannon graphs: It works by translating an arbitrary first-order formula to a Shannon graph, which is then compiled into a program. We demonstrated the method with the target language Prolog, but any other general-purpose programming language can be used. Prolog is just a very convenient language for this, as its abstraction level is very suitable for implementing first-order deduction: datastructures for variables, terms, substitutions, etc. are predefined and need not to be implemented.

The generated program resembles the search through a graph, which that can be understood as a case analysis over the truth values of the atoms in the input formulæ. The search process tries to show that no model for the formulæ can exist by exploiting properties of the graph that are logically equivalent to this. The method differs from other approaches to deduction by Horn-clause generation (e.g. [25]), in that the generated clauses have no logical relation to the formula that is to be proven (i.e., are not a logically equivalent variant of the formula), but that they are *procedurally* equivalent to the search for a model. Furthermore, out approach works for general formulæ and not only for clauses.

The described method has been implemented and showed good performance. This was documented by comparing the approach with other deduction systems. Further-

more, a successful application of the proposed calculus to hardware verification showed its practical relevance [24].

References

[1] F. L. Bauer and M. Wirsing. *Elementare Aussagenlogik*. Springer, 1991.

[2] B. Beckert, S. Gerberding, R. Hähnle, and W. Kernig. The many-valued tableau-based theorem prover $_3T^AP$. In *CADE-11, Albany/NY*, pages 758–760. Springer LNAI, 1992.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt. The *even more* liberalized δ–rule in free variable semantic tableaux. In *Proceedings of the Kurt Gödel Conference, Brno, Czech Republic*. Springer LNCS, 1993.

[4] D. A. Bell. Decision trees, tables, and lattices. In B. G. Batchelor, editor, *Pattern Recognition: Ideas in Practise*. Plenum Press, New York, 1978.

[5] J.-P. Billon. A new approach of theorem proving for non clausal first order logic with equality based on generalized shannon's decomposition principle. Technical Report ORDA/DMA/91037, Bull Corporate Research Center, Paris, France, December1991.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27^{th} ACM/IEEE Design Automation Conference*, pages 40 – 45. IEEE Press, 1990.

[7] R. Y. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C–35:677 – 691, 1986.

[8] R. Y. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical report, Carnegie Mellon University. School of Computer Science, 1992.

[9] J.R. Burch. Using BDDs to verify multipiers. In *Proc. 28th Design Automation Conference (DAC 91)*, pages 408–412, 1991.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual Symposium on Logic in Computer Science*, 1990.

[11] H. Y. Chang, E. Manning, and G. Metze. *Fault Diagnosis of Digital Systems*, volume 62. Wiley, New York, 1970.

[12] A. Church. *Introduction to Mathematical Logic*, volume 1. Princeton University Press, Princeton, New Jersey, 1956. Sixth printing 1970 .

[13] M. d'Agostino. Are tableaux an improvement on truth–tables? Cut–free proofs and bivalence. *Journal of Logic, Language and Information*, 1(3), 1992.

[14] A. Ehrenfeucht and E. Orłowska. Mechanical proof procedure for propositional calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, XV(1):25–30, 1967.

[15] M. C. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer, 1990.

[16] R. Hähnle, B. Beckert, S. Gerberding, and W. Kernig. The Many–Valued Tableau–Based Theorem Prover ${}_3T^4P$. IWBS Report 227, Wissenschaftliches Zentrum Heidelberg, IWBS, IBM Deutschland, July1992.

[17] M. Z. Hanani. An optimal evaluation of boolean expressions in an online query system. *Communications of the ACM*, 20(5):344–347, 1977.

[18] B. Ludäscher. Ein Deduktionssystem für Prädikatenlogik erster Stufe basierend auf Shannon Graphen. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, October1992.

[19] E. Orłowska. Automatic theorem proving in a certain class of formulae of predicate calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, XVII(3):117 – 119, 1969.

[20] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191 – 216, 1986.

[21] J. Posegga. Compiling proof search in semantic tableaux. In *Proc. 7^{th} International Symposium on Methodologies for Intelligent Systems*, pages 67–77, Trondheim, Norway, June1993. Springer LNAI.

[22] J. Posegga. *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe.* Infix Verlag, Sankt Augustin, 1993.

[23] J. Posegga and B. Ludäscher. Towards first-order deduction based on shannon graphs. In *Proc. German Workshop on Artificial Intelligence*, Bonn, Germany, 1992. Springer LNAI.

[24] Klaus Schneider, Ramayya Kumar, and Thomas Kropf. Hardware verification using first order bdds. In *Proc. Computer Hardware Description Languages (CHDL)*, 1993.

[25] M. E. Stickel. A Prolog Technology Theorem Prover. *Journal of Automated Reasoning*, 4(4):353–380, 1988.