An Algorithm for the Class of Pure Implicational Formulas

John Franco*†, Judy Goldsmith‡§, John Schlipf*†, Ewald Speckenmeyer¶, R. P. Swaminathan‖

Abstract

Heusch introduced the notion of pure implicational formulas. He showed that the falsifiability problem for pure implicational formulas with k negations is solvable in time $\mathbf{O}(n^k)$. Such falsifiability results are easily transformed to satisfiability results on CNF formulas.

We show that the falsifiability problem for pure implicational formulas is solvable in time $\mathbf{O}(k^k n^2)$, which is polynomial for a fixed k. Thus this problem is fixed-parameter tractable.

1 Introduction

Since Cook's [1] proof of NP-completeness for 3-SAT, numerous studies of complexity of testing satisfiability and/or falsifiability for particular classes of propositional formulas have been done. Two important classes are CNF and DNF formulas. For CNF formulas, Cook showed that testing satisfiability is NP-complete, whereas testing falsifiability can be done in linear time. For DNF the reverse is true: testing satisfiability (resp. falsifiability) for DNF formulas can easily be reduced to testing falsifiability (resp. satisfiability) for CNF formulas. Other classes of formulas, such as Horn, extended Horn, q-Horn, and SLUR, have been shown to be solved in polynomial time (see [9] for algorithms and credits). Recursively defined hierarchies of incrementally harder classes have also been defined and studied [3, 5, 7]. Formulas on level k of these hierarchies typically can be solved in $\mathbf{O}(n^k)$ time.

Downey and Fellows [4] considered hierarchies of problems. Levels of such a hierarchy are distinguished by a parameter k which in some sense measures the density of an instance; any algorithm to solve instances of length n for fixed k takes time bounded by some $c_k(1+n^{e_k})$. Now as k increases, does the value of e_k increase, or does it hold steady while only the value of c_k increases? An algorithm of the latter type they call *fixed parameter tractable*. Fixed parameter tractability has obvious implications for algorithm efficiency on instances where n is much larger than k; for more information about fixed parameter tractability we refer the reader to Downey and Fellows' papers. In this paper we show that the falsifiability problem, described below, for pure implicational

*ECECS, University of Cincinnati, Cincinnati, Ohio 45221-0030.

†Supported in part by ONR grant N00014-94-1-0382.

‡Computer Science Department, University of Kentucky, Lexington, Kentucky.

§Supported in part by NSF grant CCR-9315354.

¶Institut für Informatik, Universität zu Köln, Germany.

‖RUTCOR and DIMACS Center, Rutgers University, Piscataway, New Jersey 08903-5062.

formulas of length n containing at most k occurrences of $\mathbf{f}$, is fixed parameter tractable. In the cases of [3, 5, 7], it is not known whether there are polynomial-time fixed parameter algorithms. As far as we know, the results of this paper are the first fixed parameter tractability results for hierarchies of satisfiability.

Following the work of Peter Heusch [6], we consider formulas in a different "normal form": formulas built up from the *proposition letters* (or *propositional variables*) with $\rightarrow$ (implication) and a propositional *constant* $\mathbf{f}$ for false (so $\neg\mathbf{f}$ is always true) — plus parentheses, of course. Call a formula built up with only these connectives *pure implicational.* We consider algorithms for determining satisfiability and falsifiability of formulas containing at most two occurrences of each proposition letter and at most k occurrences of $\mathbf{f}$.

Actually, Heusch considered only falsifiability, and he did not allow occurrences of $\mathbf{f}$. Rather, he considered formulas whose only propositional connective is $\rightarrow$, which contain at most k occurrences of some distinguished proposition letter z, and which contain at most two occurrences of every other proposition letter. However, he showed that determining falsifiability was hardest in the case of formulas of the form

$$\alpha_1 \rightarrow (\alpha_2 \rightarrow (\ldots \rightarrow (\alpha_h \rightarrow z))).$$

In this case z is set to false in every falsifying truth assignment. Thus it is equally difficult to determine falsifiability of the formula

$$\alpha_1 \rightarrow (\alpha_2 \rightarrow (\ldots \rightarrow (\alpha_h \rightarrow \mathbf{f}))).$$

In this way Heusch essentially reduced his context to the one we use.

Since it slightly simplifies our arguments, we shall focus on satisfiability. It is equally difficult to test for falsifiability and to test for satisfiability, since a formula ϕ is satisfiable if and only if $(\phi \rightarrow \mathbf{f})$ is falsifiable, and ϕ is falsifiable if and only if $(\phi \rightarrow \mathbf{f})$ is satisfiable. Without loss of generality, we limit the study to satisfiability of a single pure implicational formula ϕ. For suppose $\alpha_1, \ldots, \alpha_m$ are pure implicational formulas. Then $\{\alpha_1, \ldots, \alpha_m\}$ is satisfiable if and only if

$$\alpha_1 \rightarrow (\ldots \rightarrow (\alpha_m \rightarrow \mathbf{f}))$$

is falsifiable.

Theorem 1.1 (Heusch [6]) *Testing falsifiability for pure implicational formulas containing at most two occurrences of each proposition letter is NP-complete.* □

Corollary 1.2 *Testing satisfiability for pure implicational formulas containing at most two occurrences of each proposition letter is NP-complete.* □

But by fixing the value of the extra parameter, the number of occurrences of **f** (or the number of occurrences of z in his case), Heusch was able to get a type of tractability result for special cases. From this point on, we use n to represent the length of a pure implicational formula.

Theorem 1.3 (Heusch [6]) *Testing falsifiability for pure implicational formulas, of length n, containing at most two occurrences of each proposition letter and at most k occurrences of* **f** *can be performed in time* $\mathbf{O}(n^k)$. □

The following corollary is trivial; we assume that more careful analysis of Heusch's proof would easily change the $k+1$ to a k below.

Corollary 1.4 *Testing satisfiability for pure implicational formulas, of length n, containing at most two occurrences of each proposition letter and at most k occurrences of* **f** *can be performed in time* $\mathbf{O}(n^{k+1})$. □

The organization of the paper is as follows. In Section 2 we show how to transform pure implicational formulas to a forest of trees containing no more nodes than the number of implication connectives. The trees are such that there is a collection of "consistent paths," from root to leaf, one for each tree, if and only if the original formula is satisfiable. The search for such a collection is simplified by the fact that inconsistencies arise only between a leaf and its "shadow." In Section 3 we devise a $\mathbf{O}(k^k n^2)$ algorithm for searching a forest for inconsistent paths. The collection of algorithms used in all stages is presented as a whole at the end of the section.

2 Exploring possibilities

Our algorithm consists of three stages. The first consists of parsing and translating the given formula into a different propositional form. The second constructs at most k trees of literals, where k is the number of occurrences of **f** in the given formula. The third stage consists of solving a combinatorial problem on the trees. The solution to this tree problem determines the satisfiability of the given formula.

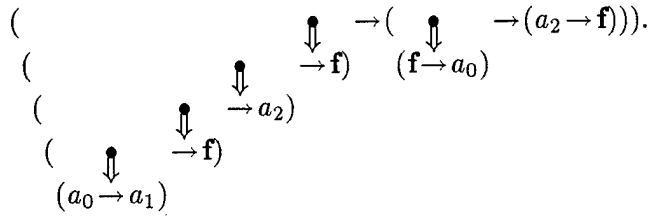
2.1 Initial parsing/translation

The first stage of the algorithm involves replacing a pure implicational formula ϕ with a set $\Theta = \{\theta_1, \ldots, \theta_m\}$ $(m \leq n)$ of formulas in a larger set of proposition letters, where ϕ is satisfiable if and only if Θ is. I.e., ϕ and Θ are *equisatisfiable*.

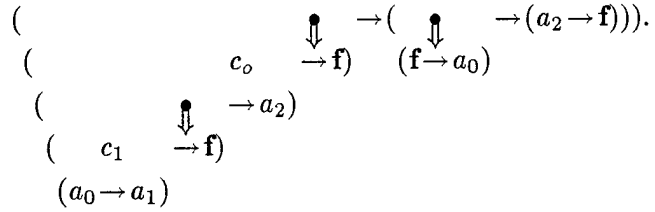
1. We start with a pure implicational formula, for example,

$$(((((a_0 \to a_1) \to \mathbf{f}) \to a_2) \to \mathbf{f}) \to ((\mathbf{f} \to a_0) \to (a_2 \to \mathbf{f}))).$$

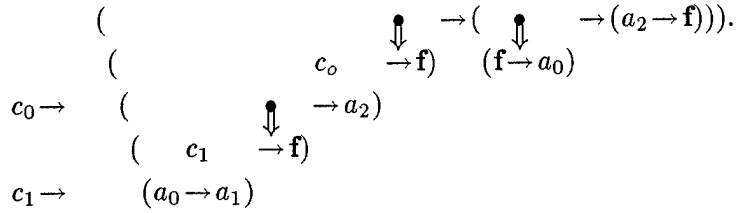
We parse, counting levels of nesting to the *left hand side* of the $\to$'s; below we show nesting on the left by dropping down to lower lines:



2. Next, at even numbered levels (> 0) of nesting on the left, we first replace the embedded formulas above with new proposition letters.



In order to make sure that our new set of formulas and the original formula ϕ are equisatisfiable, we need also to relate c_i's to the formulas they replace. It is natural to add in an axiom $c_1 \leftrightarrow (a_0 \to a_1)$. That would, however, force c_1 to occur more than twice in our final translation. Fortunately, we need assert only the $\to$ direction of the equivalence to guarantee equisatisfiability. Thus we get our translation, in this case a set Θ of three formulas:



We sketch the proof of equisatisfiability: If the original formula ϕ has a model ν, we expand ν by setting $\nu(c_i)$ to the value under ν of the corresponding subformula of ϕ, which yields a model of Θ. Conversely, suppose ν is a model of Θ; we show that ν is also a model of ϕ. As noted in our discussion of our intuition above, if for each c_i, $\nu(c_i)$ is the value of the formula it replaces in the expansion above — i.e., if $\nu(c_0) = \nu((c_1 \to \mathbf{f}) \to a_2)$ and $\nu(c_1) = \nu(a_0 \to a_1)$, then ν is a model of ϕ. So suppose not. Pick all the highest level formulas for which this

property fails. For example, suppose here that $\nu(c_0) \neq \nu((c_1 \rightarrow \mathbf{f}) \rightarrow a_2)$. Since Θ contains the formula $c_0 \rightarrow ((c_1 \rightarrow \mathbf{f}) \rightarrow a_2)$, it must be that $\nu(c_0) = \mathit{false}$ and $\nu((c_1 \rightarrow \mathbf{f}) \rightarrow a_2) = \mathit{true}$. From the truth table for propositions nested an even number of times on the left of an $\rightarrow$, we can see that if the top formula is satisfied when $\nu(c_0)$ is false, it is satisfied with *any* formula substituted in for c_0.

Note that now not only the original proposition letters (the a_i's), but also the new ones (the c_i's), occur at most twice in Θ.

3. We now have three formulas in our set Θ, each with at most one level of nesting on the left-hand side of the $\rightarrow$'s.

$$\begin{aligned}
\theta_0 &= (c_0 \rightarrow \mathbf{f}) \rightarrow ((\mathbf{f} \rightarrow a_0) \rightarrow (a_2 \rightarrow \mathbf{f})) \\
\theta_1 &= c_0 \rightarrow ((c_1 \rightarrow \mathbf{f}) \rightarrow a_2) \\
\theta_2 &= c_1 \rightarrow (a_0 \rightarrow a_1)
\end{aligned}$$

We finish with a simple substitution of tautologically equivalent formulas. We replace the *nested-on-the-left* occurrences of $\rightarrow$ with their DNF equivalents, yielding

$$\begin{aligned}
\theta_0 &= (\neg c_0 \vee \mathbf{f}) \rightarrow ((\neg \mathbf{f} \vee a_0) \rightarrow (a_2 \rightarrow \mathbf{f})) \\
\theta_1 &= c_0 \rightarrow ((\neg c_1 \vee \mathbf{f}) \rightarrow a_2) \\
\theta_2 &= c_1 \rightarrow (a_0 \rightarrow a_1);
\end{aligned}$$

we replace each remaining $\beta_1 \rightarrow (\beta_2 \rightarrow (\ldots \beta_j))$ with the equivalent $\neg\beta_j \rightarrow (\neg\beta_1 \vee \neg\beta_2 \vee \ldots \neg\beta_{j-1})$, yielding

$$\begin{aligned}
\theta_0 &= \neg\mathbf{f} \rightarrow \neg(\neg c_0 \vee \mathbf{f}) \vee \neg(\neg \mathbf{f} \vee a_0) \vee \neg a_2 \\
\theta_1 &= \neg a_2 \rightarrow (\neg c_0 \vee \neg(\neg c_1 \vee \mathbf{f})) \\
\theta_2 &= \neg a_1 \rightarrow \neg c_1 \vee \neg a_0;
\end{aligned}$$

and we simplify using deMorgan's laws and double negation, yielding finally

$$\begin{aligned}
\theta_0 &= \neg\mathbf{f} \rightarrow (c_0 \wedge \neg\mathbf{f}) \vee (\mathbf{f} \wedge \neg a_0) \vee \neg a_2 \\
\theta_1 &= \neg a_2 \rightarrow (\neg c_0 \vee (c_1 \wedge \neg\mathbf{f})) \\
\theta_2 &= \neg a_1 \rightarrow \neg c_1 \vee \neg a_0.
\end{aligned}$$

Comment 2.1 *Note that any atom that occurs only positively can be set to* **true**, *without affecting the satisfiability of these formulas. Therefore, before we continue, we perform this simplification. (This simplifies later exposition, and can be done in time* $\mathbf{O}(n^2)$.*)*

Note the important properties of Θ:

- Θ and ϕ are equisatisfiable.

- Each proposition letter — a_i or c_i — occurs at most twice in Θ.
- The constant $\mathbf{f}$ occurs k times in Θ.
- Each θ_i is an implication, whose hypothesis consists of a negative literal (either a $\neg a_i$ or a $\neg c_i$ or a $\neg\mathbf{f}$), and whose conclusion consists of a disjunction, where each disjunct is the conjunct of one negative literal and zero or more (zero or one in our example) positive literals.

All occurrences of $\mathbf{f}$ on the right-hand sides of the θ_i's could be simplified out, although our algorithm assumes that we do not do so. The complexity of the algorithm will be given in terms of the number $k' \leq k$ of occurrences of $\neg\mathbf{f}$ on the *left hand sides* of the θ_i's.

2.2 Building trees of choices

We now have a collection $\Theta = \{\theta_1, \ldots, \theta_m\}$ of formulas as described above. As we noted earlier, our strategy is now dual to the strategy of finding whether a set of Horn clauses is satisfiable. There one starts with a default truth assignment of false to every variable and uses the implications to identify variables which must be true. Here we start with a default truth assignment of true to every variable and use the implications to identify choices of variables to set to false to satisfy the formulas. If there are no occurrences of $\neg\mathbf{f}$ in the hypotheses of the θ_i's (i.e., $k' = 0$) then $\{\theta_1, \ldots, \theta_m\}$ is satisfied by the all-true assignment.

We want to search for a minimal change to the all-true truth assignment to satisfy $\{\theta_1, \ldots, \theta_m\}$. To start out, we must satisfy each of the k' formulas with head $\neg\mathbf{f}$. In order to do that we must satisfy one of the disjuncts of each body. Suppose we satisfy a disjunct $\neg v$, and suppose some other θ_i is $\neg v \rightarrow \neg w \vee \neg x$; then we will also have to set either w or x to false. To keep track of our choices, for each such formula $\theta_i = \neg\mathbf{f} \rightarrow \ldots$, we build a tree T_i with root labeled $\neg\mathbf{f}$. Associated with each interior node and leaf of these trees will be a set of labels, corresponding to the literals in the relevant disjunction of some formula, as described in the algorithm below. Later we shall find an interpretation by choosing a branch through the tree; the literals labeling a node will be literals we shall be forced to satisfy when the algorithm chooses that branch.

For notational convenience below, we assume that we have moved the θ_i's with hypothesis $\neg\mathbf{f}$ to $\theta_1, \ldots, \theta_{k'}$.

Algorithm 2.1

For $i = 1, \ldots, k'$
  Build tree T_i *as follows:*
    Create a root r_i *labeled* $\neg\mathbf{f}$.
    For each disjunct $l_1 \wedge l_2 \wedge \cdots \wedge l_j$ *of the consequent of* θ_i,

Create a child of r_i with labels $l_1, l_2, \ldots, l_j$.
Mark that child unexplored
While there are unexplored nodes:
Pick any unexplored node w and mark it explored.
If the node has a negative literal $\neg a$ as a label
and does not have **f** *as a label*
If there is a θ_j with hypothesis $\neg a$
(there can be only one such θ_j by the 2-occurrence restriction)
For each disjunct $l_1 \wedge l_2 \wedge \cdots \wedge l_j$ of the consequent of θ_j
Create a child of w with labels $l_1, l_2, \ldots, l_j$
Mark that child unexplored.
Remove all nodes all of whose descendents are labeled **f**.
If a node does not have any children, mark it as a leaf.

As we build the trees, we also build the following:

- *An array* Occurrence, *indexed by the proposition letters, whose index-a position contains pointers to the (at most 2) occurrences of a in the T_i's.*
- *For each node w, an array, indexed by the proposition letters, whose index-a position stores whether a and/or $\neg a$ label nodes on the path to w from its tree's root.*

Comment 2.2 *We immediately use the auxiliary data structures for a simplification: if any negative literal $\neg a$ occurs negatively in two leaves, those leaves can, without loss of generality, be removed, and their parent nodes marked as leaves.*

It is easy to see how to construct the Θ's and the T_i's in time n^2. (That is almost certainly not the optimal time. However, other phases of our satisfiability algorithm are also $\mathbf{O}(n^2)$, so this suffices for our final analysis.)

We shall use the trees above to help us search for a satisfying truth assignment for $\{\theta_1, \ldots, \theta_m\}$. We shall build up this assignment in stages, combining *partial* truth assignments. To simplify the exposition, we introduce a notion of a *set of literals* satisfying such a formula. We shall consider a *truth assignment* to be a set $\mathcal{A}$ of literals such that, for each proposition letter a, exactly one of a and $\neg a$ is in $\mathcal{A}$.

Definition 2.1 *Let $\mathcal{IMPL}$ be the set of formulas whose only propositional connectives are* **f**, $\neg$, $\rightarrow$, $\vee$, *and* $\vee$. *For a set of literals, $\mathcal{A}$, we say that $\mathcal{A}$* satisfies *δ, written $\mathcal{A} \models \delta$, for $\delta \in \mathcal{IMPL}$, if the following hold.*

- $\mathcal{A} \models \neg\mathbf{f}$; $\mathcal{A} \not\models \mathbf{f}$.
- *For l either a or $\neg a$, a a proposition letter, $\mathcal{A} \models l$ if $l \in \mathcal{A}$.*
- $\mathcal{A} \models \alpha \vee \beta$ *if* $\mathcal{A} \models \alpha$ *or* $\mathcal{A} \models \beta$.
- $\mathcal{A} \models \alpha \wedge \beta$ *if* $\mathcal{A} \models \alpha$ *and* $\mathcal{A} \models \beta$.
- $\mathcal{A} \models \alpha \rightarrow \beta$ *if* $\mathcal{A} \not\models \alpha$ *or* $\mathcal{A} \models \beta$.

Definition 2.2 *Let T_i be one of the trees above and let P be any path on T_i (from the root to any leaf of T_i). Then $\mathcal{A}_P$ is the set of labels of nodes of P.*

So, to satisfy $\{\theta_1, \ldots, \theta_m\}$ with a *partial truth assignment* we have an easy solution: First pick one path P_i from each T_i ($1 \leq i \leq k'$), and start with the interpretation $\mathcal{A}_0 = \mathcal{A}_{P_1} \cup \cdots \cup \mathcal{A}_{P_{k'}}$. That is enough to satisfy chains of inferences starting with $\theta_1, \ldots, \theta_k$. Then set $\mathcal{A} = \mathcal{A}_0 \cup \{\text{proposition letters } v : v, \neg v \notin \mathcal{A}_0\}$. This (vacuously) satisfies all the θ_i's not addressed before, since each θ_i that is not used in some T_i has a negated literal as hypothesis. If θ_i has hypothesis $\neg a$, and θ_i was not used in the construction of any T_j, $\neg a$ does not occur as a label in any of the T_j's. Thus, $\neg a$ does not occur in $\mathcal{A}_0$, so either $a \in \mathcal{A}_0$ or $a \in \mathcal{A}$.

The only difficulty with the above construction is that $\mathcal{A}_0$ may well be inconsistent, i.e., it may contain both a and $\neg a$ for some proposition letter a. Thus we have:

Theorem 2.1 *Let $\theta_1, \ldots, \theta_m$ and $T_1, \ldots, T_{k'}$ be as above, and let $\mathcal{A}$ be a truth assignment. Then:*

1. *If $\mathcal{A} \models \{\theta_1, \ldots, \theta_m\}$ then there are paths P_i of T_i, $i = 1, \ldots, k'$ such that $\mathcal{A}_{P_i} \subseteq \mathcal{A}$.*
2. *If (1) there are paths P_i of T_i, $i = 1, \ldots, k'$ such that $\mathcal{A}_{P_i} \subseteq \mathcal{A}$, and (2) if, for each proposition letter a not appearing in any of the P_i's, $a \in \mathcal{A}$, then $\mathcal{A} \models \{\theta_1, \ldots, \theta_m\}$.*
3. *$\{\theta_1, \ldots, \theta_m\}$ is satisfiable if and only if there are paths P_i of T_i, $i = 1, \ldots, k'$ such that for no atom a are both a and $\neg a$ in $\mathcal{A}_{P_1} \cup \mathcal{A}_{P_2} \cup \cdots \mathcal{A}_{P_{k'}}$.* □

Remark 2.2 *The above theorem can be simplified further. Since each variable can appear at most twice in $\{\theta_1, \ldots, \theta_m\}$, and since a variable is used to label a node only when it appears as a consequent in one of the θ_i's, the only way we can have both $a, \neg a$ in $\mathcal{A}_{P_1} \cup \mathcal{A}_{P_2} \cup \cdots \mathcal{A}_{P_{k'}}$ is for a to label some node of some P_i and for $\neg a$ to label the* leaf *of some P_j.*

Note that, by the 2-occurrence property for atoms, and by the construction of the T_i's, a negative literal appears in these trees either uniquely in an interior node of some T_i, or in some leaf (or possibly two leaves).

If we pick two paths, they are inconsistent with each other iff there is an atom a such that a appears in one path, and $\neg a$ appears in the other. If this happens, then $\neg a$ is a label of some leaf w. Therefore, if we wish to avoid inconsistencies, once we pick a leaf, w, with a label $\neg a$, we must avoid all paths through the (possibly nonexistent) node y with label a. In other words, no paths can contain y, nor any of its descendants. We call the set of nodes in the subtree (cone) rooted at y the *shadow* of w ($Shadow(w)$).

If a leaf does not have a negative label, we say it has *an empty shadow.*

Remark 2.3 *Let $\{\theta_1, \ldots, \theta_m\}$ and $T_1, \ldots, T_{k'}$ be as above. Then $\{\theta_1, \ldots, \theta_m\}$ is satisfiable iff there is a set of leaves $w_1 \in T_1, \ldots, w_{k'} \in T'$ where no w_i is in any w_j's shadow.*

Remark 2.4 *Using the data structures specified in Algorithm 2.1, given any two leaves w_i, w_j, we can test in constant time whether $w_i \in Shadow(w_j)$.*

2.3 The reduction to graph theory

In the previous subsections we reduced, in at most quadratic time, satisfiability of the original pure implicational formula to the following combinatorial problem on trees, where n is the length of the original formula ϕ (including parentheses — this allows for our new constants c_{γ_i}) and k' is less than or equal to the number of occurrences of $\mathbf{f}$ in ϕ.

Remember the following facts:

- $T_1, \ldots, T_{k'}$ are labeled trees, with a total of at most n nodes.
- There is a partial function *Shadow* from leaves to sets of nodes. For each leaf w, if $Shadow(w) \neq \emptyset$, $Shadow(w)$ consists of just the descendents of a single node in just one of the trees.

Definition 2.3

- *A* shadow-independent set *in $\{T_1, \ldots, T_{k'}\}$ is a set of leaves $\{w_i \in T_i : 1 \leq i \leq k'\}$ such that no leaf w_i is in any w_j's shadow (including its own).*
- *The* shadow problem *for $\{T_1, \ldots, T_{k'}\}$ is, "Does $\{T_1, \ldots, T_{k'}\}$ have a shadow-independent set?"*

Proposition 2.5 *If S_1, S_2 are two shadows, then one of the following must hold: (1) S_1, S_2 are disjoint, (2) $S_1 \subseteq S_2$, or (3) $S_2 \subseteq S_1$.*

A simple upper bound on the difficulty of the shadow problem can be found by reducing it to the independent set problem. (Since this reduction gives only an upper bound on the complexity of the problem, it is omitted.) Thus we can apply to the shadow problem any general algorithm for finding whether a graph has a k-independent set. The fastest known (at least to us) algorithm is an $\mathbf{O}(n^{k(2+\epsilon)/3})$ algorithm by Nešetřil and Poljak [8], where $2+\epsilon$ is the best known exponent for fast matrix multiplication (see [2]). This is an improvement on Heusch's algorithm, but it is still not fixed parameter tractable.

Thus, short of showing that the k-independent set problem is fixed parameter tractable, our approach must amount, essentially, to finding special features of the resultant graph which allow for faster algorithms. We already have such a result in Proposition 2.5. In the remaining sections we show that the special features given in Proposition 2.5 are in fact strong enough: we give an $\mathbf{O}(n^2)$ algorithm (for each k) for this problem.

3 Solving the Shadow Problem

3.1 Easy Simplifications

Before we start, we identify, for each leaf w of each tree T_i, the tree T_j in which $Shadow(w)$ lies. If w has an empty shadow, we pick an arbitrary tree $T_j \neq T_i$ and declare that w has an empty shadow in tree T_j.

Given the set of trees $\{T_1, \ldots, T_{k'}\}$, we can perform the following simplifications in quadratic time. Clearly, none of the simplifications changes the answer to the shadow problem. Note that the simplification steps must be repeated until no further changes are made or unfalsifiability has been determined:

Algorithm 3.1 *Repeat the following simplifications until there are no more simplifications to be performed.*

1. *If any leaf shadows itself, delete the leaf — since no unshadowed set could contain such a leaf.*
2. *If any leaf's shadow is in its own tree, but does not include the leaf itself, delete the shadow (i.e., remove the label $\neg a$ from the leaf and the label a from the base of the shadow) — since we need to pick only one leaf from each tree anyway.*

   Redefine the leaf's shadow to be an empty shadow in some other tree.
3. *If any leaf's shadow is an entire tree T_i, remove the leaf — since no unshadowed set could contain such a leaf.*

4. *Remove any node which is not marked as a leaf, yet has no children. (This just prunes interior nodes whose children have all been removed.)*

5. *If any node has only one child, merge it with its child, labeling the merged node with all literals labeling either the original node or its child. If its child was a leaf, mark it as a leaf.*

6. *If any tree becomes empty, return "unsatisfiable".*

Remark 3.1 *Algorithm 3.1 can be performed in time* $\mathbf{O}(n^2)$.

3.2 A partition into shadow patterns

To find a shadow-independent set of leaves, we must pick one leaf from each tree. Think of a directed edge going from the leaf's tree to the shadow's tree (which, by the simplifications above, must be a different tree). Call the resultant graph on the k trees the *shadow pattern* of that set of leaves. Note that each node in a shadow pattern has out-degree one, so there are $(k-1)^k$ possible shadow patterns on the k trees. We shall partition our algorithm by looking separately at each of the $(k-1)^k$ possible shadow patterns, checking to find if there is an unshadowed set inducing that shadow pattern.

Say that a leaf l_i *matches a shadow pattern* P if l_i is in tree T_i, the edge in the shadow pattern from T_i goes to T_j, and l_i's shadow is in T_j. A set of leaves *matches a shadow pattern* if every leaf in the set matches the pattern.

Note that, to search for a shadow-independent set matching a shadow pattern P_h, we can clearly search each weakly connected component of P_h (i.e., each component of the undirected version of P_h) separately. The following algorithm is then clearly correct, since each choice of k leaves, one from each tree, matches one of the shadow patterns. We use it as the outside control of our solution algorithm.

Algorithm 3.2

For each shadow pattern P_h, $1 \leq h \leq (k-1)^k$,
  Make a copy T_i^h *of each of the original trees* T_i
  Delete from T_i^h *all leaves which do not match shadow pattern* P_h.
  Simplify T_i^h *as in Algorithm 3.1.*
  For each weakly connected component W *of* P_h
    search for a shadow-independent set on W. (†)
  If every weakly connected component has a shadow-independent set
    return "satisfiable."
Return "unsatisfiable".

The difficulty, of course, is in step (†). We shall show that we can implement that step in $\mathbf{O}(n_W^2)$, where n_W is the total number of nodes in the trees $T_i^h \in W$. It will follow that the above algorithm can be performed in time $\mathbf{O}(k^k n^2)$, since we have already noted that Algorithm 3.1 takes only quadratic time. (Clearly, it suffices to call Algorithm 3.1 on each component W separately.)

3.3 Solving for a fixed component of a fixed shadow pattern

All that is left is to show, as promised above, that we can test whether a weakly connected component $\{T_{w_1}^h, \ldots, T_{w_g}^h\}$ of $\{T_1^h, \ldots, T_{k'}^h\}$ has a shadow-independent set in time quadratic in the number of nodes in the component. The essential intuition used by the algorithm is the following:

1. In acyclic parts of the component, we can easily perform the test by brute force, working our way in from the leaves. We do this by pruning away nodes which *cannot* be in a shadow-independent set.
2. In the cyclic part of the graph, we show that, unless the pattern of shadows (after the pruning described above) has a specific, easily recognized structure described in Theorem 3.3, there is a shadow-independent set.

Since P_h is a directed graph on k nodes where each vertex has out-degree one, each of its weakly-connected components is a directed tree with edges directed toward the root plus one back edge. Or, another way of looking at it, each component consists of a single directed cycle c plus a set of mutually disjoint trees $\{t_1, \ldots, t_r\}$ (directed from leaves to root), where the root of each tree is also a member of the cycle, and otherwise the cycle and the trees are disjoint. We start with a separate pruning algorithm for the trees t_i:

Algorithm 3.3

Pruning Algorithm:

Input: *A component W of P_h along with the member trees T_i^h, consisting of:*

1. *A collection of mutually disjoint trees, $t_1, \ldots, t_f$, and*
2. *A single cycle c, where*
   a. *the roots of the t_g's are elements of c,*
   b. *no other nodes are shared by the t_gs and c.*

The nodes of c and the t_g's are all trees T_i^h created before.

Action:

From each tree, prune those nodes that cannot be in any shadow-independent set.

Identify the c and the t_g's.

For each tree t_g
  For each node T_i^h *of* t_g, *starting with the leaves of* t_g
    and working backwards to the root
    If T_i^h *has children in* t_g
      For each child T_j^h *of* T_i^h *in* t_g
        For each leaf w *of* T_i^h
          If w *is in the shadow of* every *leaf of* T_j^h
            delete w *from* T_i^h.
Simplify W *as in Algorithm 3.1.*
If the simplification algorithm returned "unsatisfiable"
  return that there is no shadow-independent set matching P_h.

Lemma 3.2

1. *If a leaf* l_i *of a tree* T_i^h *is deleted by Algorithm 3.3, that leaf cannot be included in any shadow-independent set from the input trees matching the given shadow pattern.*
2. *If the pruning algorithm returns that there is no shadow-independent set matching* P_h *(last line of the algorithm), then that result is correct.*
3. *Every shadow-independent set on the cycle* c, *after the pruning algorithm is finished, can be expanded to a shadow-independent set on all of the component* W.
4. *The total running time of Algorithm 3.3 is* $\mathbf{O}(|W|^2)$, *where by* $|W|$ *we mean the total number of nodes in all the member trees* T_i^h *in* W.

Proof:

1. Straightforward.
2. If the simplification algorithm reports unsatisfiability, it is because one of its trees has become empty. If any tree becomes empty, there are no leaves left to put into the shadow-independent set.
3. Suppose we have a shadow-independent set I on the pruned cycle c. Remember that some elements of c are roots of trees t_g and thus have been pruned. We must expand I to a shadow-independent set on all of W. Remember also that every tree has its root in c. Working out from the roots of the trees t_g to the leaves of the t_g's, we have by construction, that no matter what leaf l we picked for the independent set in the parent node T_i^h in some t_g, we can pick leaves l' in each of the children T_j^h (children in t_g) which do not shadow l. Pick any such l''s and continue recursively.

4. The cycle c and the trees t_g can be identified by a standard depth-first search. Since the search is just on W, the time is $\mathbf{O}(|W|)$. The remainder of the algorithm is basically a matter of checking through all pairs of leaves from adjacent trees, hence the quadratic bound. □

Definition 3.1 *By a leaf-descendent of a node x of a tree, we mean any descendent of x which happens to be a leaf, or x itself if that is a leaf.*

Theorem 3.3 *Let c be a simple cycle in P_h as above, containing trees $T_0^h, \ldots, T_{k''-1}^h$. (We simply renumber the trees to get them numbered as stated.) There is no shadow-independent set of leaves from $T_0^h, \ldots, T_{k''-1}^h$ matching P_h if and only if the following condition holds:*

Condition (‡):

After Algorithm 3.1 is called (again) on $T_0^h \cup \ldots \cup T_{k''-1}^h$, either some T_i^h becomes empty or

1. *the root of each of these T_i^h's has exactly two children — call them ${c_i}^{left}$ and ${c_i}^{right}$ (in some order);*
2. *if we choose which child of the root of each T_i^h to name ${c_i}^{left}$ and which ${c_i}^{right}$ correctly, every leaf-descendent of ${c_i}^{left}$ (resp., ${c_i}^{right}$) has the same shadow, which is*

   for $i < k'' - 1$: ${c_{(i+1)}}^{right}$ *(resp.* ${c_{(i+1)}}^{left}$*) and all of its descendents, or*
   for $i = k'' - 1$: ${c_0}^{left}$ *(resp.* ${c_0}^{right}$*) and all of its descendents.*

Proof:

$\Rightarrow$: We are given the cycle c consisting of $T_0^h, \ldots, T_{k''-1}^h$. By the simplifications (Algorithm 3.1) we performed, each T_i^h either (i) consists of just one node, which is not in the shadow of any other node in $T_0^h, \ldots, T_{k''-1}^h$, or (ii) has at least two children, and no node in c shadows more than one of these children.

We start by looking at a very gross approximation to the existence of a shadow-independent set. For each T_i^h consisting of more than a single node (and hence having at least two children of the root by our simplifications), let $M_i^0, \ldots, M_i^{c_i}$ be the maximal subtrees of T_i^h, i.e. the subtrees starting at the children of the root. For the single node T_i^h's, let $M_i^0 = T_i^h$. Now we build a directed *accessibility graph* $\mathcal{G}$ on these M_i^j's. One can think of $\mathcal{G}$ as the cycle c with most of its nodes — those T_i^j's which are nontrivial — split into pieces, one per subtree at the root of T_i^j.

The nodes of $\mathcal{G}$ are the M_i^j's. There is a directed edge in $\mathcal{G}$ from M_i^j to $M_{(i+1)\mathrm{mod}(k'')}^{j'}$ if there is a leaf of M_i^j whose shadow is *disjoint* from $M_{(i+1)\mathrm{mod}(k'')}^{j'}$. Since no leaf of M_i^j shadows

more than one of the $M^{j'}_{(i+1)\mathrm{mod}(k'')}$'s, there are edges from each M^j_i to at least all but one of the $M^{j'}_{(i+1)\mathrm{mod}(k'')}$'s, and if there is only one $M^{j'}_{(i+1)\mathrm{mod}(k'')}$, then there is an edge from M^j_i to $M^{j'}_{(i+1)\mathrm{mod}(k'')}$. Thus also there is at least one edge out of each M^j_i.

Now we claim that if $\mathcal{G}$ has a k''-cycle then $\{T^h_0, \ldots, T^h_{k''-1}\}$ has a shadow independent set. For suppose $M^{j_0}_0, M^{j_1}_1, \ldots, M^{j_{k''-1}}_{k''-1}$ is a k''-cycle. By the definition of $\mathcal{G}$, there is a leaf l_i in each $M^{j_i}_i$ whose shadow is disjoint from $M^{j_{(i+1)\mathrm{mod}(k'')}}_{(i+1)\mathrm{mod}(k'')}$. Pick any such sequence of leaves $l_0, l_1, \ldots, l_{k''-1}$. Clearly, $\{l_0, l_1, \ldots, l_{k''-1}\}$ is shadow-independent.

The basic strategy for constructing a shadow-independent set from $\mathcal{G}$ is this: out of each node M^j_i there is at least one edge in $\mathcal{G}$. So do a depth first search of $\mathcal{G}$ to find out what nodes of $\mathcal{G}$ are accessible from each M^j_0 in e edges, $1 \leq e \leq k''$. (This search is in the correctness proof; our algorithm does *not* need to perform the search since it does not need to construct the satisfying assignment: it needs only to determine whether one exists.) It follows from the above that, from each M^j_0, (1) some node $M^{j'}_0$ is accessible in k'' edges, and (2) all but at most one of the $M^{j'}_0$'s are accessible in k'' steps.

We first show that if $\mathcal{G}$ has no k''-cycles then the root of each T^h_i has exactly two children. First suppose the root of some T^h_i has only one child; without loss of generality, we may assume it to be T^h_0. Since we performed Algorithm 3.1 immediately before entering this phase, T^h_0 has only one node, and $M^0_0 = T^h_0$. Tracing forward k'' steps along any path through $\mathcal{G}$, we come back to some M^j_0 — and thus to M^0_0, since that's the only one there is. This gives a directed k'' cycle, and thus an unshadowed set.

Next, suppose that there is no shadow-independent set, and every T^j_i's root has at least two children, but that some T^j_i has more than 2 children. Without loss of generality, we may assume that $T_{k''-1}$'s root has at least 3 children. Consider the nodes in $\mathcal{G}$ accessible in $k''-1$ and in k'' edges from M^0_0 and M^1_0. At least two of $M^0_{k''-1}$, $M^1_{k''-1}$, and $M^2_{k''-1}$ are accessible from each of M^0_0 and M^1_0; thus one of them, $M^j_{k''-1}$, is accessible from both. Now there must be an edge from $M^j_{k''-1}$ to at least one of M^0_0 and M^1_0; this edge will complete a k'' cycle. That completes showing property (‡1).

Next we show part of property (‡2): assuming that there is no shadow-independent set, out of each M^j_i there is exactly one edge in $\mathcal{G}$. Without loss of generality, choose ${M_0}^{left} = M^0_0$ and ${M_0}^{right} = M^1_0$. We know there is a path of length k'' out of ${M_0}^{left}$; since we assumed there is no shadow-independent set in the T^h_i's, and hence k''-cycle in $\mathcal{G}$, every such path must lead to ${M_0}^{right}$. Similarly, there is a k''-length path out of ${M_0}^{right}$, and every such path must lead to ${M_0}^{left}$. Finally, if there were more than one k''-length path out of either — say out of ${M_0}^{left}$ — then one of those paths would have to share a node with the path out of ${M_0}^{right}$, which would imply the existence of a k''-cycle. (Thus we see that $\mathcal{G}$ is a simple $2k''$ cycle, doubled

up on c like a Möbius strip.) Without loss of generality, we may call the nodes of $\mathcal{G}$ accessible from ${M_0}^{left}$ (resp., ${M_0}^{right}$) in i edges, $1 \leq i \leq k'' - 1$, ${M_i}^{left}$ (resp., ${M_i}^{right}$), and we have that the only edge out of ${M_{k''-1}}^{left}$ (resp., ${M_{k''-1}}^{right}$) goes to ${M_0}^{right}$ (resp., ${M_0}^{left}$).

Now we prove the rest of property (‡2). Suppose that there is some i where there is a leaf of ${M_i}^{left}$ whose shadow is not all of ${M_{i+1}}^{right}$. (The case for some leaf of ${M_i}^{right}$ whose shadow is not all of ${M_{i+1}}^{left}$ is analogous.) Without loss of generality, we may assume that $i = 0$. We shall now construct a k''-cycle in $\mathcal{G}$, contradicting the assumption that there is no shadow-independent set in the set of T_i^h's.

Pick w_0 to be a leaf in ${M_0}^{left}$ whose shadow is not all of ${M_1}^{right}$, and pick w_1 to be any leaf of ${M_1}^{right}$ which is not in the shadow of w_0. For $i = 2, \ldots, k''$, pick w_i to be any leaf of ${M_i}^{right}$. By our choice of w_0, w_1, w_0 does not shadow w_1. By our analysis above, w_i does not shadow w_{i+1} for any $i < k'' - 1$. Finally, again by our analysis above, since $w_{k''-1} \in {M_i}^{right}$, the shadow of $w_{k''-1}$ is in ${M_0}^{right}$, so $w_{k''-1}$ does not shadow w_0. So we have constructed a k''-cycle, contradicting our assumption.

$\Leftarrow$: Obvious. □

Proposition 3.4 *Condition* (‡) *of Theorem 3.3 can be checked in time* $\mathbf{O}(|W|)$. *(*$|W|$ *is defined in Lemma 3.2.)*

Assembling all the pieces we have constructed we have a faster algorithm for checking satisfiability for our class of pure implicational formulas, giving our fixed parameter tractability result.

Algorithm 3.4 The Combined Algorithm

Input: *Pure implicational formula* ϕ *with each proposition letter occurring at most twice*

Output: *"Satisfiable" or "Unsatisfiable"*

Translate ϕ *to equisatisfiable formulas* $\{\theta_1, \ldots, \theta_{k'}\}$, *as in Section 2.*
For $i = 1, \ldots, k'$
  Build tree T_i *as follows:*
    Create a root r_i *labeled* $\neg\mathbf{f}$.
    For each disjunct $l_1 \wedge l_2 \wedge \cdots \wedge l_j$ *of the consequent of* θ_i,
      Create a child of r_i *labeled* $l_1, l_2, \ldots, l_j$.
      Mark that child unexplored
    While there are unexplored leaves:
      Pick any unexplored leaf w *and mark it* explored.
      If the leaf has an negative literal $\neg a$ *as a label*

and does note have **f** *as a label*
If there is a θ_j *with hypothesis* $\neg a$
(there can be only one such θ_j *by the 2-occurrence restriction)*
For each disjunct $l_1 \wedge l_2 \wedge \cdots \wedge l_j$ *of the consequent of* θ_j
Create a child of w *labeled* $l_1 \wedge l_2 \wedge \cdots \wedge l_j$
Mark that child unexplored.
Remove all nodes all of whose descendents are labeled **f**.
For each shadow pattern P_h, $1 \leq h \leq (k-1)^k$,
Make a copy T_i^h *of each of the original trees* T_i
Delete from T_i^h *all leaves which do not match shadow pattern* P_h.
Simplify T_i^h *as in Algorithm 3.1.*
If the simplification algorithm returned "unsatisfiable"
continue to next P_h
For each shadow pattern P_h, $1 \leq h \leq (k-1)^k$,
Identify the weakly connected components of P_h
For each weakly connected component W *of* P_h
Identify the cycle c *and the attached trees* t_g *of* P_h
For each tree t_g
For each node T_i^h *of* t_g, *starting with the leaves of* t_g
and working backwards to the root
If T_i^h *has children in* t_g
For each child T_j^h *of* T_i^h *in* t_g
For each leaf w *of* T_i^h
If w *is in the shadow of* every *leaf of* T_j^h
delete w *from* T_i^h.
Simplify W *as in Algorithm 3.1.*
If the simplification algorithm returned "unsatisfiable"
continue to the next P_h.
else
test the cycle c *for property* (‡) *of Theorem 3.3*
If property (‡) *fails for* c, *continue to next* P_h
Return "Satisfiable".
Return "Unsatisfiable".

Putting together the previous complexity results, we have that:

Theorem 3.5 *Satisfiability for pure implicational formulas with at most two occurrences of each proposition letter and at most* k *occurrences of* **f** *can be checked in time* $\mathbf{O}(k^k \cdot n^2)$. □

4 Boolean functions representable by pure implicational formulas

It is well known that there are Boolean functions which cannot be represented by pure implicational formulas, while any such functions can be represented by implicational formulas which allow for occurences of the Boolean constant **f**. We address the question of characterizing those Boolean functions which can be represented by pure implicational formulas and show that the number of such functions is surprisingly large.

A Boolean function $\varphi : B^n \to B$ is called an *implicant* of the Boolean function φ iff $\varphi \to \psi$ is a tautology.

Theorem 4.1 *A Boolean function $\psi : B^n \to B$ is representable by a pure implicational formula ϕ iff there is a Boolean variable z, which is an implicant of ψ*

Proof:

Suppose ψ is represented by the pure implicational formula ϕ. Then ϕ has the form $\phi_1 \to (\phi_2 \to \ldots(\phi_h \to z)\ldots)$, where z is the rightmost variable of ϕ.

It is easy to see that assigning z to true results in the value of true for ϕ. That is, z is an implicant of ϕ and so of ψ.

Now suppose the variable z is an implicant of ψ. That is, $z \to \psi$ is a tautology. Denote by $\psi_{z=\mathbf{f}} : B^{n-1} \to B$ the Boolean function defined by ψ when assigning z to false. Define $\psi_{z=\mathbf{t}}$ the same way, assigning z to true. Applying Shannon decomposition to ψ and z we obtain:

$$
\begin{aligned}
\psi &= (z \wedge \psi_{z=\mathbf{t}}) \vee (\bar{z} \wedge \psi_{z=\mathbf{f}}) \\
&= z \vee \psi_{z=\mathbf{f}} && \text{(because } z \text{ is an implicant of } \psi\text{)} \\
&= (\psi_{z=\mathbf{f}} \to z) \to z \\
&= (\phi_1 \to z) \to z && (\phi_1 \text{ is a Boolean formula representing } \psi_{z=\mathbf{f}} \\
& && \text{consisting of Boolean variables, the con-} \\
& && \text{stant } \mathbf{f} \text{ and the operator } \to.) \\
&= (\phi_2 \to z) \to z && (\phi_2 \text{is obtained from } \phi_1 \text{ by substituting ev-} \\
& && \text{ery occurence of } \mathbf{f} \text{ in } \phi_1 \text{ by the variable} \\
& && z. \text{ That is, } \phi_2 \text{ is a pure implicational for-} \\
& && \text{mula.)} \\
&= \phi,
\end{aligned}
$$

and ϕ is a pure implicational formula. □

Our proof of Theorem 4.1. tells us more about Boolean functions representable by pure implicational formulas. If ψ is representable by a pure implicational formula then there is a Boolean variable z, which is an implicant of ψ. If we fix however the value of z to **false**, the function $\psi_{z=\mathbf{f}} : B^{n-1} \to B$ can be an arbitrary Boolean function without any restriction. This immediately yields upper and lower bounds on the number i_n of Boolean functions $\psi : B^n \to B$ representable by pure implicational formulas.

Corollary 4.2 $\quad 2^{2^{n-1}} \leq i_n \leq n2^{2^{n-1}}$. □

From Corollary 4.2, the number of Boolean functions in n arguments representable by pure implicational formulas is about the square root of the number of all Boolean function in n arguments.

References

[1] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[2] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication: extended summary. *Proc. of the 22nd Symposium on Foundations of Computer Science*, pages 82–90, 1981.

[3] M. Dalal, and D. W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing letters* **44**:173–180, 1992.

[4] R. G. Downey, and M. R. Fellows. Fixed parameter tractability and NP-completeness. *Congressus Numeratum* 87, pages 161–178, 1992.

[5] G. Gallo, and M. G. Scutella. Polynomially solvable satisfiability problems. *Information Processing Letters* **29**:221–227, 1988.

[6] P. Heusch. The complexity of the falsifiability problem for pure implicational formulas. *Proc. 20th Int'l Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, J. Wiedermann, P. Hajek (Eds.), Prague, Czech Republic. *Lecture Notes in Computer Science* (LNCS **969**), Springer-Verlag, Berlin, pages 221–226, 1995.

[7] H. Kleine Büning. On generalized Horn formulas and k resolution. *Theoretical Computer Science* **116**, pages 405–413, 1993.

[8] J. Nešetřil and S. Poljak. On the asymptotic complexity of matrix multiplication. Commentationes Mathematicae Unversitatis Carolinae 26, pages 415–419, 1985.

[9] K. Truemper. Effective Logic Computation. To appear in 1996.