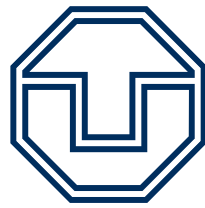


European Master in Computational Logic
Master Thesis

Backdoors for SAT

Marco Gario

October 2011

Supervisor: Dr. rer. nat. habil. Steffen Hölldobler
Advisor: Dipl.-Inf. Norbert Manthey

Declaration

Author:	Marco Elio Gustavo Gario
Matriculation number:	3730629
Title:	Backdoors for SAT
Degree:	Master of Science
Date of submission:	11/10/2011

Hereby I certify that this thesis has been written by me. Any help that I have received in my research work has been acknowledged. Additionally, I certify that I have not used any auxiliary sources and literature except for those cited in this thesis.

Abstract

The concept of backdoor was introduced to try to explain the good performances achieved on real world SAT instances by solvers. A backdoor is a set of variables that, once decided, makes the rest of the problem simple (i.e. polynomial-time).

In this thesis we provide a comprehensive overview on the state of the art of backdoors for SAT. Moreover, we study the relation between backdoors and parameterized complexity. In order to do so, we consider the problem of finding a smallest strong Horn-backdoor and we study it by means of the parameterized version of Vertex Cover. We conclude by presenting some interesting results obtained when analysing strong Horn-backdoors in instances from different domains.

Acknowledgements

This work wouldn't have been possible without the contribution of several people.

First of all, I'd like to thank my supervisor for showing me how to dot my I's and cross my T's. Thanks to the SAT group of the TUD for the many interesting discussion and to Robert for providing the implementation of the *atMost* constraint generator. A special thank goes to Norbert, for the help, the feedbacks, the many emails and useful discussions.

I'm really grateful to prof. Hölldobler, prof. Tessaris, Sylvia, Julia, Federica and all the people behind the EMCL. Organizing such an international program is not a trivial task, but I think the result is just amazing. In these two years I met many interesting people in Bolzano and Dresden, and I know that I will save the memories of these two years forever with joy.

Finally, I would like to thank my family and relatives because, no matter how far we are, they are always supportive of what I do. Thanks to Tatiana for the proof-reading and, more importantly, for being there. Thanks to my father, mother and sister for pushing me in pursuing my passions, and teaching me the value of hard work.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Structure of this work . . . 1
1.3. Contributions . . . 2

2. Background knowledge **3**
2.1. Boolean satisfiability (SAT) . . . 3
2.1.1. Notation . . . 3
2.1.2. SAT Solvers . . . 5
2.1.3. SAT Classes . . . 8
2.1.4. Subsolver . . . 14
2.2. Parameterized Complexity . . . 16
2.2.1. Basic definitions . . . 16
2.2.2. The hierarchy . . . 18
2.2.3. Practical FPT . . . 24
2.3. Chapter summary . . . 25

3. Backdoors **27**
3.1. Definitions . . . 27
3.2. Extending the concept of Backdoors . . . 31
3.2.1. Deletion backdoors . . . 31
3.2.2. Learning-sensitive backdoors . . . 32
3.2.3. Pseudo backdoors . . . 35
3.3. Backdoor-related concepts . . . 38
3.3.1. Backdoor trees . . . 38
3.3.2. Backdoor key . . . 40
3.4. Complexity results . . . 43
3.4.1. Problems definition . . . 43
3.4.2. Complexity analysis . . . 44
3.4.3. Parameterized complexity . . . 44
3.4.4. Modern SAT solvers and backdoors . . . 47
3.5. Experimental results . . . 50
3.5.1. Finding backdoors in practice . . . 50
3.5.2. Backdoors in benchmarks . . . 52
3.5.3. Exploiting backdoors . . . 53
3.6. Chapter summary . . . 54

4. Horn-Backdoors and Vertex Cover **55**
4.1. Strong Horn-backdoor detection is FPT 55
4.1.1. The algorithm 55
4.1.2. Considerations on the algorithm 58
4.2. Reduction to vertex cover 59
4.3. Vertex Cover implementations 61
4.3.1. Local search: COVER 61
4.3.2. FPT algorithm 62
4.3.3. Reduction to SAT 66
4.4. Experimental results 68
4.4.1. Goals and methodology 68
4.4.2. Benchmark statistics 69
4.4.3. Upper-bounds 71
4.4.4. Lower-bounds 71
4.4.5. Local search quality 73
4.4.6. Strong Horn-backdoor and features 75
4.5. Chapter summary 77

5. Conclusion **79**
5.1. Future directions for backdoors 79
5.2. Future directions for parameterized complexity 80
5.3. Summary of this work 80

Bibliography **84**

A. Instances with unverified lower-bound below 150 **85**

1. Introduction

1.1. Motivation

Boolean satisfiability (SAT) has become a major field in the AI community due to the impressive improvements achieved in solving techniques in the last decade. While the role played by clause learning and optimized data structures is well understood, we still have little knowledge of why some heuristics perform so incredible well on real world (i.e. non-random) instances. In particular, we experience run-times that are much better than the expected worst-case exponential behaviour. A possible explanation is that the complexity really lays only in few constraints; therefore, once these are satisfied, the rest of the problem becomes "easy" to solve. This concept has been formalized in two areas that turned out to be strongly related: backdoor variables and parameterized complexity.

The notion of *backdoor variables set* was first introduced by Williams et al. ([45]) and refers to a set of variables that, once they are assigned a value, leave the rest of the problem in some polynomial-time solvable class.

The field of *parameterized complexity*, developed by Downey and Fellows ([14]), is focused on confining the complexity of a problem into a subset of the problem called *parameter*. Of particular interest in the area of parameterized complexity is the class of fixed-parameter tractable (FPT) problems, that are problems which complexity is confined by the parameter only, and is influenced only polynomially by the size of the problem.

Early works by Nishimura, Szeider and others ([29],[43]) showed that parameterized complexity is a powerful tool to better understand backdoors. This thesis tries to continue on this direction by exploring the relation between backdoors and fixed parameterized tractable problems. In particular, we focus on a specific problem (strong Horn-backdoor detection) and we try to solve it by applying results from the parameterized complexity community.

1.2. Structure of this work

In writing this thesis, we assumed the reader to be familiar with some basic concepts of classical complexity (such as the classes P, NP and the concept of reduction) and propositional logic. Starting from there, we introduce all necessary background knowledge regarding SAT and parameterized complexity in Chapter 2. Chapter 3 will focus on providing a comprehensive overview of the state of the art on backdoors, focusing on

definitions and theoretical results (Section 3.1), complexity results with a focus on parameterized complexity (Section 3.4), and experimental results (Section 3.5). In Chapter 4 we will study more in detail the relation between strong Horn-backdoor detection and vertex cover; after reviewing the theoretical relation, we will take this problem as candidate to investigate the current situation in terms of Vertex Cover algorithms. Moreover, we will try to answer the question of whether local search algorithms are "good enough" to solve strong Horn-backdoor detection on SAT benchmarks. Finally, we conclude the chapter by showing the importance of having good datasets to study the relation between backdoors and other properties of our instances. Chapter 5 concludes this thesis, by providing a summary of this work, and highlighting some possible research directions for future work.

1.3. Contributions

Our intent in writing this thesis was to provide a small contribution to the SAT and parameterized complexity community. In order to do so we:

- Provide an overview on the state of the art of backdoors that is accessible to any reader with a minimum knowledge of propositional logic and classical complexity;
- Provide a (brief) introduction to parameterized complexity by looking into it from a rather practical point of view;
- Develop and release a framework to implement parameterized algorithms, together with some simple (yet, powerful) FPT algorithms for vertex cover;
- Build a dataset containing the upper- and lower-bound on the size of the smallest strong Horn-backdoor set for some instances taken from SAT benchmarks;
- Use the dataset to evaluate the performance of a local search algorithm for finding strong Horn-backdoors.

2. Background knowledge

This chapter is dedicated to provide the background knowledge needed in order to understand the rest of this work. Whenever we introduce a new concept, we provide a definition and an example to make it easier to follow the explanation. The only exception is for concepts that would require much more space to be explained. In this case we provide the reader with an intuitive understanding of the idea and provide references to go more in detail.

Section 2.1 covers the necessary definitions and concepts of the boolean satisfiability problem (SAT). The first part of the section is quite standard for a reader familiar with SAT, but starting from Section 2.1.3 we introduce some concepts that are fundamental to understand this work. Section 2.2 provides an introduction to parameterized complexity.

2.1. Boolean satisfiability (SAT)

We first provide a recall on the boolean satisfiability problem (SAT) then we provide the definition of several classes of problems (Section 2.1.3) and conclude by presenting the idea of subsolver (Section 2.1.4) introduced by Williams et al. ([45]). For a more detailed introduction to SAT we recommend the first Chapter of the *Handbook of Satisfiability* [5].

2.1.1. Notation

We assume an infinite set of propositional *variables* Var and we recall, from propositional logic, that variables can be combined through connectives (e.g. $\wedge, \vee, \neg$ $\top$,$\bot$) and recursively define *formulas* (e.g. $F_1 = v_1 \wedge \neg F_2$). Recall also that, for two formulas, $F_1 = F_2$ denotes syntactical equivalence between F_1 an F_2, while $F_1 \equiv F_2$ denotes semantic equivalence, i.e. F_1 and F_2 share exactly the same models. Given a formula F, we denote with $var(F)$ the set of variables occurring in F. A *partial interpretation* J is a partial mapping from $var(F)$ to the boolean values $\top, \bot$. We talk about *interpretation* I if the mapping is total. Given a formula F, we denote with $F[x/v]$ the formula obtained from F by replacing all the occurrences of x with v. Given a formula F and a (possibly partial) interpretation J, the *reduct* of F w.r.t. J ($F|_J$) is obtained by replacing each variable v in F with $J(v)$ (if defined) and simplifying the formula. F is *satisfiable* iff there exists an interpretation $I : var(F) \to \{\top, \bot\}$ s.t. $F|_I \equiv \top$; similarly F is *tautological* iff for all interpretations $I : var(F) \to \{\top, \bot\}$ it holds that $F|_I \equiv \top$. We call a formula that is not satisfiable *unsatisfiable* and a formula that is not a tautology *contingent*. The *satisfiability problem* (SAT) is the decision problem that, given

a formula F, returns whether F is satisfiable or not. This problem can be formulated for any kind of propositional formula involving any kind of connectives; nevertheless, most of the results in SAT have been proved for a particular class of formulas: formulas in conjunctive normal form. We call *literal* l a variable or its negation $\bar{l}$; a *clause* is a disjunction of literals and a formula in *conjunctive normal form* (CNF) is a conjunction of clauses. Since every formula can be transformed into an equivalent one in CNF, from now on, when talking about SAT, we assume our formulas to be in CNF. Since the order of the literals in a clause and the order of the clauses in the formula do not influence its satisfiability, we often consider a clause as a *set* of literals and a formula as a *set* of clauses. Computing the reduct of a formula in CNF is quite straightforward: we remove the clauses in which at least one literal is mapped to true, and we remove the literals mapped to false in the remaining clauses. Finally, we use the words interpretation and assignment with the same meaning, and we represent an interpretation compactly by listing the literals in it.

Example

F is a propositional formula not in CNF:

$$F = \neg(a \rightarrow (b \vee (d \leftrightarrow e)))$$

we obtain the CNF version by applying some well-known rules:

$$\begin{aligned}
F &= \neg(a \rightarrow (b \vee (d \leftrightarrow e))) && (1)\\
&\equiv \neg(\neg a \vee (b \vee ((d \rightarrow e) \wedge (e \rightarrow d)))) && (2)\\
&\equiv \neg(\neg a \vee (b \vee ((\neg d \vee e) \wedge (\neg e \vee d)))) && (3)\\
&\equiv a \wedge \neg b \wedge ((d \wedge \neg e) \vee (e \wedge \neg d)) && (4)\\
&\equiv a \wedge \neg b \wedge (d \vee e) \wedge (\neg e \vee e) \wedge (d \vee \neg d) \wedge (\neg e \vee \neg d) && (5)\\
&\equiv a \wedge \neg b \wedge (d \vee e) \wedge (\neg e \vee \neg d) && (6)
\end{aligned}$$

in step (2)-(3) we apply the definition of $\rightarrow$ and $\leftrightarrow$, in step (4) we push the negation inside (by using the DeMorgan rules and double negation elimination) and finally we apply distributivity in the last step. F is satisfiable, for example by the interpretation $I = \{a, \bar{b}, e, \bar{d}\}$.

This CNF formula can be represented in the set notation as:

$$F = \{\{a\}, \{\bar{b}\}, \{d, e\}, \{\bar{e}, \bar{d}\}\}$$

Given the partial interpretation $J = \{e\}$, in which we assign to the variable e the value *true*, we can compute the reduct of F w.r.t. J:

$$F|_J \equiv a \wedge \neg b \wedge \cancel{(d \vee e)} \wedge (\cancel{\neg e} \vee \neg d) \equiv a \wedge \neg b \wedge \neg d$$

Note that the conversion to an equivalent CNF can lead to an exponentially bigger formula:

$$\begin{aligned} G &= (a \wedge b) \vee (c \wedge d) \\ &\equiv (a \vee c) \wedge (b \vee c) \wedge (a \vee d) \wedge (b \vee d) \end{aligned}$$

therefore, complexity results expressed on a CNF formula, might not hold in the general case.

Finally, given a CNF formula F and a set of variables $V \subseteq var(F)$ we denote with $F - V$ the formula obtained from F by replacing in each clause of F all occurrences of x and $\neg x$ with $\bot$ (for each variable $x \in V$) and simplifying the clause. This operation intuitively "removes" the variables of V from F.

Example

Given the previous CNF formula F and $V = \{e\}$ we obtain:

$$\begin{aligned} F - \{e\} &= a \wedge \neg b \wedge (d \vee \cancel{e}) \wedge (\cancel{\neg e} \vee \neg d) \\ &= a \wedge \neg b \wedge d \wedge \neg d \end{aligned}$$

2.1.2. SAT Solvers

A SAT solver is an algorithm that decides the satisfiability of a given CNF formula F. An easy way to do this, is to explore the search space of all the $2^{|var(F)|}$ possible assignments of the variables occurring in F. A complete introduction to SAT solving algorithms is out of the scope of this work[1]; nevertheless, we will introduce two key ideas of modern SAT solvers:

- The widely used Davis Putman Logemann Loveland (DPLL [9]) procedure
- Clause Learning

DPLL

The Davis Putman Logemann Loveland (DPLL) procedure is a satisfiability algorithm based on complete search ([5]). In DPLL we explore a binary search tree where each node is associated with a (partial) interpretation. If there is a node which interpretation satisfies the formula, we declare the instance satisfiable. Conversely, if no such node exists, we declare the instance unsatisfiable.

[1] A detailed introduction to SAT solving algorithms can be found in Chapters 3 to 6 of [5]

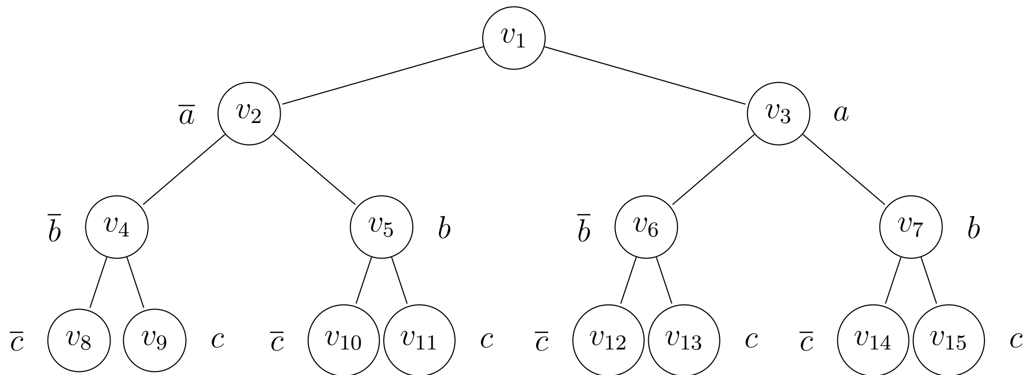
Definition 1 (**Decision Tree**). *A* decision tree *is a rooted binary tree T, such that every node in T is either a leaf or has exactly 2 children. The nodes of T, except for the root, are labelled with literals s.t. the following conditions are satisfied:*

- *two nodes v_i and v_j with the same father are labelled with complementary literals x and $\bar{x}$;*
- *the labels of the node on a path from the root to a leaf do not contain the same literal twice nor a complementary pair of literals.*

We indicate with J_v the (partial) interpretation composed by the labels of the nodes on the path from the root to the node v. If we consider a decision tree where all paths have length $n = |var(F)|$, for a given formula F, we see that we have exactly 2^n leaf nodes associated with the 2^n possible assignments. In this case, we notice that we can associate total assignments to leaf nodes and partial assignments to internal nodes.

Example

Consider the formula $F = (a \vee b) \wedge (b \vee \neg c)$. From the set $var(F) = \{a, b, c\}$ we can build the following decision tree:



Each path from the root (v_1) to a node gives us a unique interpretation. For example, J_{v_5} is the partial interpretation $\{\overline{a}, b\}$. To obtain a total interpretation we need to consider a leaf node: e.g. $J_{v_{10}} = \{\overline{a}, b, \overline{c}\}$.

In order to detect satisfiability of a formula, a complete search algorithm needs, in the worst case, to visit all leaves of the decision tree. For a CNF formula F, we notice that this is not always necessary, in fact we can detect unsatisfiability at some internal node and exclude the sub-tree originating from it; moreover, we have situations in which we do not need to branch, but can assign directly the value of a variable. This behaviour is formalized in DPLL by a set of rules:

- UNSAT
- SAT

- Unit
- Pure Literal
- Split

UNSAT / SAT A clause with all literals assigned to false is called empty clause and it can never be satisfied. If after assigning a variable we obtain an empty clause, then we can abandon this branch of the search, because the current partial interpretation can not be extended into a total satisfying interpretation. On the other hand, if we satisfy all the clauses, we can stop the search because we found a solution.

Unit Whenever we have an unsatisfied clause with only one unassigned literal l we know that, in order to satisfy the clause and therefore the formula, we must assign l to true. The Unit rule is based on this idea. After we assign a new variable and compute the reduct of the formula, we need to check whether there are any clauses that have only one unassigned literal (unit clauses); if this is the case, we assign the literal to true and repeat. The domino effect that this rule might cause is called *unit propagation* (UP).

Pure Literal (PL) If a literal never appears negated in the formula, we can assign the literal to true. By doing so, we satisfy all the clauses in which this literal appears, and we do not touch any other clause.

Split If none of the above rules is applicable, then we pick a variable and test whether either $F[v/\top]$ or $F[v/\bot]$ is satisfiable. This is the real branching step of our search algorithm.

Example

$$F|_I \equiv (a \lor b \lor c) \land (\neg d \lor e) \land (d \lor \neg e) \land \neg c \qquad I = \{\} \tag{1}$$

$$\equiv (a \lor b \lor \cancel{c}) \land (\neg d \lor e) \land (d \lor \neg e) \land \cancel{\neg c} \qquad I = \{\overline{c}\} \tag{2}$$

$$\equiv \cancel{(a \lor b)} \land (\neg d \lor e) \land (d \lor \neg e) \qquad I = \{\overline{c}, a\} \tag{3}$$

$$\equiv (\neg d \lor e) \land (d \land \neg e) \qquad I = \{\overline{c}, a\} \tag{4}$$

$$\equiv \cancel{(\neg d \lor e)} \land (\cancel{d} \lor \neg e) \qquad I = \{\overline{c}, a, \overline{d}\} \tag{5a}$$

$$\equiv (\cancel{\neg d} \lor e) \land \cancel{(d \lor \neg e)} \qquad I = \{\overline{c}, a, d\} \tag{5b}$$

$$F_I \equiv \top \qquad I = \{\overline{c}, a, \overline{d}, \overline{e}\} \tag{5a.1}$$

$$F_I \equiv \top \qquad I = \{\overline{c}, a, d, e\} \tag{5b.1}$$

The example formula (1) contains a unit clause, therefore the rule *unit* is applied in step (2). The resulting formula has a pure literal, namely a (3), that once assigned gives us (4). Since we cannot apply *unit* or *pure literal* anymore, we need to branch.

> Assuming we pick the variable d, we still can decide to branch on its positive or negative value, obtaining either (5a) or (5b). In both cases we end up with a unit clause and can extend the satisfying assignment with e (5a.1) or $\overline{e}$ (5b.1).
>
> Note that we performed only one branching step, and assigned three variables deterministically. Moreover, we never decided a value for the variable b. This means that the final solution is independent on the value of b, we can obtain a total interpretation by adding either b or $\overline{b}$ to the current interpretation.

These rules provide us with a complete search algorithm that still has $O(2^n)$ worst-time complexity, but it can often apply non-branching rules (UP or PL) and skip entire sub-trees (SAT/UNSAT). The extreme situation is given by instances that can be solved by applying only UP and PL. Since these two rules run in polynomial time, we obtain an exponential speed-up over the theoretical worst-case runtime.

Among other reasons, modern SAT solvers perform incredibly well by having smart data structures to detect Units (i.e. two watched literals [27]) and smart heuristics to pick variables and values for branching (e.g. activity heuristic [27]). In Section 3.4.4 we will discuss the importance of heuristics in the context of backdoors.

Clause Learning

The basic idea of *conflict driven clause learning* (CDCL) is to prune the search space by avoiding exploration of sub-trees that were already explored before. This is achieved by adding new clauses to the original instance. These clauses encode the information that a particular interpretation for a set of variables will lead to a contradiction and, therefore, it doesn't make sense to continue the search in that direction. For example, we might add the clause $(a \vee b)$ to force the solver to skip sub-trees where both a and b are false. The name of this technique is given by the fact that we learn new clauses only when we reach a conflict. In a DPLL solver, a conflict is given by the unit propagation assigning a literal that makes another clause empty. After the conflict we take into account all the decisions (branching points) that we made up to that point and we construct a new clause encoding some information about this branch of the search. How we obtain this clause depends on the *learning scheme*[2]. The inner working of the learning scheme is not relevant for our discussion. In Section 3.2.2 we will talk about learning-sensitive backdoors and see how learning influences the size of the backdoor, but these results are independent of the learning scheme.

2.1.3. SAT Classes

We call *class* a set of instances that share some property. We will now see some well-known classes of SAT: 2SAT, Horn, Anti-Horn, RHorn and UP+PL. Later on we will

[2] We will not go into technical details on how clause learning works, therefore we refer the reader to Chapter 4 of the Handbook of Satisfiability ([5]) or to the original publication ([39]).

introduce two particular classes: matched and cluster formulas. These classes will be relevant when studying complexity results in Section 3.4.

Base classes

2SAT, Horn and Anti-Horn are *syntactic* classes: we can decide if a CNF formula F belongs to a syntactic class C by simply looking for some syntactic characteristics.

2SAT $F \in 2SAT$ iff each clause of F has at most two literals,

Horn $F \in Horn$ iff each clause of F has at most one positive literal,

Anti-Horn $F \in AHorn$ iff each clause of F has at most one negative literal.

It should be clear, by the definition, that Horn and Anti-Horn are somehow complementary classes. Any theoretical result presented for the class Horn should be considered valid for the class Anti-Horn, unless otherwise stated.

Some other simple classes that are not syntactic but that we will consider later on are:

Renamable Horn (RHorn) We call a variable flipping for the variable x the substitution of occurrences of x in F with $\neg x$ and, similarly, of all occurrences of $\neg x$ with x. $F \in RHorn$ iff there exists a set of variables that, once flipped, makes the formula in Horn.

Unit Propagation + Pure Literal $F \in UP + PL$ iff it can be solved by applying only unit propagation and pure literal elimination to F (Section 2.1.2).

We are interested in all these classes, because there are polynomial time algorithms to solve them, and this will turn out to be a key concept in the definition of backdoors.

Example

The following formula F_a belongs to both 2SAT and Horn:

$$F_a = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2).$$

F_a is Horn, therefore, it is also RHorn since we can consider the flipping that leaves all variables unchanged. Note that there is also a variable flipping that keeps all the clauses Horn: $\{x_1, x_2, x_3\}$. By applying it we obtain:

$$(x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_1) \wedge (x_3 \vee \neg x_2)$$

where all clauses are Horn. An example of an *invalid* flipping is $\{x_1\}$:

$$(x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_1) \wedge (\neg x_3 \vee x_2)$$

where the first clause becomes non Horn.

F_a is not in UP+PL, since (i) there are no unit clauses and (ii) each variable appears both positive and negative.

An example formula in UP+PL is $F_b = (x_0 \vee \neg x_1) \wedge x_1 \wedge F_a$. F_b can be solved by applying PL to x_0 and then performing unit propagation:

$$F_b = (x_0 \vee \neg x_1) \wedge x_1 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2) \quad (7)$$

$$F_b|_{\{x_0\}} \equiv \cancel{(x_0 \vee \neg x_1)} \wedge x_1 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2) \quad (8)$$

$$F_b|_{\{x_0\}} \equiv x_1 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2) \quad (9)$$

$$F_b|_{\{x_0,x_1\}} \equiv \cancel{x_1} \wedge (\cancel{\neg x_1} \vee x_3) \wedge \cancel{(\neg x_2 \vee x_1)} \wedge (\neg x_3 \vee x_2) \equiv x_3 \wedge (\neg x_3 \vee x_2) \quad (10)$$

$$F_b|_{\{x_0,x_1,x_3\}} \equiv \cancel{x_3} \wedge (\cancel{\neg x_3} \vee x_2) \equiv x_2 \quad (11)$$

$$F_b|_{\{x_0,x_1,x_3,x_2\}} \equiv \top \quad (12)$$

In (8) we assign x_0 by pure literal, obtaining x_1 as unit clause. Therefore, unit propagation is sufficient to solve the formula (10-12).

When talking about deletion backdoor (in Section 3.2.1), it will become important to know whether a class is *clause induced* or *closed under clause removal*:

Definition 2 (**Clause induced**[7]). *A class $\mathcal{C}$ is said to be* clause induced *whenever a formula belongs to a class iff each of its clauses (viewed as a formula) belongs to the class; i.e. $F \in \mathcal{C} \leftrightarrow \forall G_i \in F.\ G_i \in \mathcal{C}$.*

A weaker property is being closed under clause removal:

Definition 3 (**Closed under clause removal** [7]). *A class $\mathcal{C}$ is* closed under clause removal *if for all formulas in the class, it holds that each subset of the clauses (when treated as a formula) belongs to the class; i.e. $\forall F \in \mathcal{C}$ it holds that $\forall G_i \in F.\ F \backslash \{G_i\} \in \mathcal{C}$.*

We observe that being clause induced implies being closed under clause removal, but the converse does not hold. Intuitively, clause induced means that we can focus on each of the clauses individually and not consider the interaction among them. 2SAT and Horn are clause induced, while RHorn is only closed under clause removal.

Example

Let's illustrate, by means of an example, first that Horn is closed under clause removal and later that it is also clause induced. Let's consider F_a from the previous example:

$$F_a = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2)$$

we already discussed the fact that F_a is 2SAT, Horn and RHorn. We notice how, by removing any of the clauses, the formula will remain in Horn, since the remaining

clauses will still be Horn clauses, e.g.:

$$F_a = \cancel{(\neg x_1 \vee x_3)} \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2)$$

this example illustrates how also RHorn and 2SAT are closed under clause removal.

To show that a class is *not* clause induced, we can take two formulas G_1 and G_2 belonging to the class, and see whether $G = G_1 \wedge G_2$ belongs to the class too:

$$\begin{aligned} G_1 &= (x_1 \vee x_2 \vee x_3) \\ G_2 &= (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ G &= G_1 \wedge G_2 \end{aligned}$$

G_1 is not Horn, but it is RHorn. In fact there are 4 valid flippings: $\{x_1, x_2\}$, $\{x_2, x_3\}$, $\{x_1, x_3\}$ and $\{x_1, x_2, x_3\}$. However, any of these flippings will make G_2 non Horn since at least two literals will become positive. Therefore, G is not a RHorn formula and it is a witness of the fact that RHorn is not clause induced.

Showing that a class is clause induced can be done by induction on the number of the clauses ([7]). Here we provide just an example to give an intuitive idea of why 2SAT and Horn are clause induced. Let's consider the following formulas:

$$\begin{aligned} F_a &= (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2) \\ F_b &= (\neg x_1 \vee \neg x_3) \\ F &= F_a \wedge F_b = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \end{aligned}$$

we already discussed F_a being Horn and 2SAT, and it is immediate to verify that also F_b belongs to Horn and 2SAT. Finally, note that all clauses in F have at most two literals, and at most one positive literal, that are the conditions for being 2SAT and Horn, respectively.

For the following discussion we need to point out that there might be some inconsistency in the use of these terms in the literature. Crama et al. ([7]) discuss formulas in disjunctive normal form (DNF)[3] and therefore they use the word *term* to denote the equivalent of a clause in CNF. We adapt their definition to clauses, as done (for *closed under clause removal*) by Dilkina et al. ([12], [11], [10]); however Szeider, Nishimura, Samer and Kottler (e.g. [28], [44], [23], [38], [15]) use *clause induced* to indicate the idea of *closed under clause removal*. We think that the intuitive understanding of the words *clause induced* is closer to the definition by Crama and therefore, in the following, we will stick to the definition provided above.

[3] In CNF we deal with conjunction of disjunctions, in DNF we deal with disjunction of conjunctions

Empty clause in syntactic classes

Syntactically defined classes are interesting because they are easy to check. Another important syntactic class, that is relevant for SAT, is the class of all formulas containing an empty clause ([11]):

Definition 4 (**Empty clause class**). *$\mathcal{C}_{\{\}}$ is the class of all formulas that contain the empty clause. For any class $\mathcal{C}$ of formulas, $\mathcal{C}^{\{\}}$ denotes the class $\mathcal{C} \cup \mathcal{C}_{\{\}}$*

In general, we can extend any class $\mathcal{C}$ in this way, for example 2SAT$^{\{\}}$ or Horn$^{\{\}}$. $\mathcal{C}_{\{\}}$ is, in general, not closed under clause removal: removing the empty clause might give us a formula not in $C_{\{\}}$. Therefore, neither 2SAT$^{\{\}}$ or Horn$^{\{\}}$ are closed under clause removal, even if 2SAT and Horn are.

Example

The following formula F is in 2SAT$^{\{\}}$ *only* because of the empty clause:
$F = \{\{a, b, c\}, \{\}\}$. The formula obtained by removing the empty clause ($F' = \{\{a, b, c\}\}$) is neither in 2SAT nor in $\mathcal{C}_{\{\}}$.

Matched and Cluster formulas

We introduce two additional polynomial classes that are studied by Nishimura et al. ([28]) and Szeider ([44]): cluster and matched formulas.

Definition 5 (**Hitting formula (HIT)**). *We say that a CNF formula F is a* hitting formula *(HIT) if any two clauses in F contain at least one complementary literal: i.e. $\exists l.l \in C_i$ and $\bar{l} \in C_j$ for all $C_i, C_j \in F$*

Definition 6 (**Cluster formula (CLU)**). *A CNF formula F is a* cluster formula *(CLU) if it is a variable-disjoint union of hitting formulas: i.e. $F = \cup G_i$ where $G_i \in HIT$ and $var(G_i) \cap var(G_j) = \emptyset$ for all $i \neq j$*

Example

Let's consider the following hitting formulas:

$$G_1 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_2)$$
$$G_2 = (y_1 \vee y_2 \vee y_3) \wedge (\neg y_2 \vee y_3) \wedge (\neg y_3)$$

we see that every two clauses have at least one complementary literal. Therefore, $F = G_1 \wedge G_2$ is a cluster formula since $G_1, G_2 \in HIT$ and $var(G_1) = \{x_1, x_2, x_3\}$, $var(G_2) = \{y_1, y_2, y_3\}$.

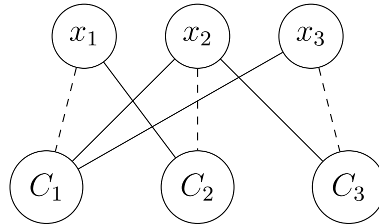
#SAT is the problem of counting how many interpretations (if any) satisfy a given formula. We can solve #SAT for hitting (and therefore cluster) formulas in polynomial time ([22]); therefore, we can solve SAT for hitting (cluster) formulas in polynomial time.

Definition 7 (Matched formula). *A CNF formula F is* matched *if there exists a total injective function $M : F \to var(F)$ that maps each clause of F to a variable occurring in F.*

Note that in the previous definition we require M to be both total (i.e. every clause is mapped to a variable) and injective (i.e. no two clauses are mapped to the same variable). Matched formulas are always satisfiable, since we can satisfy each clause by taking the correct assignment for the variable associated to it. Recognizing matched formulas can be done in polynomial time by reduction to bipartite graph matching. The idea is to build the incidence graph of the formula; this graph contains a node for each clause and variable in the formula. A clause is connected to all the variables occurring in it. Solving the bipartite graph matching problem on this graph provides the assignment of each clause to its variable.

Example

$F = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_2)$ has the following incidence graph:



We used a dashed line to indicate the mapping between clauses and variables. Once this matching is computed, we can easily build the satisfying assignment $I = \{x_1, x_2, \overline{x_3}\}$.

Definition 8 (Maximum deficiency). *A formula F has* maximum deficiency *r, if by removing r clauses from F we can obtain a matched formula.*

Definition 9 ($\mathcal{M}_r$). *We indicate with $\mathcal{M}_r$ the class of formulas with maximum deficiency r.*

Example

In the previous example we saw that $F \in \mathcal{M}_0$. We can build $F' \in \mathcal{M}_1$ by adding a clause to F:

$$F' = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_2) \wedge (x_1 \vee x_2)$$

where the last clause remains unmatched.

Matched formulas are interesting because their satisfiability can be decided in polynomial time, moreover also for r constant we can decide $\mathcal{M}_r$ in polynomial time ([42]). In fact, to solve a formula $F \in \mathcal{M}_r$ for a fixed r, we can generate (in polynomial time) all the formulas obtained by removing exactly r clauses from F. Since testing whether any of these new formulas is matched can be done in polynomial time, the overall problem is polynomial time.

2.1.4. Subsolver

The definition of subsolver, and many of the following definitions, were developed with constraint satisfaction problems (CSP) in mind. We can think of CSP as a generalization of SAT, in which the variables can assume different values from a domain D, and the constraints are subsets of the domains that a group of variables can assume at the same time. In the definitions we use "problem P" when we do not want to distinguish between CSP and SAT, and "formula F" when the difference matters.

We associate to the idea of a class $\mathcal{C}$ an algorithm, that we call subsolver C, that has some properties. In the future we will not make too much difference between the subsolver and the class it solves. For brevity, we will use the notation $C_1{+}C_2$ to indicate a subsolver for the class $\mathcal{C}_1 \cup \mathcal{C}_2$. We will come back on the definition of the subsolver in Section 3.2, when discussing generalizations of the backdoor concept, but for now we define a subsolver C as follows ([45]):

Definition 10 (**Subsolver**)**.** *We call C a* subsolver *if, given an input problem P:*

> *Tricotomy: C either rejects the input P, or "determines" P correctly (as unsatisfiable or satisfiable, returning a solution if satisfiable),* [4]
>
> *Efficiency: C runs in polynomial time,* [5]
>
> *Trivial solvability: C can determine if P is trivially true (has no constraints) or trivially false (has contradictory constraints),*
>
> *Self-reducibility: if C determines P, then for any assignment v of the variable x C determines $P[x/v]$.*

We explicitly assume that the trivial solvability check is performed only if the formula has not been rejected. This assumption, that is not part of the original definition by Williams et al., allows us to maintain the original definition of subsolver and still be able to distinguish between the classes $\mathcal{C}$ and $\mathcal{C}^{\{\}}$. For example, a formula $F \in 2\text{SAT}^{\{\}}$ might contain clauses that are not binary and an empty clause, therefore it would be rejected by a 2SAT subsolver but would be determined false by a $2\text{SAT}^{\{\}}$ subsolver.

[4]By weakening this axiom we obtain heuristic backdoors
[5]By removing this axiom we obtain pseudo backdoors

Example

A simple example of a subsolver is an algorithm to solve one of the polynomial time classes of SAT presented, for example 2SAT:

> *Tricotomy:* a 2SAT subsolver must reject an input formula F if it is not 2SAT (e.g. there's a clause with 3 literals), or determine F as SAT or UNSAT correctly;
>
> *Efficiency:* There are many algorithms to solve 2SAT in polynomial time, based on graph algorithms or look-ahead methods (e.g. [20])
>
> *Trivial solvability:* Our 2SAT subsolver needs to check whether the input formula F contains the empty clause (trivially unsatisfiable) or doesn't contain clauses at all (trivially satisfiable)
>
> *Self-reducibility:* In a CNF formula, after assigning a literal, we remove the clauses containing the literal, and remove the complementary literal from the remaining clauses. Therefore our simplified instance remains in 2SAT, since no clause can grow in size due to a new assignment. Therefore, our subsolver 2SAT can work on the simplified instance.

For what concerns the other base classes, we note that trivial solvability and self-reducibility can be achieved in all of them. Tricotomy is easy for Horn (being a syntactic class), but requires some effort for RHorn and UP+PL. Recognition of a RHorn formula can be done by reduction to 2SAT that is, as we discussed a polynomial time problem. Unfortunately for UP+PL we do not have a "smart" decision procedure, but we must run our UP+PL algorithm and reject if we get to a point in which branching is needed. For what concerns efficiency, we note that there are linear time algorithms for solving Horn and therefore also RHorn is polynomial time solvable, moreover, as discussed in Section 2.1.2, UP+PL also runs in polynomial time.

2.2. Parameterized Complexity

In classical complexity theory, we use the size of the input to define the worst case running time. In case of NP-complete problems (e.g. SAT), this translates into having exponential complexities in the size (n) of the input (e.g. $O(2^n)$). Nevertheless, in many cases, we have an exponential complexity only in a part of the input. The field of parameterized complexity ([16]) tries to shed some light on this aspect by introducing the idea of parameter. In this section we present the most important definitions from the parameterized complexity framework (Section 2.2.1) and provide an overview of the W-hierarchy (Section 2.2.2). We conclude this section with a short introduction on practical methods for FPT problems (Section 2.2.3).

2.2.1. Basic definitions

The most important idea of the parameterized complexity theory is the concept of *parameter*. The parameter shifts the focus of the complexity from the input size to some property of the problem. This allows us to perform a more accurate complexity analysis.

Definition 11 (**Parameterized problem**). *A* parameterized problem *is a language* $L \subseteq \Sigma^* \times \mathbb{N}$, *where* Σ *is a finite alphabet and the second component is called parameter.*

The key idea is that often we deal with problems whose exponential behaviour can be bounded by some parameter of the problem and, therefore, we should be able to have lower worst-time complexities. An example is the class of fixed-parameter tractable problems:

Definition 12 (**Fixed-parameter tractable (FPT)**). *A parameterized problem* L *is* fixed-parameter tractable *if* $(x, k) \in L$ *can be decided in time* $T(x, k) = f(k)p(|x|)$, *for a computable function* f, *a polynomial* $p(n)$ *and where* $|x|$ *is the size of the input.*

A simple example of an FPT problem is p-SAT:[6]

Definition 13 (**p-SAT**).

Instance: A propositional formula F

Parameter: Number k *of variables in* F, *i.e.* $|var(F)|$

Question: Decide whether F *is satisfiable*

Example

When talking about CNF, we noted that every formula can be converted into an equivalent one in CNF by accepting an (at most) exponential blow-up in the size of the formula:

$$F = (a \wedge b) \vee (c \wedge d)$$
$$F_{cnf} = (a \vee c) \wedge (b \vee c) \wedge (a \vee d) \wedge (b \vee d)$$

therefore, if we were to consider the size of the formula (as number of connectives) as input size $|x|$, we would obtain a different measure for F and F_{cnf}: $|F| = 3$ and $|F_{cnf}| = 7$. The associated p-SAT problem uses the number of variables as parameter k . Therefore, F and F_{cnf} have the same value for the parameter: 4. We indicate the parameter together with the instance by using a tuple: $(F, 4)$ and $(F_{cnf}, 4)$.

That p-SAT $\in$ FPT can be easily verified, by considering the worst-case complexity of checking the satisfiability of a formula with k variables and size m: $O(2^k * q(m))$. In fact, for each possible assignment (thus 2^k assignments) we test the satisfiability of F (that can be done in polynomial time bounded by the polynomial $q(m)$). In particular, $(F, 4)$ and $(F_{cnf}, 4)$ have the same exponential behaviour (2^4) that is what we intuitively expect when looking at these formulas.

Of course a few clarifications are needed ([13]):

- The function $f(k)$ can be any computable function, as long as it depends only on k. Therefore, we could run into incredibly steep functions like $2^{2^{2^k}}$. From a theoretical point of view, we note that the problem is still polynomial time; from a more practical point of view, we expect "real" problems to have "nice" parameterizations;
- Even though some parameterizations are easy to identify, there might be a different (and harder to find) parameterization for which we can have a lower worst-case running time.

Similarly to classical complexity, where we have polynomial time reductions from one problem to another, we define FPT-reductions:

Definition 14 (**FPT-reduction**). *Let L, L' be two parameterized languages. We say that $L \leq_{fpt} L'$ iff there is a mapping M, computable functions f and g and a constant c such that M maps an instance (G, k) to a new instance (G', k') so that*

- *$M(G, k)$ can be computed in time $\leq g(k)|G|^c$*
- *$k' \leq f(k)$*
- *$(G, k) \in L$ iff $(G', k') \in L'$*

By means of FPT-reductions we can create FPT-equivalence classes and compose them into a hierarchy ([16]). We present now the complete hierarchy, but we will introduce each level in Section 2.2.2:

$$\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq ... \subseteq \mathsf{W}[SAT] \subseteq \mathsf{W}[P] \subseteq \mathsf{XP}$$

[6] In general, for a well known decision problem X, we denote the parameterized version with p-X or k-X. In most cases the parameter is clear from the context, nevertheless each definition must include the parameterization explicitly.

where only FPT $\neq$ XP has been proved. XP has a clear characterization: it contains all problems that have a worst case runtime of $|x|^{f(k)}$. This means that the complexity depends exponentially on the parameter of the problem (k), but also on the size of the input ($|x|$). For what concerns the other levels, as for the polynomial hierarchy, if we show that two levels coincide, then the W-hierarchy would collapse. Moreover, if we show that FPT $\neq$ W[P] then we show P $\neq$ NP. This motivates the following assumption: whenever we deal with an algorithm that is W[i]-hard (for any i) we say that it is "unlikely" to be FPT.

2.2.2. The hierarchy

We try now to provide an intuitive understanding of the different levels of the hierarchy by presenting a representative problem for the main levels.

The first levels

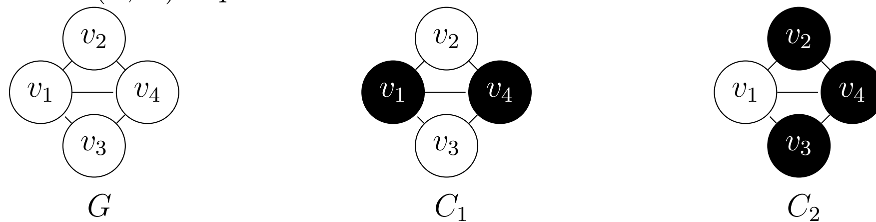
Since the initial goal of parameterized complexity was to study problems concerning graphs, the "canonical" problems for FPT, W[1] and W[2] are Vertex Cover, Independent Set and Dominating Set, respectively. We will not discuss neither Dominating Set nor Independent Set, but we will present other two problems that are complete for W[1] and W[2]: Short NTM Acceptance and Weighted CNF Satisfiability. We do so in the hope of having chosen problems whose non-parameterized version is familiar to the reader.

Definition 15 (**Vertex cover**). *Given a graph $G = (V, E)$, we call $C = \{v_1, .., v_n\} \subseteq V$ a* vertex cover *of G iff for all $e \in E$ there exists a $v_i \in C$ s.t. $v_i \in e$.*

In other words, we have a vertex $v_i \in C$ as representative of each edge in E. We call $|C|$ the size of the vertex cover.

Example

Consider $G = (V, E)$ depicted below:



then $C_1 = \{v_1, v_4\}$ and $C_2 = \{v_2, v_3, v_4\}$ are two different vertex covers for G of size 2 and 3, respectively.

Definition 16 (**k-Vertex Cover**).

Instance: A graph $G = (V, E)$

Parameter: A positive integer k,

Question: Does G have a vertex cover of size $\leq k$?

k-Vertex cover is one of the most studied parameterized problems. Currently, the best algorithm to solve it has a complexity of $O(1.2738^k + kn)$ ([6]). We will present some techniques for solving this problem in Chapter 4, when studying its relation with strong Horn-backdoor detection.

Definition 17 (**Short Non-deterministic Turing Machine Acceptance**).

Instance: A single-tape non-deterministic Turing Machine (NTM) M

Parameter: A positive integer k,

Question: Does M accept the empty string in $\leq k$ steps?

It is interesting to note that the non-parameterized version of the problem (the halting problem) is undecidable.

Example

The main issue with the halting problem is the ability of defining a method to detect whether *any* input machine M will accept or not. We try to provide two machines M and M' that implement two similar algorithms and show how different their acceptance conditions are. However, if we define a bound on the number of steps that they can perform, it becomes possible to devise an algorithm that decides whether they will accept. Recall that we say that a Non-deterministic Turing machine accepts if there exists a computation (i.e. a path in the transition diagram) that leads to a final state.

Let's consider a simple NTM M that performs the following operations:

- Write a random number N on the tape
- Accept if N is odd

Let's consider M' that performs the following operations:

- Write a random number N on the tape
- If N is odd add 1
- Accept if N is odd

It should be clear that M' will never accept, however we cannot decide this in general (halting problem). If we add a parameter k, solving Short NTM acceptance for (M, k) and (M', k) becomes simple: we just need to analyse all possible paths on the transition diagram of length up-to k.

For completeness, we include the transition diagrams for M and M'. We indicate the accepting state with a rectangular node. Each transition is labelled $X, Y, \{L, R\}$ where X is the symbol read from the tape and Y is the symbol written, and L, R indicates the direction of the movement of the head of the NTM. Recall that the tape is initialized with the empty symbol $\emptyset$.

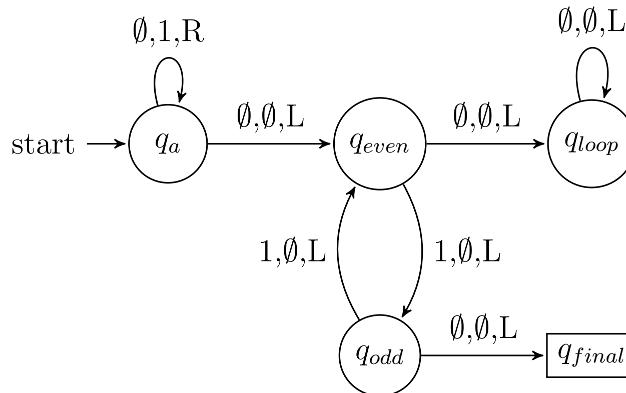


Figure 2.1.: Transition diagram of M

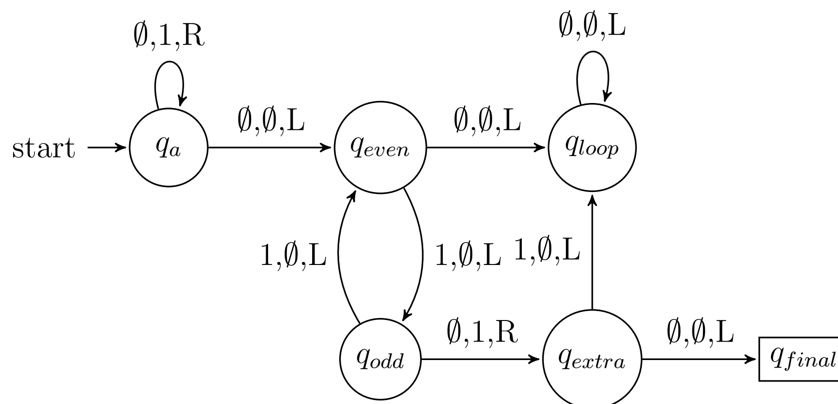


Figure 2.2.: Transition diagram of M'

Since we are interested in one accepting run, we can see that (M, k') accepts for all $k' \geq 4$. For example, $q_a, q_a, q_{even}, q_{odd}, q_{final}$ is an accepting run of size 4 (note that we count the transitions, not the states).

For M' we cannot, in general, determine that it will never accept. Note that the transition $q_{extra} \to q_{final}$ will never be taken because the transition $q_{odd} \to q_{extra}$ forces a 1 to be on the tape, therefore at the next step M' will transition to q_{loop}. However, fixed a k we can perform a search on all computations of size up to k and see that none of them accepts.

Definition 18 (**Weight**). *Given an interpretation I, we call* weight *of I the number of variables mapped to true.*

For example, $I_1 = \{x, \overline{y}, \overline{z}\}$ has weight 1 while $I_2 = \{x, y, z\}$ has weight 3.

Definition 19 (**Weighted CNF SAT**)**.**

Instance: A CNF formula F

Parameter: A positive integer k,

Question: Does F have a satisfying assignment of weight exactly k?

Example

Let's consider the instance $(F, 2)$ of Weighted CNF SAT with $F = (a \vee \neg b) \wedge (\neg a \vee b)$. $I = \{a, b\}$ is a solution for $(F, 2)$ because I is a satisfying assignment for F and has weight 2. Note, however, that $I' = \{\overline{a}, \overline{b}\}$ is not a solution for $(F, 2)$: I' is a satisfying assignment for F, but it has weight 0.

Property 1. *These problems are complete for the respective classes ([13]):*

- *Vertex Cover is* FPT*-complete*
- *Short NTM Acceptance is* W*[1]-complete*
- *Weighted CNF SAT is* W*[2]-complete*

When a problem is shown complete for some W-level, it is considered *unlikely* for that problem to be FPT. Differently from classical complexity, this does not correspond to less interest in the problem. In fact, we saw that two different parameterizations of SAT, namely p-SAT and Weighted CNF SAT, have different complexities: FPT and W[2]-complete. This suggests that it might be worth looking for a different parameterization of a problem to make it FPT.

The W-hierarchy

The W-hierarchy is defined by considering a particular syntactical restriction on formulas for SAT. For simplicity, we provide only some intuitive idea of the challenges of parameterized complexity leaving a more comprehensive exposition to textbooks ([16]).

Definition 20 (3-CNF). *A CNF formula F is in* 3-CNF *iff each of its clauses contains at most 3 literals.*

We define Weighted 3-CNF SAT as the parameterized version of the satisfiability problem for 3-CNF formulas.

Definition 21 (Weighted 3-CNF).

Instance: A 3-CNF formula F

Parameter: A positive integer k,

Question: Does F have a satisfying assignment of weight exactly k?

Theorem 1. *Weighted 3-CNF is* W*[1]-complete. [13]*

This result might be surprising, since Weighted CNF SAT is complete for W[2] and we know (from classical complexity) that there is a polynomial time reduction from any CNF formula to 3-CNF. Unfortunately, it is unlikely for a similar parameterized reduction to exist, since we cannot guarantee anything during the reduction about the parameter (weight) k. In fact, without knowing which variables of the original instance are mapped to true, we cannot compute how the weight will change in the the new 3-CNF instance.

Example

Let's take a generic CNF formula with one big clause:

$$F = ... \wedge (q_0 \vee q_1 \vee .. \vee q_n) \wedge ...$$

we recall that we can replace this clause with many clauses with at most 3 literals, by adding some auxiliary variables $z_1, .., z_m$, with $m = n - 2$:

$$F_{cnf} = ... \wedge (q_0 \vee q_1 \vee z_1) \wedge (\neg z_1 \vee q_2 \vee z_2) \wedge ... \wedge (\neg z_m \vee q_{n-1} \vee q_n) \wedge ...$$

Let's assume that the assignment $I = \{q_1\}$ is a satisfying assignment of weight 1 for F. We can build a satisfying assignment I_{cnf} for F_{cnf} starting from I. We note, in fact, that $I_{cnf} = \{q_1, \overline{z_1}, ..., \overline{z_m}\}$ satisfies all the additional clauses of F_{cnf}. In this particular case I_{cnf} has weight 1, but we will present another example in which this is not the case. Let's assume that F has only one satisfying assignment $I' = \{q_2\}$ of weight 1. We build the assignment for F_{cnf}: $I'_{cnf} = \{q_2, z_1, \overline{z_2}, .., \overline{z_m}\}$ that has weight

2. In general, depending on which q_i is satisfied in F, we need to satisfy all the z_j in F_{cnf} such that $j < i$. Therefore, we have no way to predict how the weight of the interpretation will change.

This does not mean that a reduction from Weighted CNF SAT to Weighted 3-CNF SAT does not exist: it could simply be based on a completely different principle. However, this is considered unlikely to be the case since such reduction would make the W-hierarchy collapse.

We define the rest of the hierarchy by considering formulas with $t + 1$ alternations of $\wedge$ and $\vee$ (product of sums) and we call the related problem Weighted t-PoS SAT, thus defining the level $\mathsf{W}[t]$ of the hierarchy. Finally, when we don't restrict syntactically the formula, we obtain Weighted SAT that we use to define the W[SAT] level.

Example

Given:

$$F_2 = \big((x_2) \vee (x_3)\big) \wedge \big((x_3) \vee (x_4) \vee (x_5)\big)$$
$$F_3 = \Big(\big(x_3\big) \vee \big(x_4\big) \vee \big(x_5\big)\Big) \wedge \Big(\big(x_3\big) \vee \big(x_2 \wedge (x_4 \vee x_4)\big)\Big)$$

For technical reasons, we need to consider F_2 as a conjunction of disjunctions of conjunctions of single literals; this way we see that F_2 is in 2-PoS and the associated Weighted 2-PoS SAT problem is in $\mathsf{W}[2]$. Similarly we see that F_3 has the shape $\wedge \vee \wedge \vee$ and therefore belongs to 3-PoS, that is in $\mathsf{W}[3]$.

Finally, if we consider a circuit of unbounded depth instead of a formula, we define Weighted Circuit SAT that is a complete problem for W[P]. Intuitively, we can think of an unbounded circuit as to a propositional formula in which we do not bound the number of alternations of $\wedge$ and $\vee$ and allow negation at any level.

Comparison with classical complexity To get a better intuitive understanding of the hierarchy, we can try to draw a comparison between its levels and the ones of the polynomial hierarchy: FPT can be associated with P, W[1] with NP and W[P] with PSPACE. In particular, the analogy for W[1] is strongly motivated by the completeness of Short NTM Acceptance for this class; moreover, when we consider PSPACE as the union of all levels of the polynomial hierarchy, we can see that there is some similarity with W[P]. Note, however, that this comparison is purely intuitive. In fact, there are problems that are NP-hard but for which the best parameterization makes them belong to W[P]. We will see one of these problems when discussing UP-backdoors in Section 3.4.3.

2.2.3. Practical FPT

We would like to obtain FPT algorithms for our problems. In order to achieve this, we can follow four strategies:

- Reduction to a known FPT problem (eg. Vertex Cover),
- Bounded search,
- Kernelization,
- Iterative compression

Reduction to a known FPT problem Vertex cover is one of the FPT problems that received more attention, and for which there are good solving algorithms[7] making it an ideal candidate for a reduction. We apply this idea in Chapter 4 when solving strong Horn-backdoor via reduction to vertex cover.

Bounded search If there is an existing algorithm to solve our problem that is based on a search tree, we can easily obtain an FPT algorithm by bounding the depth of this tree. This is the method used by Nishimura et al. ([29]) to prove that strong 2SAT and Horn backdoor detection are in FPT (see Section 4.1).

Kernelization Kernelization is based on the idea of pruning the search space by means of the parameter. As example, we can consider (one of) the kernelizations of k-Vertex Cover ([13]). Take a graph G; if G has a vertex v of degree[8] $> k$, then v must be part of the vertex cover.[9] Delete v. If in the new graph $G - \{v\}$ there are more than k vertices with degree $> k$ we can reject, since there cannot be a k-Vertex cover for G. If $G - \{v\}$ has more than k^2 vertices, reject. Otherwise run a complete search for the remaining k^2 vertices, or use a bounded search. We will use this kernelization (known as High Degree Kernelization) and provide more details in Section 4.3.

In practice, kernelizations are important to define good preprocessing techniques, perform early rejection, and can provide a good speed-up if we apply them during the search.

Iterative compression Given an instance and a solution of size k, an iterative compression algorithm ([33]) returns a new solution with $k' < k$ or shows that there is no better solution than the current one. This approach is particularly useful if we use heuristic or local search algorithms to quickly achieve a non-optimal solution of size k, and then iteratively improve it.

[7] A list of the state-of-the-art lower-bound complexities for many FPT problems is maintained at `http://fpt.wikidot.com/fpt-races`

[8] We call *degree* of a vertex the number of vertices it is connected to

[9] If we would decide not to take v, we would need to take all its neighbours. But since they are more than k this is impossible.

We will not use this technique in our work, but we think it might be interesting to provide an idea of how an iterative compression algorithm looks like. Let's consider a vertex cover C for the graph $G = (V, E)$. The idea of the iterative compression algorithm is to reduce by one the size of C at each iteration. In order to do so, we consider Y and S total partitions of C (i.e. $Y \cup S = C$ and $Y \cap S = \emptyset$) and try to replace all the vertices in S with $|S| - 1$ vertices from the vertices that are not currently in the cover ($V \setminus C$). This means that we keep in the cover the vertices Y and we modify only S. In order to do so, we construct G' by removing all vertices in Y from G. By doing so we remove also all the edges that are already covered by Y. Then we exchange each vertex in S for a neighbour that is not in S. If there is a vertex in S that has only neighbours in S then this choice of S cannot lead to a solution (is invalid). Remember, in fact, that we want to replace *all* vertices in S with new vertices. Since this procedure can be carried out in polynomial time ($O(m)$), the complexity lies in finding the correct partition of C into Y and S. By trying all possible partitions we end up with an algorithm that has complexity $O(2^{|C|}m)$ for each iteration and thus is FPT. For a more detailed explanation of this iterative compression for vertex cover we refer the reader to [19].

2.3. Chapter summary

In this chapter we provided a quite wide overview on SAT and parameterized complexity. We started by introducing SAT, the concept of base classes and some of their properties and we concluded by talking about subsolvers. The overview on parameterized complexity provided an introduction to the main theoretical definitions and an example problem for each of the first levels of the W-hierarchy.

For sake of completeness we introduced many concepts and definitions, even though some of them are just marginal to the understanding of the rest of this work. The base classes and the concept of subsolver will be central in Chapter 3. In Chapter 4 we will focus mainly on vertex cover as a parameterized problem, and we will present some examples that should provide a better understanding of FPT algorithms.

3. Backdoors

In this chapter we study the concept of backdoor in detail. We provide the classical definition (Section 3.1) together with some extensions (Section 3.2). Later on, we focus on complexity results for the associated backdoor detection problem (Section 3.4) both from a classical and parameterized point of view. Section 3.5 will focus on experimental work performed on backdoors by different authors. We conclude the section with a short summary.

3.1. Definitions

We follow the definition of backdoors given by Williams et al. ([45]), but we would like to point out that the same idea was proposed earlier by Crama et al. ([7]) under the name of *control set* variables.

We define a backdoor set with respect to a class of problems $\mathcal{C}$ for which there exists a subsolver C that meets the requirements of Definition 10 (ref. pg. 14): tricotomy, efficiency, trivial solvability and self-reducibility. Intuitively, a backdoor is a set of variables such that once their value is fixed the instance becomes simple to solve for the subsolver. As done in Section 2.1.4, we introduce these definitions for the more general constraint satisfaction problem (CSP); in this context P denotes a problem instance and D indicates the domain of the variables (e.g. $\top$,$\bot$ for SAT).

Definition 22 (Weak C-backdoor). *A non-empty subset B of the variables of the problem instance P is a* weak backdoor *w.r.t. the subsolver C for P iff there exists a partial interpretation $J : B \to D$ such that C returns a satisfying assignment of $P|_J$.*

Implicit in the definition of weak backdoor is the idea of P being satisfiable. To be able to work also on unsatisfiable instances, we introduce the notion of *strong backdoor*:

Definition 23 (Strong C-backdoor). *A non-empty subset B of the variables of the problem instance P is a* strong backdoor *w.r.t. the subsolver C for P iff for all partial interpretations $J : B \to D$, C returns a satisfying assignment or concludes unsatisfiability of $P|_J$.*

If P is satisfiable, then every strong backdoor is also a weak backdoor; but if P is unsatisfiable, it doesn't make sense to talk about weak backdoors. In the following, whenever the class $\mathcal{C}$ is not important, we simply talk about "backdoor", specifying "strong" or "weak" only if necessary.

Example

Let's consider the satisfiable formula F_0:

$$\begin{aligned} F_0 = {} & (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_4 \vee x_1) \wedge (\neg x_1 \vee x_6) \wedge \\ & (\neg x_1 \vee \neg x_6 \vee x_5) \wedge (\neg x_5 \vee x_4) \wedge (\neg x_5 \vee \neg x_6 \vee x_2) \end{aligned}$$

$B_1 = \{x_1\}$ is a weak UP+PL-backdoor for F_0, in fact if we consider the partial assignment $J = \{x_1\}$ we obtain:

$$\begin{aligned} F_0|_{\{x_1\}} \equiv {} & (\neg x_2 \vee x_3) \wedge \cancel{(\neg x_3 \vee \neg x_4 \vee x_1)} \wedge (\cancel{\neg x_1} \vee x_6) \wedge \\ & (\cancel{\neg x_1} \vee \neg x_6 \vee x_5) \wedge (\neg x_5 \vee x_4) \wedge \\ & (\neg x_5 \vee \neg x_6 \vee x_2) \qquad = \\ = {} & (\neg x_2 \vee x_3) \wedge x_6 \wedge \\ & (\neg x_6 \vee x_5) \wedge (\neg x_5 \vee x_4) \wedge (\neg x_5 \vee \neg x_6 \vee x_2) \\ F_0|_{\{x_1, x_6\}} \equiv {} & (\neg x_2 \vee x_3) \wedge x_5 \wedge (\neg x_5 \vee x_4) \wedge (\neg x_5 \vee x_2) \\ F_0|_{\{x_1, x_6, x_5\}} \equiv {} & (\neg x_2 \vee x_3) \wedge x_4 \wedge x_2 \\ F_0|_{\{x_1, x_6, x_5, x_2\}} \equiv {} & x_3 \wedge x_4 \\ F_0|_{\{x_1, x_6, x_5, x_2, x_4, x_3\}} \equiv {} & \top \end{aligned}$$

where at each step we have a unit clause (x_6, x_2, x_4, x_3) or a pure literal (x_5).

Similarly, $B_2 = \{x_1, x_2\}$ is a strong 2SAT-backdoor and therefore, since F_0 is satisfiable, it is also a weak 2SAT-backdoor. To verify this, we need to test the partial assignments $J_0 = \{x_1, x_2\}$, $J_1 = \{x_1, \overline{x_2}\}$, $J_2 = \{\overline{x_1}, x_2\}$ and $J_3 = \{\overline{x_1}, \overline{x_2}\}$. We can see that all of them produce a 2SAT formula:

$$\begin{aligned} F_0|_{J_0} &\equiv x_3 \wedge x_6 \wedge (\neg x_6 \vee x_5) \wedge (\neg x_5 \vee x_4) \\ F_0|_{J_1} &\equiv x_6 \wedge (\neg x_6 \vee x_5) \wedge (\neg x_5 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \\ F_0|_{J_2} &\equiv x_3 \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_5 \vee x_4) \\ F_0|_{J_3} &\equiv (\neg x_3 \vee \neg x_4) \wedge (\neg x_5 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \end{aligned}$$

Let's consider the unsatisfiable formula F_1:

$$\begin{aligned} F_1 &= (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg x \vee q \vee r) \wedge H \\ \text{where } H &= (\neg p_1 \vee \neg p_2 \vee q) \wedge (\neg q \vee a) \wedge (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & \quad (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \end{aligned}$$

$B_1 = \{x, q\}$ is a strong Horn-backdoor for F_1. We start by noting that H is already in Horn; since we are looking for a strong backdoor, we need to verify all possible

assignments for the variables in B_1:

$$\begin{aligned}
F_1|_{\{x,q\}} \equiv\ & a \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg r \vee a) \wedge \\
& (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \\
F_1|_{\{x,\overline{q}\}} \equiv\ & r \wedge (\neg p_1 \vee \neg p_2) \wedge (\neg r \vee a) \wedge \\
& (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \\
F_1|_{\{\overline{x},q\}} \equiv\ & p_1 \wedge p_2 \wedge a \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge \\
& (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \\
F_1|_{\{\overline{x},\overline{q}\}} \equiv\ & p_1 \wedge p_2 \wedge (\neg p_1 \vee \neg p_2) \wedge (\neg r \vee a) \wedge \\
& (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b)
\end{aligned}$$

Note how $F_1|_{\{\overline{x}\}}$ is already in Horn, but when we consider $\{x\}$ we need to add another variable to our Horn-backdoor, in this case q:

$$\begin{aligned}
F_1|_{\{\overline{x}\}} &\equiv p_1 \wedge p_2 \wedge H \\
F_1|_{\{x\}} &\equiv (q \vee r) \wedge H \\
F_1|_{\{x,\overline{q}\}} &\equiv r \wedge H \\
F_1|_{\{x,q\}} &\equiv H
\end{aligned}$$

The idea that a subset of a strong backdoor might constitute a weak backdoors will be explored in Section 3.3.1, when talking about backdoor trees.

Given a particular class C, for the same instance there might be several C-backdoors, possibly of different size. Therefore, we focus on “small” ones:

Definition 24 (Minimal backdoor). *A strong (resp. weak) C-backdoor B is called* minimal *iff there is no proper subset of B that is a strong (weak) $\mathcal{C}$-backdoor, i.e. $\forall B' \subset B$, B' is not a strong (weak) $\mathcal{C}$-backdoor.*

Intuitively, a *minimal* backdoor is no longer a backdoor as soon as we remove a variable from it, i.e. there is no redundant or unnecessary variable in it.

Definition 25 (Smallest backdoor). *A C-backdoor B is called* smallest *iff it is minimal and $|B| \leq |B'|$ for any minimal C-backdoor B'.*

Note that there might be more than one smallest backdoor. Intuitively, we can think of the relation between minimal and smallest backdoor as local- and global-minima in a search space where we try to minimize the size of B.

Example

Let's reconsider the formula F_1 of the previous example, and the Horn-backdoor $B_1 = \{x, q\}$. It should be clear, from the example, that neither $\{x\}$ nor $\{q\}$ are

strong Horn-backdoors for F_1. Therefore B_1 is a minimal strong Horn-backdoor for F_1. We can build a simple non-minimal backdoor by adding *any* of the remaining variables of F_1 to B_1: e.g. $B_2 = \{x, q, p_1\}$ is a strong Horn-backdoor, but $B_1 \subset B_2$ therefore it is not minimal.

$B_3 = \{q, p_1, p_2\}$ is another strong Horn-backdoor for F_1 and we can easily check its minimality: none of the subsets of B_3 is a strong Horn-backdoor for F_1. To do so, we show that the following partial interpretations $J_1 = \{q, p_1\}$,$J_2 = \{q, p_2\}$ and $J_3 = \{p_1, p_2\}$ do not lead to a Horn formula:

$$\begin{aligned} F_1 &= (x \lor p_1) \land (x \lor p_2) \land (\neg x \lor q \lor r) \land H \\ F_1|_{J_1} &\equiv (x \lor p_2) \land H \\ F_1|_{J_2} &\equiv (x \lor p_1) \land H \\ F_1|_{J_3} &\equiv (\neg x \lor q \lor r) \land H \end{aligned}$$

We know now that $B_3 = \{q, p_1, p_2\}$ is not the smallest strong Horn-backdoor for F_1, since $|B_1| < |B_3|$. We can actually see, by testing all the variables involved in the three non-Horn clauses, that there is no strong Horn-backdoor of size 1, therefore B_1 is a smallest strong Horn-backdoor for F_1. Note that, by replacing q with r in B_1, we obtain another smallest strong Horn-backdoor for F_1, namely $B_4 = \{x, r\}$:

$$\begin{aligned} F_1|_{\{x,r\}} &\equiv (\neg p_1 \lor \neg p_2 \lor q) \land (\neg q \lor a) \land (\neg q \lor \neg a \lor b) \land \\ &\quad (\neg q \lor \neg a \lor \neg b) \land (\neg a \lor b) \land (\lor \neg a \lor \neg b) \\ F_1|_{\{x,\overline{r}\}} &\equiv q \land (\neg p_1 \lor \neg p_2 \lor q) \land (\neg q \lor a) \land \\ &\quad (\neg q \lor \neg a \lor b) \land (\neg q \lor \neg a \lor \neg b) \\ F_1|_{\{\overline{x},r\}} &\equiv p_1 \land p_2 \land (\neg p_1 \lor \neg p_2 \lor q) \land (\neg q \lor a) \land (\neg q \lor \neg a \lor b) \land \\ &\quad (\neg q \lor \neg a \lor \neg b) \land a \land (\neg a \lor b) \land (\neg a \lor \neg b) \\ F_1|_{\{\overline{x},\overline{r}\}} &\equiv p_1 \land \lor \land q \land (\neg p_1 \lor \neg p_2 \lor q) \land (\neg q \lor a) \land \\ &\quad (\neg q \lor \neg a \lor b) \land (\neg q \lor \neg a \lor \neg b) \end{aligned}$$

3.2. Extending the concept of Backdoors

In the previous section we introduced the definition of backdoors as given by Williams et al. ([45]); in this section we look into three extensions of backdoors:

1. Deletion backdoors ([28]) (Section 3.2.1)
2. Learning-sensitive backdoors ([12]) (Section 3.2.2)
3. Pseudo backdoors ([35]) (Section 3.2.3)

The first two received much attention by the SAT community and are based on different definitions of backdoors; the latter is obtained, instead, by just weakening the requirements on the subsolver.

3.2.1. Deletion backdoors

Recall that, given a CNF formula F and a set of variables V, we indicate with $F' = F - V$ the formula obtained by replacing in each clause in F the occurrences of x and $\neg x$ with $\bot$ (for all $x \in V$) and simplifying the clause itself (Section 2.1.1).

Definition 26 (Deletion C-backdoor). *A non-empty subset B of the variables of the formula F is a* deletion backdoor *w.r.t. a class C for F iff $F - B \in C$. [28]*

Deletion backdoors extend the concept of backdoors by not considering the reduct of the formula, but the simplification of the formula obtained by removing some literals from it.

Example

Let's consider F_1 again:

$$\begin{aligned} F_1 = {} & (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg x \vee q \vee r) \wedge \\ & (\neg p_1 \vee \neg p_2 \vee q) \wedge (\neg q \vee a) \wedge (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \end{aligned}$$

then $B = \{q, a\}$ is a deletion 2SAT-backdoor for F_1. Note, in fact, that all the clauses of size 3 contain q or a:

$$\begin{aligned} F_1 - B = {} & (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg x \vee \cancel{q} \vee r) \wedge \\ & (\neg p_1 \vee \neg p_2 \vee \cancel{q}) \wedge (\cancel{\neg q} \vee \cancel{a}) \wedge (\cancel{\neg q} \vee \cancel{\neg a} \vee b) \wedge (\cancel{\neg q} \vee \cancel{\neg a} \vee \neg b) \wedge \\ & (\neg r \vee \cancel{a}) \wedge (\neg r \vee \cancel{\neg a} \vee b) \wedge (\neg r \vee \cancel{\neg a} \vee \neg b) \\ F_1 - B = {} & (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg x \vee r) \wedge \\ & (\neg p_1 \vee \neg p_2) \wedge (\bot) \wedge (b) \wedge (\neg b) \wedge \\ & (\neg r) \wedge (\neg r \vee b) \wedge (\neg r \vee \neg b) \end{aligned}$$

we will see, shortly, that B is also a strong 2SAT-backdoor.

Deletion backdoors are of interest because of the following property:

Lemma 1. *If the class $\mathcal{C}$ is closed under clause removal then every deletion C-backdoor is also a strong C-backdoor ([28])*

The converse does not usually hold, since assigning a variable from the strong backdoor usually leads to some additional simplification; for example, there are formulas for which the smallest strong RHorn-backdoor is exponentially smaller than any deletion RHorn-backdoor ([10]). Nevertheless, Crama et al. show that:

Lemma 2. *If the class $\mathcal{C}$ is clause induced then strong C-backdoor and deletion C-backdoor are equivalent ([7]).*

Recall that classes that are clause induced are, for example, 2SAT and Horn. This property is important because it is easier to verify whether a backdoor set B is a deletion backdoor rather than a strong backdoor. In fact, we just need to verify membership to the target class $\mathcal{C}$ of $F - B$, and not of all the possible $2^{|B|}$ assignments of B. This fact will become more evident when studying the complexity results in Section 3.4.

3.2.2. Learning-sensitive backdoors

As discussed in Section 2.1 one of the features that considerably improved SAT solving is clause learning. Recall that clause learning works by adding a clause to the underlying formula F after each conflict. To take this change into account, Dilkina et al. ([12]) introduce the idea of *learning-sensitive backdoor*. Before introducing this concept, we need to introduce an auxiliary definition:

Definition 27 (**Search tree exploration**). *Given a formula F, we call* search tree exploration *a list of literals $(l_1, .., l_n)$ such that $l_i \in \{v_i, \overline{v_i}\}$ with $v_i \in var(F)$. Additionally, some of these literals are marked as branching literals.*

Branching literals are the result of the application of the *split* rule (Section 2.1.2). Graphically, we indicate that l is a branching literal by writing $\dot{l}$. For our purposes, we are interested in knowing which are the branching variables of a search tree exploration.

Definition 28 (**Branching variable**). *Given a search tree exploration S, we call x a* branching variable *if $\dot{x}$ or $\dot{\overline{x}}$ appears in S.*

If we are given a search tree exploration, we can reconstruct the search performed by an algorithm on the decision tree of the given formula.

Example

Let's consider the search tree exploration $(\dot{x_1}, x_2, \overline{x_3}, \overline{x_1}, x_3)$ for the following formula:

$$F = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$$

this tells us that the search algorithm traversed the search space as follows:

$$\begin{aligned} F|_{\{x_1\}} &\equiv x_2 \wedge (\neg x_2 \vee x_3) \wedge \neg x_3 \\ F|_{\{x_1,x_2\}} &\equiv (x_3) \wedge (\neg x_3) \\ F|_{\{x_1,x_2,\overline{x_3}\}} &\equiv \bot \\ F|_{\{\overline{x_1}\}} &\equiv (\neg x_2 \vee x_3) \\ F|_{\{\overline{x_1},x_3\}} &\equiv \top \end{aligned}$$

Thinking about a DPLL algorithm, we can see this search tree exploration as follows: we start by assigning the first literal (x_1) and then we find the others by unit propagation; after assigning $\overline{x_3}$ we find a conflict, therefore we backtrack and change x_1 to $\overline{x_1}$, obtaining a satisfiable formula. We backtrack to x_1 because it is the closest literal marked as branching literal.

Exploiting the idea of search tree exploration, Dilkina et al. introduce the concept of learning-sensitive backdoors:

Definition 29 (**Learning-sensitive C-backdoor**). *A non-empty subset B of the variables of the formula F is a* learning-sensitive *C-backdoor for F iff there exists a search tree exploration S such that the following conditions are met:*

- *B and the set of branching variables of S coincide;*
- *a clause learning SAT solver branching only on the branching variables in S, in this order and with C as subsolver at every leaf of the search tree, either finds a satisfying assignment or proves F unsatisfiable.*

We first note that learning-sensitive backdoors are a particular type of strong backdoors: in fact, we run the subsolver C at every leaf of the search tree.

The definition of learning-sensitive backdoor is characterized by the search tree exploration order: this is the order in which the variables in B and their values must be chosen, in order to guarantee that we will reach all the conflicts needed to justify the addition of some clauses.

Example

Let's reconsider our example F_1:

$$\begin{aligned} F_1 = & (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee q) \wedge \\ & (\neg q \vee a) \wedge (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (\neg x \vee q \vee r) \wedge (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \end{aligned}$$

We say that $B = \{x\}$ is a learning-sensitive UP+PL-backdoor for F_1 for the search tree exploration $S = (\dot{\overline{x}}, p_1, p_2, q, a, b, x, \overline{q}, r, a, b)$. Note that B coincides with the branching variables of S, therefore the first condition of Definition 29 is already met. We need to verify the second condition. When we set $\overline{x}$, we obtain (by unit-propagation) $\{p_1, p_2\}$:

$$\begin{aligned} F_1|_{\{\overline{x}\}} \equiv & \ p_1 \wedge p_2 \wedge (\neg p_1 \vee \neg p_2 \vee q) \wedge (\neg q \vee a) \wedge \\ & (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b). \end{aligned}$$

In turn this leads to q that leads to a (since the second clause can be seen as $q \rightarrow a$):

$$\begin{aligned} F_1|_{\{\overline{x}, p_1, p_2\}} \equiv & \ q \wedge (\neg q \vee a) \wedge (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b). \end{aligned}$$

Finally $\{q, a\}$ make both b and $\neg b$ true, causing a conflict:

$$F_1|_{\{\overline{x}, p_1, p_2, q, a\}} \equiv b \wedge \neg b \wedge (\neg r \vee b) \wedge (\neg r \vee \neg b).$$

By using a 1-UIP learning scheme ([26]), we obtain the clause $\neg q$, from which we can build the new formula $F_1 \wedge \neg q$. Now when we set x and apply unit propagation to $F_1 \wedge (\neg q)$ we obtain immediately by unit propagation $\{\overline{q}\}$:

$$\begin{aligned} F_1|_{\{x, \overline{q}\}} \equiv & \ (\neg p_1 \vee \neg p_2) \wedge r \wedge (\neg r \vee a) \wedge \\ & (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b). \end{aligned}$$

We obtain r and a and both b and $\neg b$, concluding the unsatisfiability of the formula:

$$F_1|_{\{x, \overline{q}, r, a\}} \equiv (\neg p_1 \vee \neg p_2) \wedge b \wedge \neg b$$

The tree search exploration is important. In fact, $B = \{x\}$ would not have been an UP+PL-backdoor if we had considered x before $\overline{x}$:

$$\begin{aligned} F_1|_{\{x\}} \equiv & \ (\neg p_1 \vee \neg p_2 \vee q) \wedge (\neg q \vee a) \wedge \\ & (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (q \vee r) \wedge (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \\ F_1|_{\{x, \overline{p_1}\}} \equiv & \ (\neg q \vee a) \wedge \\ & (\neg q \vee \neg a \vee b) \wedge (\neg q \vee \neg a \vee \neg b) \wedge \\ & (q \vee r) \wedge (\neg r \vee a) \wedge (\neg r \vee \neg a \vee b) \wedge (\neg r \vee \neg a \vee \neg b) \end{aligned}$$

at this point UP+PL is not enough to decide the formula.

A final remark is that there are no strong UP+PL-backdoors of size 1 for F_1, and that one of the smallest is of size 2: $B' = \{x, q\}$. We will extend this intuition shortly, but in this example we want to point out that learning-sensitive backdoors actually work on a different version of the formula. In particular, F enhanced by the clauses learnt by the conflict in the search tree exploration.

The introduction of learning-sensitive backdoors is important to understand the interaction between search and clause learning in modern SAT solvers.

Dilkina et al. ([12]) show some interesting properties regarding the size of learning-sensitive UP-backdoors:

Lemma 3. *There are unsatisfiable SAT instances for which the smallest learning-sensitive UP-backdoors are exponentially smaller than the smallest strong UP-backdoor.*

Lemma 4. *There are satisfiable SAT instances for which there exists a learning-sensitive UP-backdoor that is smaller than the smallest strong UP-backdoors.*

Lemma 3 and Lemma 4 highlight the fact that clause learning is a key component for the success of modern SAT solvers, but they also tell us that we must take into account the effect of clause learning when using a SAT solver to find backdoors.

Lemma 5. *There are unsatisfiable SAT instances for which a search tree exploration can lead to exponentially smaller learning-sensitive UP-backdoors than a different one.*

With this lemma, we formalize the fact that the heuristics choosing the variables and their sign in our SAT solver are vital for the quality of the learnt clauses.

A final important result for learning-sensitive backdoors is related to classes that are closed under clause removal:

Lemma 6. *Clause learning does not reduce the size of weak backdoors with respect to syntactic classes that are closed under clause removal.*

This result tells us that learning-sensitive backdoors are not useful when studying weak backdoors for 2SAT and Horn.

3.2.3. Pseudo backdoors

Rossi et al. ([35]) introduce *pseudo backdoors* by weakening the *efficiency* constraint on the subsolver. In particular we are not interested anymore in a subsolver that can solve in polynomial time the problem instance, but we are just interested in being able to check in polynomial time whether the instance belongs to the class associated to the subsolver. To do so, they modify the definition of backdoor from a set of variables B to a partial assignment J, and define the concept of *backdoor condition*. This idea does not apply only to SAT but to CSP in general (Section 2.1.4) therefore, in the following, we use P to indicate the problem instance and D to indicate the domain of the variables.

Definition 30 (**Backdoor condition**). *Given a CSP P, a* backdoor condition *w.r.t. to a subsolver C is a (global) constraint π on the subset $B \subseteq V$ of the decision variables in P that are currently instantiated, such that if the partial assignment $J : S \subseteq V \rightarrow D$ satisfies π, then J is a backdoor in P for C. Determining if J satisfies π must be performed in polynomial time.*

Note that the authors state that J is a backdoor (instead of B) in order to provide a common definition of backdoor condition for both weak and strong backdoor.

The idea of the backdoor condition is to be able to detect the membership of a problem instance P in C during the solving process. We saw that for syntactic classes (such as 2SAT and Horn) this task is trivial, but for more sophisticated classes (e.g. UP+PL) we do not have any other option than running our solver and see whether it rejects. Of course, if our subsolver does not work anymore in polynomial time, we cannot guarantee a membership check in polynomial time.

Example

Let's define the class 2SAT+k as the class of 2SAT formulas with exactly k clauses that have more than 2 literals. Any $F \in 2SAT + k$ can be solved by a 2SAT subsolver after satisfying the k clauses with more than 2 literals. Therefore, we define the backdoor condition π to be composed of exactly these k clauses. If we find a partial assignment J such that J satisfies π, then J is also a 2SAT-backdoor for F.

Note however that this condition is quite restrictive, because it must be expressed on the set of instantiated variables.

From here it follows naturally the definition of *pseudo backdoor*:

Definition 31 (**(Weak) Pseudo C-backdoor**). *A non-empty subset B of the variables of the problem instance P, is a* pseudo C-backdoor *for P if for some $J : B \rightarrow D$, C returns a satisfying assignment of $P|_J$.* [1]

Similarly Rossi et al. define strong pseudo backdoors:

Definition 32 (**Strong Pseudo C-backdoor**). *A non-empty subset B of the variables of the problem instance P, is a* strong pseudo C-backdoor *for P if for all $J : B \rightarrow D$, C returns a satisfying assignment of $P|_J$ or concludes unsatisfiability of $P|_J$.*

By using this definition, we are allowed to use subsolvers that are not polynomial time. The authors suggest a first example in the *multiple knapsack* problem. This is an extension of the knapsack problem, in which two knapsacks are available. They claim

[1] The exact definition presented in [35] paper has an additional part stating:" [..] returns a satisfying assignment of $P|_J$ or concludes unsatisfiability of $P|_J$.". We believe this definition to be a typo, since in the previous section they introduce weak backdoors without the requirement of C returning $P|_J$ unsatisfiable.

that, once the first knapsack has been filled in a way that no other object can fit, then you just need to fill the other knapsack. This sub-problem is "simpler" in the sense that there are many optimized algorithms for the knapsack problem (e.g. dynamic programming approach); note, however, that this sub-problem still presents an exponential worst-case time complexity.

Finally we can extend the concept of backdoor condition to pseudo backdoor condition:

Definition 33 (**Pseudo backdoor condition**). *Given a CSP, P, a* pseudo backdoor condition *w.r.t. to a subsolver C is a (global) constraint π on the subset $S \subseteq V$ of the decision variables in P that are currently instantiated, such that if the partial assignment $J : S \subseteq V \to D$ satisfies π, then J is a backdoor in P for C. Determining if J satisfies π must be performed in polynomial time.*

Example

We define a CSP problem with n boolean variables and m integer variables with domain $[min, max]$. Once a value has been fixed for the m integer variables, we can solve the remaining problem by a trivial reduction to SAT, since we are dealing only with boolean variables.

Note how the sub-problem is still as hard as the original problem (NP), but we probably have better techniques to solve it.

In the same paper ([35]) Rossi et al. introduce *heuristic backdoors* by removing the *completeness* constraint on the subsolver: the solver might reject on instances that are satisfiable in order to work faster (e.g. local search). We are not interested in this kind of backdoor, but we point out that all pseudo backdoors are heuristic backdoors.

3.3. Backdoor-related concepts

In this section we focus on two measures defined on backdoors: backdoor trees ([38]) and backdoor key ([36]).

3.3.1. Backdoor trees

Backdoor trees are used by Samer and Szeider ([38]) to quantify the number of subsets of a strong backdoor that are weak backdoors. The intuition behind this idea is that, for strong backdoors, we need to test both the positive and negative assignment of each variable: for one assignment the single variable may already be a weak backdoor, but for the complementary assignment we may need to introduce another variable in the backdoor set (see example in Section 3.1).

We can model this behaviour with a decision tree (as described in Definition 1), associating to each leaf node a partial assignment that makes the formula belong to our target class. We can show that there are backdoor trees that have less than $2^{|B|}$ leaves nodes and this positive result allows us to test less assignments.

Definition 34 (Decision tree size). *Given a decision tree T, we indicate with $|T|$ the number of leaves of T and we call $s = log_2|T|$ the* size *of T.*

Extending our notation for formulas, we use $var(T)$ to indicate the variables appearing in T. Recall that we call J_v the partial interpretation expressed by the path that links the root to the node v.

Example

This is an example decision tree with $|T| = 6$ and size $log_2(6)$

v_1

$\overline{a}$ v_2 v_3 a

$\overline{b}$ v_4 v_{10} b $\overline{b}$ v_{11} v_5 b

$\overline{c}$ v_6 v_7 c $\overline{d}$ v_8 v_9 d

The partial interpretations for the leaf nodes v_8 and v_7 are $J_{v_8} = \{a, b, \overline{d}\}$ and $J_{v_7} = \{\overline{a}, \overline{b}, c\}$, respectively.

Definition 35 (C-backdoor tree). *A binary decision tree T (with $var(T) \subseteq var(F)$) is a* C-backdoor tree *of F iff $F|_{J_v} \in C$ for every leaf v of T.*

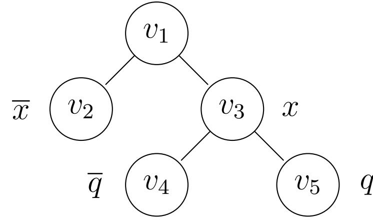
Definition 36 (**Smallest C-backdoor tree**). *A C-backdoor tree with the smallest number of leaves is a* smallest C-backdoor tree *for F.*

Example

We reconsider one of our running examples from Section 3.1:

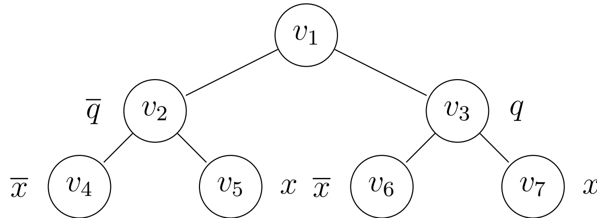
$$F_1 = (x \vee p_1) \wedge (x \vee p_2) \wedge (\neg x \vee q \vee r) \wedge H$$

and we recall that H is a Horn formula, $B_1 = \{x, q\}$ is a strong Horn-backdoor for F_1, but neither $\{x\}$ nor $\{q\}$ are. The smallest Horn backdoor tree for F_1 looks like:



In particular, we note that there are 3 leaf nodes (v_2,v_4,v_5) and therefore 3 partial interpretations that lead to a Horn formula: $J_{v_2} = \{\overline{x}\}$, $J_{v_4} = \{x, \overline{q}\}$ and $J_{v_5} = \{x, q\}$. Without using the concept of backdoor trees, we would end up testing more assignments, in this case $2^{|B|} = 4$.

Note that the order of the variables is important for the size of the tree, a bigger backdoor tree for F_1 is obtained by considering q before x, and has 4 leaf nodes:



The previous example should provide an intuitive understanding of the following property ([38]):

Property 2. *Let $\mathcal{C}$ be a base class and F a CNF formula. If B is a smallest strong C-backdoor for F and T is a smallest C-backdoor tree of F, then:*

$$|B| + 1 \leq |T| \leq 2^{|B|}$$

This property states that, in some cases, we can test exponentially fewer assignments if we use a backdoor tree to enumerate them. A secondary result, less important but still interesting, is that smallest strong backdoors are in general bigger than weak backdoors. In fact each leaf node in the backdoor tree justifies the existence of a weak backdoor.

Only if the tree is perfect (all paths from the root to each leaf have the same length) the strong backdoor and the weak backdoors coincide in size.

3.3.2. Backdoor key

Latin squares A Latin square is composed by $n \times n$ cells that can assume n different values, but each value must occur exactly once in each row and in each column. A common example of Latin square is the Sudoku puzzle.

Example

1	2	3
2	3	1
3	1	2

is a Latin square of size $n = 3$.

Given a partially filed square, deciding whether it can be completed into a Latin square is an NP-complete problem ([4]).

A sub-problem of Latin square completion is Quasi-group with holes (QWH). In this version of the problem, we start from a complete Latin square (also called quasi-group) and we remove some elements from it. Thus, the resulting problem instance is guaranteed to be satisfiable.

Example

1	2	3	
2		4	1
	4	1	2
4	3		1

is a QWH of size $n = 4$ with 4 holes.

Ruan et al. ([36]) studied the relation between backdoor size and the hardness in QWH problems, and noted that the size of the backdoor is correlated with *hardness*, where by hardness they consider the average solving time. It is important to stress out that they do not compute smallest backdoors but used a local search algorithm to find minimal zChaff-backdoors. This means that they run several times the zChaff SAT solver and register the decision literals and, afterwards, minimize these non-minimal backdoor sets.

To better measure the hardness, they introduce the concept of backdoor key and backdoor key fraction, and show that the latter has a good correlation with hardness of problems like QWH, but it is not a good predictor of hardness for problems like planning or bounded model checking.

The idea of backdoor key is related to a particular kind of dependency among variables. In particular they define it as follows:

Definition 37 (**Dependent Variable**). *A variable $v \in var(F)$ is a* dependent variable *of F w.r.t. a partial assignment J if either $F|_J \wedge v$ or $F|_J \wedge \neg v$ is satisfiable (but not both).*

Example

Let's consider the formula $F = (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_3)$. We say that x_2 and x_3 are dependent variables of F with respect to the partial assignment $J = \{x_1\}$. Indeed we easily see that $F|_{\{x_1\}} \equiv x_2 \wedge (x_2 \rightarrow x_3)$ that leads us to $F|_{\{x_1,x_2\}} \equiv x_3$, forcing x_3 to be true.

It is important to note that the concept of *dependency* is not related to the sub-solver; in the simple example above we can see how x_3 follows from x_1 by unit propagation, but this is not always the case:

$$F' = x_1 \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee x_3) \tag{1}$$
$$F'|_{\{x_1\}} \equiv (\neg x_2 \vee x_3) \wedge (x_2 \vee x_3) \tag{2}$$
$$F'|_{\{x_1,x_2\}} \equiv (\cancel{\neg x_2} \vee x_3) \wedge \cancel{(x_2 \vee x_3)} \equiv x_3 \tag{3}$$
$$F'|_{\{x_1,\overline{x_2}\}} \equiv \cancel{(\neg x_2 \vee x_3)} \wedge (\cancel{x_2} \vee x_3) \equiv x_3 \tag{4}$$

at step (2) we are forced to branch on a variable, for example x_2. Since for both x_2 and $\overline{x_2}$, we end up with x_3, we can say that x_3 is a dependent variable in F w.r.t. the partial assignment $J = \{x_1\}$.

Note how the dependency is defined for a particular partial assignment J and does not hold in general. Moreover, dependent variables are defined only for satisfiable instances.

In the sequel, let B be a backdoor for F.

Definition 38 (**Backdoor key set**). *A backdoor variable $v \in B$ is in the* backdoor key set *of B w.r.t. an interpretation $J : B \rightarrow \{\top, \bot\}$ iff v is a dependent variable in F w.r.t. J', where J' is obtained from J by removing the information concerning v.*

The authors define an additional measure related to backdoor sets, indicating how many variables in the backdoor are also in the backdoor key set:

Definition 39 (**Backdoor Key Fraction**). *Given a backdoor set B and its backdoor key set $B' \subseteq B$, the* backdoor key fraction *is the ratio of the size of the backdoor key to the size of the corresponding backdoor set: $frac = \frac{|B'|}{|B|}$.*

The final conclusion of the paper is that the backdoor key fraction is a good indicator of hardness for problems like QWH and morphed graph colouring ([36]), but not for problems like planning and model checking, since the backdoor key set in these problems has usually size 0. As a final remark, we would like to stress that these results are

somehow specific: the authors consider only zChaff-backdoors and define all experiments with respect to these backdoors, therefore other kinds of backdoors might show a different behaviour; moreover, they use “incomplete” data since they do not look neither for *all* backdoors nor for smallest backdoors.

3.4. Complexity results

In this section we introduce complexity results that are relevant for the study of backdoors. We start this section by presenting the results related to classical complexity analysis. Later on we show how parameterized complexity allows us to have a finer-grade distinction of the complexity of our problems. Finally, we present the original algorithms by Williams et al. that were designed to explain how (even though not explicitly) modern SAT solvers already implement backdoor detection.

3.4.1. Problems definition

In the following we study the complexity of two main problems: C satisfiability and *C-backdoor detection*.

Definition 40 (C **satisfiability**).

Input: A problem instance P belonging to the class C ($P \in C$)

Question: Is P satisfiable?

We presented in Section 2.1.3 some classes for which satisfiability can be decided in polynomial (or even linear) time. Recall, that the classical definition of backdoors (by Williams et al.) requires C satisfiability to be polynomial time, while pseudo backdoors do not impose this restriction.

When talking about satisfiability, we might be interested in knowing at least one satisfying assignment:

Property 3. *Given a formula $F \in C$ with n variables, if C satisfiability is in P, then we can obtain a satisfying assignment for F in polynomial time. In particular, if C satisfiability is bounded by a polynomial $p(n)$, we can obtain a satisfying assignment for F in $O(n * p(n))$.*

This property simply states that we can find an assignment for our formula, by assuming a literal l and testing satisfiability of $F' = F \wedge l$: if F' is unsatisfiable we add $\overline{l}$ to our partial interpretation J, otherwise we add l. We simply repeat this operation for each variable left in $F|_J$, obtaining a total interpretation.

We define the problem of detection for classical backdoors and backdoor trees (Section 3.3.1). We do not consider pseudo backdoors, since complexity results for this class depends on the complexity of the subsolver.

Definition 41 (**{Weak, Strong, Deletion}** C**-backdoor detection**).

Input: A problem instance P and an integer $k \geq 0$

Question: Does P have a {weak, strong, deletion} C-backdoor of size at most k?

Definition 42 (C**-Backdoor tree detection**).

Input: A problem instance P and an integer $k \geq 0$

Question: Does P have a C-backdoor tree with at most k leaves?

3.4.2. Complexity analysis

We state the following complexity upper-bound:

Lemma 7. *Given a formula F with n variables and an integer $k \geq 0$, {weak, strong, deletion} C-backdoor detection is in* NP.

Proof. The proof follows from the fact that membership in C can be checked in polynomial time. For weak and strong backdoors, we can devise an algorithm that selects a subset B ($B \subseteq var(F)$) of size k in linear time, generates all possible partial assignments $J_i : B \to \{\top, \bot\}$ ($1 \leq i \leq 2^k$) and checks whether $F|_{J_i} \in C$. For weak backdoors we can stop as soon as we find a satisfying assignment, while for strong backdoors we need to try them all. Nevertheless, the worst case complexity is the same: $O(2^k p(n))$ with $p(n)$ polynomial bounding the runtime of the membership check of the subsolver C. For deletion backdoors we just need to guess the set V of variables to be deleted (with $|V| = k$), and check whether $F - V \in C$. This can be done in $O(p(n))$. $\square$

Lemma 7 does not provide any information concerning the hardness of this problem. We summarize the following hardness results for C-backdoor detection for different classes:

Class	Weak	Strong	Deletion	Tree
2SAT,Horn	NP-Hard [29]	NP-Hard [29]	NP-Hard [29]	NP-Hard [38]
RHorn		NP-Hard [7]	NP-Hard [7]	NP-Hard [38]
$2SAT^{\{\}}$, $Horn^{\{\}}$		NP-Hard,coNP-Hard [11]		
Cluster				
Matched	NP-Hard [44]	NP-Hard [44]	NP-Hard [44]	
UP,PL,UP+PL				

These results are not surprising. In fact, if backdoor detection wasn't NP-hard, we could solve SAT in polynomial time. We will explore this aspect in more detail in the next section.

Since detecting a backdoor is as hard as solving SAT, we dedicate the next section to a different formalization that will lead us to some improvement.

3.4.3. Parameterized complexity

We saw how backdoor detection is NP-hard for all the classes that we are interested in. We are now interested in seeing whether parameterized complexity analysis can give us more information about the different classes. We introduce the parameterized version of the detection problem:

Definition 43 ({Weak, Strong, Deletion, Tree} k-C-backdoor detection).

Input: A problem instance P

Parameter: an integer $k \geq 0$, size of the backdoor

Question: Does P have a {weak, strong, deletion,tree} C-backdoor of size at most k?

We first note that, in general, this problem is in XP for small values of k ($k << n$): in order to find a C-backdoor of size k we need to try all 2^k assignments of all $\binom{n}{k} = \frac{n!}{k!(n-k)!} \approx n^k$ possible subsets of variables of size k. Therefore we obtain an algorithm that runs in time $O(2^k * n^k)$ that is the runtime of XP.

This result is an upper-bound for our parameterized complexity analysis, better lower bounds are summarized in the following table:

Class	Weak	Strong	Deletion	Tree
2SAT	W[2]-complete [29]	FPT [29]	FPT [29]	FPT [38]
Horn	W[2]-complete [29]	FPT [29]	FPT [29]	FPT [38]
RHorn		W[1]-hard[2]	FPT [32][1]	
Cluster		W[2]-hard [28]	FPT [28]	
Matched $\mathcal{M}_r$	W[2]-hard [44]	W[2]-hard [44]	W[2]-hard [44]	
UP,PL,UP+PL	W[P]-complete [43]	W[P]-complete [43]		
$2SAT^{\{\}}$,$Horn^{\{\}}$,$RHorn^{\{\}}$		W[1]-hard[2]		

The table should be read as follows: all problems that are FPT can be solved quite efficiently, all the others are "unlikely" to have a solution that is better than brute force. In particular, for Horn and 2SAT we have algorithms that run in $O(2^k n)$ and $O(3^k n)$ respectively ([29]). Recall that RHorn is not clause induced but closed under clause removal, therefore we have that (Lemma 1) every deletion backdoor is also a strong backdoor but not the converse. Since deletion RHorn-backdoor detection is FPT, we have an efficient way to find "some" strong backdoors of size at most k (i.e. the subset of strong backdoors that are also deletion backdoors) but we cannot efficiently show that there is no strong backdoor of a particular size.

UP+PL-backdoor detection is in W[P], that is the hardest[3] class in the W-hierarchy. This result tells us that UP+PL defines a class for which backdoor detection is quite hard. This matches our expectation. For all the other classes, we have a syntactical characterization that makes it easy to perform membership check without running the real solver. However, as we already noted, we cannot find such simple characterizations for UP+PL, but we are forced to run the subsolver and see whether it rejects or accepts.

An additional interesting result for FPT membership of backdoor tree detection, was proved by Samer and Szeider ([38]):

[1] Razgon and O'Sullivan show that almost 2-SAT is FPT, and deletion RHorn is equivalent to almost 2-SAT [38].

[2] Section 13.4 of [5]; the result for strong RHorn follows from a reduction from CLIQUE, that is W[1]-complete

[3] Under the theoretical assumption that each level in the hierarchy differs from the previous

Definition 44 (**Loss-free kernelization**). *We say that a base class* $\mathcal{C}$ *admits a* loss-free kernelization *if there exists a polynomial-time algorithm that, given a CNF formula* F *and an integer* k*, either correctly decides that* F *has no strong* $\mathcal{C}$*-backdoor set of size at most* k*, or computes a set* $X \subseteq var(F)$ *s.t. the following conditions hold: (i)* X *contains all minimal* $\mathcal{C}$*-backdoor sets of* F *of size at most* k*; (ii) the size of* X *is bounded by a computable function that depends on* k *only.*

Property 4. $\mathcal{C}$*-backdoor tree is FPT for every base class* $\mathcal{C}$ *that admits a loss-free kernelization.*

Backdoor as parameter

In the framework of parameterized complexity we can define different parameterizations for the same problem. We already saw p-SAT and W-SAT as characterizations of SAT that are FPT and W[2]-complete, respectively. If we use the backdoor as parameter we obtain the following problem:

Definition 45 (**bd-SAT**).

Input: A formula F *and a* $\{$*weak, strong, deletion*$\}$ *backdoor set* B

Parameter: The size of the backdoor $k = |B|$

Question: Is F *satisfiable?*

The following result follows from the definition of backdoor:

Theorem 2. *Given a formula* F *and a* $\mathcal{C}$*-backdoor* B*, bd-SAT for* F *and* B *is FPT for any class* $\mathcal{C}$*.*

Proof. Given a backdoor B of size $k = |B|$ for F, we need to test only 2^k possible assignments to decide the satisfiability of F. If B is a strong backdoor, then all the 2^k partial interpretations $J_i : B \rightarrow \{\top, \bot\}$ will provide us with a reduct $F|_{J_i}$ that can be solved by the subsolver $\mathcal{C}$. Since the subsolver, by Definition 10, runs in polynomial time, the complexity of this procedure is $O(2^k * p(n))$ for some polynomial p. This shows that bd-SAT is FPT for strong backdoors. For deletion backdoors we perform a similar reasoning, we run the subsolver $\mathcal{C}$ on all the 2^k reducts to verify the satisfiability of the formula. The case of weak backdoors is actually trivial: a formula can have a weak backdoor iff it is satisfiable. Assuming that we don't have the information on the type of backdoor, we can still run our subsolver $\mathcal{C}$ on all the 2^k reducts and decide F. □

This result justifies the NP-hardness of backdoor detection. In fact, under the theoretical assumption that $\mathsf{P} \neq \mathsf{NP}$, if we could solve backdoor detection in polynomial time, then we could decide SAT in polynomial time. Since SAT is NP-complete, this would imply that $\mathsf{P} = \mathsf{NP}$. The results of Section 3.4.2 are interesting because they provide a direct proof of the NP-hardness.

Note that Theorem 2 doesn't require the backdoor to be minimal. In fact, a trivial backdoor is composed by all the variables in the instance. By doing so, we can see that p-SAT is a particular case of bd-SAT. In general, we are interested in having a small value of k only because this influences exponentially our complexity.

3.4.4. Modern SAT solvers and backdoors

In the original paper, Williams et al. try to justify the good performance of SAT solvers by introducing the idea of backdoors. The idea is that modern SAT solvers behave so well (especially in structured instances) because they are good at finding and exploiting backdoors. We just saw that backdoor detection is NP-hard, therefore we need to justify how SAT solvers can find and exploit backdoors in an efficient way. To explain this, they introduce three algorithms ([45]):

- Deterministic
- Randomized
- Heuristic

In the following we take a formula F with n variables and a smallest C-backdoor of size $B(F)$. We will present some results from [45] without going into detail on the proof. In particular, we refer the reader to the original publication for a justification of the constants that will appear in the various properties.

Deterministic

In the deterministic algorithm, we perform a brute force search. We first assume that $B(F) \leq n/2$. For all subsets of variables $S \subseteq vars(F)$ of size $|S| = B(F)$ we perform a backtrack search in S until we find an assignment J such that C can solve $F|_J$.

Property 5. *For boolean formulas with a backdoor of size at most $n/4.404$, the deterministic algorithm solves the formula in time $O(c^n)$ with $c < 2$. [45]*

Randomized

In the randomized algorithm, we randomly pick a set of variables that is bigger than $B(F)$ and look whether a backdoor is contained in this set.

For a constant $b > 1$, we repeat $n * (\frac{(n/B(F)-1)}{(b-1)})^{B(F)}$ times (and at least once):

1. Randomly choose a subset S of V s.t. $|S| = b * B(F)$;
2. Perform a standard backtrack search on variables in S,
3. If F is ever solvable by the subsolver C, return the satisfying assignment.

Property 6. *For boolean formulas with a backdoor of size at most $n/2.443$, the randomized algorithm solves the formula in time $O(c^n)$ with $c < 2$. [45]*

The main difference between the deterministic and the randomized algorithm is the size of the smallest backdoor required to obtain an exponential improvement. For example, we obtain $c < 2$ if the smallest backdoor contains less than 20% of the variables

in the deterministic, and less than 40% of the variables in the randomized algorithm. For completeness, we show how the value of c is a function of $k = n/B(F)$:

$$T = max\{1, 2^{n/k*(1+ln(2))/ln(2))} * (ln(2) * (k-1))^{n/k}\}$$
$$c = 2^{log2(x)/n}$$

For example, for $n = 10^4$, $B(F) = 10$ and $k = 10^3$, the randomized algorithm runs with time $O(c^n)$ with $c \approx 1.008$.

Heuristic

The heuristic algorithm tries to take into account the importance of variable selection heuristic in SAT solvers. If we manage to pick all the variables of some backdoor "quickly" we have an incredible improvement over a systematic search. Let (DFS, H, C) be a compact notation for our depth first search algorithm running with heuristic H and subsolver C. We call $1/f(n)$ the probability of H picking a backdoor variable of a particular set.

Property 7. *Given F with a backdoor of size $O(\frac{logn}{logf(n)})$ for which H has success probability $1/f(n)$, (DFS, H, C) has a polynomial time restart strategy that solves F in polynomial time. [45]*

If the success probability is a constant $(1/f(n) = d)$ and we have a $O(logn)$ backdoor we can solve the problem instance with a polynomial time restart strategy. Otherwise, the search runtime is still exponential but decreases quickly with $B(F) = n/k$, since we have runtime proportional to $c^{n/k}$ for some constant c related to the success probability.

Final remarks The following runtime summary table ([45]) helps to understand the importance of these results:

$B(F)$	Deterministic	Randomize	Heuristic
n/k	small $exp(n)$	small $exp(n)$	tiny $exp(n)$
$O(logn)$	$\left(\frac{n}{\sqrt{logn}}\right)^{O(logn)}$	$\left(\frac{n}{logn}\right)^{O(logn)}$	$poly(n)$
$O(1)$	$poly(n)$	$poly(n)$	$poly(n)$

If the backdoor size $B(F)$ is a constant ratio of n, we are still dealing with exponential worst-case complexity, but the base decreases and gets reasonably small if we have a good heuristic; better improvements can be achieved if $B(F)$ is logarithmic w.r.t. n and finally, we deal with a polynomial time problem if the backdoor size is independent from n (constant).

These results do not take explicitly into account the existence of multiple smallest backdoors. While this does not influence the worst case complexity[2] of the deterministic algorithm, it should be taken into account when discussing the randomized and the

[2]Average case complexity would be influenced by this information, but this is not considered.

heuristic algorithm. In the randomized algorithm we take into account the probability of choosing $b * B(F)$ variables out of n that contain a particular backdoor; however, if we have t_i minimal backdoors of size i, then, when picking $b * B(F)$ random variables, we need to take into account that we might pick (at least) *one of* the $\sum_{i=B(F)}^{b*B(F)} t_i$ backdoors. Similarly, for the heuristic algorithm, we define the success probability of picking a variable in a particular backdoor as depending only on the size of the problem instance. If we have several backdoors that share a subset of variables, we should take into account the ability of picking a variable that belongs to *many* backdoor sets. These considerations would make the theoretical framework much more complicated, but it might be interesting to consider them.

3.5. Experimental results

We provide now an overview on some experimental results regarding backdoors. We will focus mainly on methods to detect backdoors and statistics on the size of backdoors for different classes and domains. We conclude this section by citing a few empirical results on the use of backdoors in SAT solvers.

3.5.1. Finding backdoors in practice

In Section 3.4.4 we provided some complexity results for three algorithms that find and use backdoors: deterministic, randomized and heuristic. In practice, to find backdoors, two methods are used: complete and local search. While the complete search method is basically the same as the deterministic algorithm, the local search is usually performed after we already found a backdoor set by other means.

Complete search

A complete search for backdoors can be performed only on small instances, since the runtime is proportional to n^k with n variables and k backdoor size. This approach is used by Li and van Beek ([25]) to search for smallest strong and weak UP+PL+2SAT+Horn+anti-Horn backdoors. By iteratively looking for backdoors of increasing size, they can guarantee to find the smallest backdoors; moreover, minimality is guaranteed by simply avoiding extending a known backdoor. This can be seen as an iterative deepening depth-first search, in which we avoid expanding a node if the path from the root to the node already denotes a backdoor.

We fix an ordering of the variables $V = \{v_1, .., v_n, ..\}$ of the CNF formula F and an integer k. We can use any ordering, for example lexicographical. In order to find all the C-backdoors of F up to size k, we call expand($\{v_i\}$,$\{\}$,k) for all v_i in $var(F)$:

expand(V,S,k):
for Each possible assignment J_v of V **do**
  if $F|_{J_v} \in C$ **then**
    $S = S \cup V$
    return S
  end if
end for
if $k < 1$ **return** False
j = last index of the variables in V
for $i = (j+1) \rightarrow (n-1)$ **do**
  expand($V \cup \{v_i\}$, S, $k-1$)
end for
return False

> The returned set S contains all the (weak) backdoors of F up to size k. As soon as we find a backdoor, we return from the recursive call abandoning that branch of the search tree: e.g. if (x_1, x_2) is a backdoor, we will not test (x_1, x_2, x_i) for any $x_i \in var(F)$.

In the experimental evaluation, Zi and van Beek use this method for tiny instances with ∼1000 variables and smallest backdoor of size 2, and only one instance with ∼18000 variables but with backdoor of size 1.

Local search

To tackle bigger instances (both in the number of variables and size of the backdoors) Kilby et al. ([21]) propose a local search algorithm to find Satz-backdoors, i.e. backdoors for the class of problems solved by the SAT solver Satz without branching (i.e. applying the *split* rule). Their algorithm is later improved by Li and van Beek ([25]) and extended to a wider class of backdoors: UP+PL+2SAT+Horn+anti-Horn.

The idea of the local search strategy is to solve the instance with a SAT solver (e.g. SATz) and record the set W of branching variables. This is not, in general, a minimal SATz-backdoor. Note in fact that the SAT solver might branch on more variables that the ones required to solve the instance.[3] Therefore, the algorithm from Li and van Beek tries both to minimize the set, and to randomly change the variables in W. To guarantee minimality, we simply need to test for each variable $v \in W$ whether $W - \{v\}$ is still a backdoor. The major drawback of this method is that we can only guarantee minimality, but not that our backdoor is a smallest backdoor or that no backdoor of a particular size exists. In order to cover more of the search space they restart the local search with different initial sets W.

The local search strategy is suitable for instances with hundreds of thousands of variables and backdoor size around 0.2%.

Class specific algorithm

The complete and local search algorithms provide a framework to discover backdoors for any subsolver C, and we pay this flexibility with high-running times. As discussed in Section 3.4.3 we can devise efficient (FPT) algorithms that can find strong backdoors (and backdoor trees [38]) for particular classes such as 2SAT, Horn ([29]) and to some extent, RHorn ([32]).

To compute Horn backdoors, Paris et al. ([30]) propose a local search algorithm that first tries to find a renaming for the non-Horn clauses and then (if such renaming does not exist) provides a maximum renaming such that the obtained formula has the biggest

[3] This is exactly the reason why we are interested in backdoors: if we managed to build a SAT solver that only branches on backdoor variables, we would obtain an FPT solving algorithm parameterized by the size of the backdoor.

number of Horn clauses possible; the algorithm then focuses on finding a deletion Horn-backdoor on the remaining non-Horn part of the formula.

Kottler et al. ([23]) describe two algorithms to compute RHorn backdoors based on conflict detection on dependency graphs. In this work they show that it is possible to approximate deletion RHorn-backdoor, with a known approximation ratio.

3.5.2. Backdoors in benchmarks

We started this overview on backdoors by claiming that they are one of the cause of the good performances of SAT solvers. Assuming we are given a backdoor set for our formula, we still need to test at most 2^k assignments for our backdoor set of size k to show satisfiability (for a weak backdoor) or unsatisfiability (for a strong one), therefore our claim makes sense only if k is relatively small.

When looking into experimental results on backdoors size, we need to take into account several informations like class, type, minimality and domain. In particular, the *bigger* the class C is, the smaller we expect the backdoors to be: e.g. UP+PL+2SAT+Horn-backdoors ([25]) are more likely to be smaller than Horn-backdoors.

Unless otherwise stated, in the following we talk mainly about structured problems; structured problems are all these SAT problems that are the result of a reduction from a domain to SAT. For example, whenever we take a problem instance from graph colouring, planning or model checking and generate the equivalent SAT encoding, we are generating a structured problem. The converse of structured problems are random problems in which we generate the CNF formula by randomly assigning literals to clauses.

Dilkina et al. ([10]) provide a nice comparison among different classes of backdoors: Horn, deletion RHorn, Satz, UP+PL and UP. From their experiment it is possible to establish an order based on the range of the size of the backdoor. We indicate the percentage of variables in a backdoor and, since there are many backdoors for the same family of problems, we indicate both the lower and upper-bound on this percentage. This ordering looks like: Horn ($9-67\%$), deletion RHorn ($2-66\%$), UP ($0.15-12\%$), UP+PL ($0.13-1.4\%$) and Satz ($0-0.6\%$). The difference between UP+PL and Satz is due mainly to the look ahead features of Satz. This upper-bound on Satz-backdoors was also shown by Williams et al. ([45]) on a smaller number of instances. The main issue with these results is that Satz implements some preprocessing techniques that make it more powerful than pure DPLL: Gregory et al. ([17]) studies minimal weak DPLL-backdoors for many different domains and instances with results that are slightly higher ($2-5\%$). Recalling the complexity results from Section 3.4.3 we can see how classes with "smaller" backdoors tend to be harder to detect: e.g. Horn is FPT while UP+PL is $\mathsf{W}[P]$-hard.

Clause learning Dilkina et al. ([12]) study the upper-bound size of learning-sensitive backdoors for RSat solver (RSat-backdoors) ([31]) and compare them to weak/strong backdoors for Satz solver (Satz-backdoors). They notice that learning-sensitive backdoors are usually smaller, as the theory suggests (Section 3.2.2). This improvement

can be quite big in some case: e.g. a blocks-world instance (bw_large.d) has a learning-sensitive backdoor that is 40% smaller than the classical backdoor. Similar results are obtained by Gregory et al. ([17]).

Pre-processing Dilkina et al. ([10]) are the only who provide a comparison on the effect of preprocessing on backdoor size. They take 4 preprocessing techniques and try to understand what happens to the upper bound of the size of the Satz and UP+PL backdoors. The results are somehow inconclusive, since sometimes the upper-bound of the backdoors becomes bigger and sometimes smaller.

Backdoor trees Samer and Szeider ([38]) show that strong Horn and deletion RHorn backdoor trees are smaller than strong Horn and deletion RHorn backdoors (where for RHorn we consider only deletion backdoors). This result confirms the theoretical property (Section 3.3.1) that tells us that by considering backdoor trees instead of backdoor sets, we need to test fewer assignments to solve our instance.

Random instances Gregory et al. ([17]) show how weak UP+PL-backdoors are usually bigger in random instances (∼10%); the main difference between their result and the result from Kilby et al. ([21]) is that the latter uses Satz *with* preprocessing as subsolver: whenever the preprocessor is sufficient to solve the instance, Kilby et al. count the backdoor as being of size 0. Therefore, a direct comparison is not possible.

In general, we consider that empirical results on backdoors are not easily available. We identify two reasons for this. The first is the computational cost of obtaining these results, the second is the wide range of possible changes in the methodology that would affect the final result: the choice of the class, the subsolver and the type of search.

3.5.3. Exploiting backdoors

From the theory, we expect a SAT solver that only branches on backdoor variables to perform incredibly well. This experiment has been conducted by Paris et al. ([30]) by computing strong Horn-backdoors for some instances and comparing the running time of zChaff with the running time of zChaff forced to branch only on the backdoor variables. The results match our expectations, in particular even for instances that do not have a particularly small backdoor set, zChaff+backdoors perform usually orders of magnitude better. Unfortunately, we are missing more generic results that take into account different classes of backdoors and different domains.

Finally, no result is provided on using ad-hoc methods for solving problems with a known polynomial time algorithm. Since the classical definition of backdoors requires the subsolver C to run in polynomial time, we could try to use specific subsolvers (e.g. 2SAT, Horn, RHorn) and compare the runtime for the same instance by comparing different classes of backdoors.

3.6. Chapter summary

This chapter presented a comprehensive overview of backdoors for SAT. We started by presenting the classical definition of strong, weak and deletion backdoor (Section 3.1) and continued with several extensions of this idea (Section 3.2). Later on we presented some complexity results; there (Section 3.4) we focused mainly on the parameterized complexity results. These results tell us that finding backdoors efficiently might be possible for some classes, like Horn and 2SAT. Finally, we provided an overview on the experimental results related to backdoor detection and usage (Section 3.5).

This chapter shows how vast the area of backdoors has grown. The idea of backdoors has been used also in other contexts such as quantified boolean formulas ([37]) and answer set programming ([15]); however we will not cover these topics here, since our focus is on SAT.

Among all the ideas introduced in this chapter, we will take deletion Horn-backdoor detection and explore this problem in the parameterized complexity framework in the next chapter.

4. Horn-Backdoors and Vertex Cover

In order to study backdoors, we need to be able to compute them in a reasonable amount of time. This would allow us to generate big datasets on which to make and verify hypothesis. As described in the previous chapter, backdoor detection is a NP-hard problem and, therefore, we have little hope of finding an efficient way to solve the generic problem. Therefore, we need to focus on classes for which backdoor detection is easier, for example in FPT, like Horn. In this chapter we study the strong Horn-backdoor detection problem via reduction to Vertex Cover. By doing so, we are able to use existing results to solve our problem. In Section 4.1, we present the original proof by Nishimura et al. ([29]) of strong Horn-backdoor detection being FPT. Afterwards, we present the reduction to Vertex Cover introduced by Samer and Szeider ([38]) in Section 4.2. Before starting experimenting with our new problem set, we introduce a local search algorithm, a parameterized algorithm and a reduction to SAT to solve the Vertex Cover problem (Section 4.3). Finally, we perform some experiments to try to understand how hard it is to compute strong Horn-backdoors for SAT instances taken from different benchmarks, and whether we can use local search to solve our problem (Section 4.4). We conclude the section by presenting a few examples in which we relate the size of the backdoor with some other properties of our instances.

4.1. Strong Horn-backdoor detection is FPT

Nishimura et al. ([29]) show that strong Horn-backdoor detection and strong 2SAT-backdoor detection are fixed-parameter tractable. While being conceptually simple, the proof was the first to perform a parameterized analysis on the backdoor detection problem. This approach was then applied to several other classes with both positive (FPT) and negative (non-FPT) results (Section 3.4.3). In this section we present the original proof by Nishimura et al., apply it to an example and discuss some limitations of this approach.

4.1.1. The algorithm

Recall that a clause is Horn iff it has at most one positive literal. A CNF formula is Horn iff all its clauses are Horn. Finally, recall that the class Horn is clause-induced and therefore every deletion backdoor is also a strong backdoor (Section 3.2.1).

The algorithm by Nishimura et al. tries to find a strong Horn-backdoor by finding a deletion Horn-backdoor. The main idea is quite straight-forward: given a non-Horn clause, how many literals do we need to *delete* to make it a Horn clause? To answer this question, we can simply consider each non-Horn clause and select all but one of its positive literals. The set of literals obtained this way is a deletion Horn-backdoor, although it is probably not minimal. But now it should be clear that we can perform a search by branching on the decision of which literals to take for each clause.

Example

Consider the non-Horn formula

$$F = (x \vee y \vee z) \wedge (\neg x \vee z \vee w) \wedge (y \vee w)$$

to make it Horn we need to remove all but one positive literal from each clause. We could start by x:

$$F - \{x\} = (y \vee z) \wedge (z \vee w) \wedge (y \vee w)$$

but we still have three non-Horn clauses. Therefore we need to remove another literal:

$$F - \{x, y\} = (z) \wedge (z \vee w) \wedge (w)$$

while the first clause is now Horn, the second is not. We decide to remove also z:

$$F - \{x, y, z\} = (w) \wedge (w)$$

we obtain a deletion/strong Horn-backdoor for F, namely $B_1 = \{x, y, z\}$. It should be clear that our initial choice of picking x wasn't the best choice. We can safely remove x from B_1 and obtain $B_2 = \{y, z\}$ that is a minimal deletion Horn-backdoor for F:

$$F - \{y, z\} = (x) \wedge (\neg x \vee w) \wedge (w).$$

As discussed in Section 2.2.3, whenever we have a search tree, we can easily obtain an FPT algorithm by bounding the depth of the tree. In this case, we fix a value k to be the size of our backdoor set, and perform our search by bounding the number of literals in the solution to be less than k. By doing so, we obtain a bounded search and, by iteratively increasing the value of k, we are guaranteed to find a globally minimal solution (i.e. a smallest backdoor in this case).

In order to show that the problem is indeed FPT we still need to show that the branching factor of our search doesn't depend on the size of the problem (n) but is either a constant or a function of k.

Assume we have a non-Horn clause C. By definition, C has at least two positive literals p_1 and p_2. In order for C to become Horn, we remove either p_1 or p_2 to obtain C'. If C' is still non-Horn, we can apply the same reasoning recursively until we obtain a Horn clause. By removing one positive literal at a time, we are guaranteed to obtain,

eventually, a Horn clause. This idea provides us with a search algorithm with two possible branches. Therefore, by bounding the depth of this search algorithm, we obtain an FPT algorithm.

Algorithm 1 FPT algorithm for strong Horn-backdoor detection

sb-horn(F,k):
  if $F \in Horn$ **return** $\emptyset$
  if $k = 0$ **return** False
  C = getNonHornClause(F)
  (p_1, p_2) = getPositiveLiterals(F)
  B = sb-horn($F - p_1$,k-1)
  if $B \neq False$ **return** $B \cup \{p_1\}$
  B = sb-horn($F - p_2$,k-1)
  if $B \neq False$ **return** $B \cup \{p_2\}$
  return False

Theorem 3. *[29] Strong Horn-backdoor detection can be solved in* $O(2^k * n)$*, with* k *size of the backdoor set and* n *size of the formula. Therefore, strong Horn-backdoor detection is* FPT.

Example

We can try to apply Algorithm 1 to the formula F to find a backdoor of size 2:

$$F = (x \vee y \vee z) \wedge (\neg x \vee z \vee w) \wedge (y \vee w)$$

```
sb-horn(F,2)
-- C = {x,y,z}
-- p_1 = x, p_2 = y
-- sb-horn(F-{x},1)
-- -- C = {y,z}
-- -- p_1 = y, p_2 = z
-- -- sb-horn(F-{x,y},0)
-- -- -- F-{x,y} not Horn
-- -- -- k==0 return false
-- -- sb-horn(F-{x,z},0)
-- -- -- F-{x,z} not Horn
-- -- -- k==0 return false
-- -- return false
-- sb-horn(F-{y},1)
-- -- C = {x,z}
-- -- p_1 = x, p_2 = z
-- -- sb-horn(F-{y,x},0)
```

```
-- -- -- F-{x,y} not Horn
-- -- -- k==0 return false
-- -- sb-horn(F-{y,z},0)
-- -- -- F-{y,z} is Horn
-- -- -- return {}
-- -- return {z}
-- return {z,y}
{z,y}
```

4.1.2. Considerations on the algorithm

One of the nice side effects of showing FPT membership of a particular problem, by means of bounded search or kernelization, is that we are provided with a concrete algorithm to solve it. In this particular case the proof provides us with a bounded search algorithm that we can use. Therefore, we started playing with Algorithm 1 to see for what values of the parameter we could solve our instances in a reasonable time. Our implementation, with few optimizations to better handle backtracking, allowed us to solve instances with solution size (i.e. parameter k) around 30 within 1 hour.[1].

While the complexity of this algorithm is much better than the complexity of a trivial enumeration algorithm ($O(n^k)$), we are probably interested in dealing with bigger values of k.

Another drawback of this approach is that we don't have a kernelization technique for it. A kernelization is useful for two main reasons: reduction of the size of the instance and fast rejection. The first allows us to ignore the size of the instance by simply considering it bounded by a function of the parameter. The second allows us to reject some unsatisfiable instances in polynomial time.

We could have tried to come up with optimizations for our implementation, pre-processing techniques and kernelization algorithms for strong Horn-backdoor detection, however, the scope of this work would have been limited to this new and particular problem. Therefore, we decided to move our focus on a more generic, used and better studied FPT problem: Vertex Cover. By doing so, we hope to provide results that have a bigger impact on the parameterized complexity field.

[1] As for all our experimental results, the time refers to an Intel Centrino 1.7Ghz with 1GB Ram. Note that the runtime of this algorithm, for unsatisfiable instances, grows exponentially with the value of the parameter k

4.2. Reduction to vertex cover

In Samer and Szeider ([38]), the authors propose a reduction from strong Horn-backdoor detection to Vertex Cover. This reduction goes as follows:

Definition 46 (G_F)**.** *G_F is the graph of variables of the CNF formula F in which two variables v, u are adjacent iff v and u appear positively in a clause from F. More formally, $G_F = (V, E)$ with $V = var(F)$ and $E = \{\{u, v\} \mid \exists C \in F.u, v \in C\}$.*

Once we build this graph we simply compute a vertex cover for it. The solution obtained is related to a strong Horn-backdoor:

Lemma 8. *A set $B \subseteq var(F)$ is a strong Horn-backdoor for F iff B is a vertex cover of G_F*

The proof of Lemma 8 was omitted from the original paper; therefore, we provide here a possible proof:

Proof. To prove this result we need to prove a simple property of the graph G_F.

For a graph $G = (V, E)$ and a set of vertices $S \subseteq V$, we use $G - S$ to indicate the graph obtained from G by removing all vertices in S and all edges that have at least one vertex in S.

We can see that $G_F - B = G_{F-B}$, in fact:

- They contain the same set of vertices $V \setminus B$ (by definition of G_F and $G_F - B$).

- They contain the same set of edges. To prove this, we first see that $G_F - \emptyset = G_{F-\emptyset}$. Now we see that there is an edge $e = (v, w)$ in G_F iff v and w appear positively in a clause in F. If $v \in B$, then $G_F - B$ doesn't contain the edge e, since there is no vertex v; also G_{F-B} doesn't contain the edge e anymore, because there is no variable v anymore.

B is a deletion Horn-backdoor for F iff $F - B$ is Horn. $F - B$ is Horn iff there is no clause with two positive literals. But this is the case iff G_{F-B} doesn't have edges. That is equivalent to saying that $G_F - B$ doesn't have edges; but $G_F - B$ doesn't have edges iff B is a vertex cover for G_F. Therefore, B is a deletion Horn-backdoor for F iff B is a vertex cover for G_F. □

Example

Let's consider the formula F and build the graph G_F:

$$F = (x \vee y \vee z) \wedge (\neg x \vee z \vee w) \wedge (y \vee w)$$

We can see the relation between the vertex cover $\{y, z\}$ and the deletion Horn-backdoor $B = \{y, z\}$.

Minimum vertex cover is the optimization version of the vertex cover problem in which we want to minimize the size of the cover. Solving minimum vertex cover would allow us to find a smallest strong Horn-backdoor.

The existence of this reduction allows us to apply existing theory and algorithms from Vertex Cover to strong Horn-backdoor detection. In the next section we will explore three solutions that we applied in order to study strong Horn-backdoor detection.

4.3. Vertex Cover implementations

Since vertex cover is an NP-complete problem, it is an interesting challenge to try to solve it efficiently and avoid the worst case exponential runtime. In order to do so, there has been a lot of work on both complete ([3]) and local search ([34]) algorithm. In this work we decided to apply three different techniques for solving it: local search, FPT algorithms and reduction to SAT.

4.3.1. Local search: COVER

COVER ([34]) is a local search algorithm for k-vertex cover. The search is performed by generating candidate solutions by exchanging two vertices u and v, where u is currently in the cover, while v is not. The initial solution is generated by a greedy algorithm that simply adds vertices with the highest degree[2] in the cover, until a set of size k is generated. Afterwards, the local search is performed until a cover is obtained or a maximum number of iterations (MAX_ITERATIONS) is performed. This version of the algorithm works when the value of k is known. Otherwise, we can simply perform an iterative search and run COVER with decreasing values of k. Recall that, because it is a local search algorithm, we cannot guarantee that we will find the optimal solution.

Richter et al. developed the local search algorithm COVER ([34]) and performed a comparison between local search and parameterized algorithms. In fact, they show how COVER performs better than the parallel parameterized vertex cover of Abu-Khzam et al. ([3]). What they show is that COVER can be orders of magnitude faster than Abu-Khzam's algorithm, while maintaining a high solution quality. The solution quality is an indication of the distance between the optimal solution and the solution obtained by the algorithm. They use a triple a-b-c, where a is the number of runs in which the optimal k^* was found; b is the number of runs in which a solution of size $k^* + 1$ was found but none of size k^*; c is the number of runs in which only solutions of size $k^* + 2$ or worse were found. In their experiments on the biological dataset used by Abu-Khzam, they found out that only for one instance the solution quality was 98-1-1, while in all the other cases it was 100-0-0.

This result motivated us to formulate the following hypothesis:

- It is possible to use COVER to solve strong Horn-backdoor detection, while maintaining a good solution quality.

We decided to use this algorithm for two reasons. First, the authors had already compared this algorithm with parameterized algorithm and published promising results. Second, they provided the source code of the implementation of COVER, and therefore it was simple for us to use it and modify it to fit our needs.

[2] Recall that the degree of a vertex is simply the number of its neighbours.

Figure 4.1.: Rule P3

4.3.2. FPT algorithm

Vertex cover is a well studied problem in the FPT community. While there are many publications discussing several algorithms, unfortunately there is no available implementation of any of them. Therefore, we had to implement our own algorithms. We decided to find a compromise between complexity of the implementation and worst case complexity of the algorithm. Therefore, we decided to implement[3] a kernelization algorithm called *High degree kernelization* ([2]) and a bounded search algorithm by Hüffner ([19]).

High degree kernelization

The high degree kernelization is based on a set of preprocessing rules and one additional kernelization rule. Given a graph $G = (V, E)$ and a parameter k, we define a rule as a transformation of the problem instance (G, k) to a new instance (G', k') where $|V'| < |V|$ or $k' < k$ (or both). Additionally, whenever we reduce k' we keep track of a partial solution C.

Preprocessing For preprocessing we use the following four rules:

P1 A vertex of degree 0 cannot be part of any cover, therefore we obtain G' by removing it from G.

P2 If there is a vertex u of degree 1, then there is an optimal vertex cover in which its neighbour v is in the cover. Therefore, we remove both u and v from G, reduce k by one and add the vertex v to C (Figure 4.3).

P3 If there is a vertex of degree 2 with two adjacent neighbours, then there is an optimal vertex cover containing both these neighbours. Let's say v has degree two and u and w are its neighbours. If u and w are adjacent, then there is an edge (u, w) that must be covered. Since u and w might have degree bigger than two, we cannot guarantee an optimal cover if we take u and v or w and v. Therefore, we remove the three vertices from G, reduce k by two and add to C both u and w (Figure 4.1).

[3]Our implementation is available at `http://marco.gario.org/work/master`

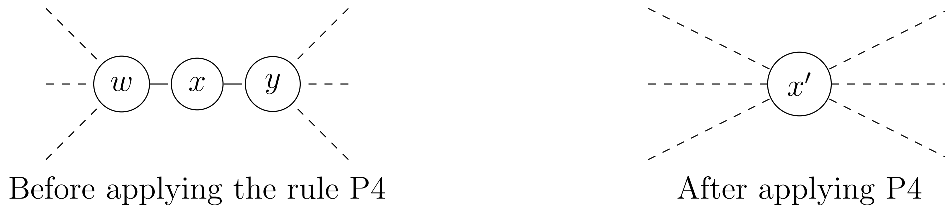


Figure 4.2.: Rule P4

P4 If there is a vertex u of degree 2 with two non-adjacent neighbours v and w, then u can be removed by contracting the edges (v, u) and (u, w). We obtain G' by replacing u with u' and connecting to it all the neighbours of v and w, and removing u,v and w from G. k is reduced by one but we cannot say anything on C. In fact, after applying this rule we cannot take the result C' from the problem instance (G', k') as solution, but we need to reconstruct the solution. This goes as follows: if u' is in the cover C', it means that both v and w are in the cover of G. Note that in this case the size of the cover doesn't change, since we *remove* u' and add v and w instead. Similarly, if u' is *not* in C' then u is in C (Figure 4.2).

We apply each of these rules as many times as possible. This preprocessing can be carried out in $O(n^2)$, with n the number of vertices.

Kernelization The high degree kernelization is very simple but, nevertheless, quite powerful:

HdK A vertex of degree $> k$ must be in any cover of size $\leq k$. In fact, if v is a vertex of degree $> k$ and it is not included in the cover, then we need to add all its neighbours to the cover; but this is impossible because we are looking for a cover of size $< k$.

We apply this rule repeatedly and interleave it with the preprocessing rules. By doing so we can apply the following result:

Theorem 4. *If i) G is a graph with a vertex cover of size k and ii) there is no vertex of G with degree $> k$ or degree < 3, then $|V| \leq \frac{k^2}{3} + k$. [2]*

It should be clear that the preprocessing rules R1-4 guarantee that there is no vertex of degree less than three; moreover, the HdK rule guarantees that there is no vertex of degree bigger than k.

We provide here the proof of Theorem 4 in order to provide a better understanding of this result.

Proof. Let's assume we have a cover C for G of size k (first assumption of the theorem). We can construct the complementary set $\overline{C}$ containing all the $|V| - k$ vertices that are not in the cover. All the vertices in $\overline{C}$ have degree > 3 (because of the second assumption of the theorem). Therefore each element of $\overline{C}$ has at least three neighbours in C, in fact,

Figure 4.3.: Rules P2 and S1

if there was a neighbour not in C then there would be an uncovered edge and C wouldn't be a vertex cover. Let F be the set of all edges with one vertex in $\overline{C}$. From the previous argument, we can say that there must be at least $3(|V| - k)$ edges in F. Since each element in G has at most k neighbours, we also know that each element in the cover C can have at most k neighbours. Therefore, $|C| * k = k^2$ is an upper-bound on the size of F. We summarise this result with the inequality $3(|V| - k) \leq |F| \leq k^2$. This can be easily transformed in $3(|V| - k) \leq k^2$ and, therefore, $|V| \leq \frac{k^2}{3} + k$. □

In this theorem lies the power of the kernelization techniques. We have two benefits from applying this technique:

- V is bounded in size by the parameter only. This allows us to perform a simpler analysis on the behaviour of this algorithms. We are not interested in the exact value of $|V|$ since we know that it is bounded by our choice of the parameter;
- The counter positive of the theorem states that, whenever $|V| > \frac{k^2}{3} + k$ then G does not have a vertex cover of size k. This allows us to perform early rejection of instances without performing search at all.

$O(1.47^k)$ bounded search

Hüffner presents ([19]) a quite simple bounded search algorithm for vertex cover. There are more efficient algorithms to solve the problem, but they usually are based on more sophisticated branching rules. We decided that this algorithm, having only three branching rules and a worst-case complexity of $O(1.47^k)$, was a good compromise between simplicity and worst case complexity. These are the rules:

S1 If there is a vertex of degree one, put its neighbour into the cover. This rule is exactly the same as P2 (Figure 4.3).

S2 If there is a vertex v of degree two, then either i) both neighbours of v are in an optimal vertex cover, or ii) v is in an optimal cover together with all neighbours of its neighbours (Figure 4.4).

S3 If there is a vertex v of degree at least three, then either v or all its neighbours are in the cover (Figure 4.5).

After applying these rules, we obtain a search tree in which the leaf nodes are either solutions or unsatisfiable instances $(G', 0)$ where G' has uncovered edges. It should be

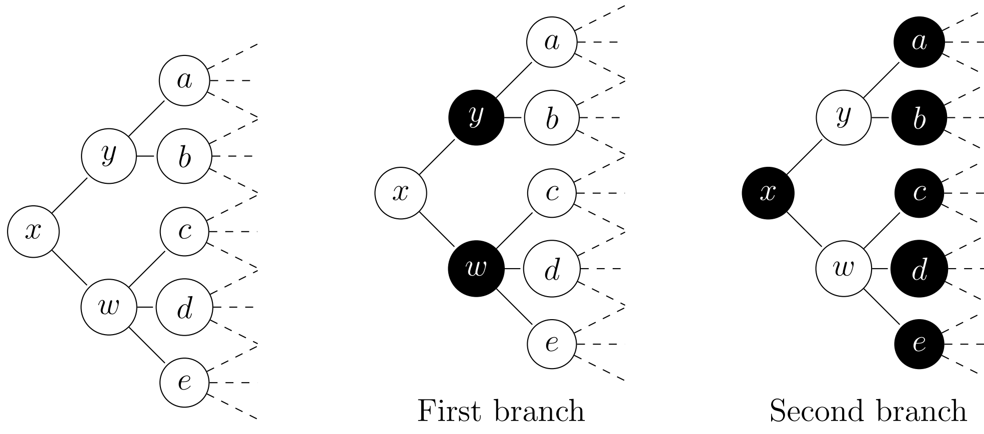


Figure 4.4.: Rule S2

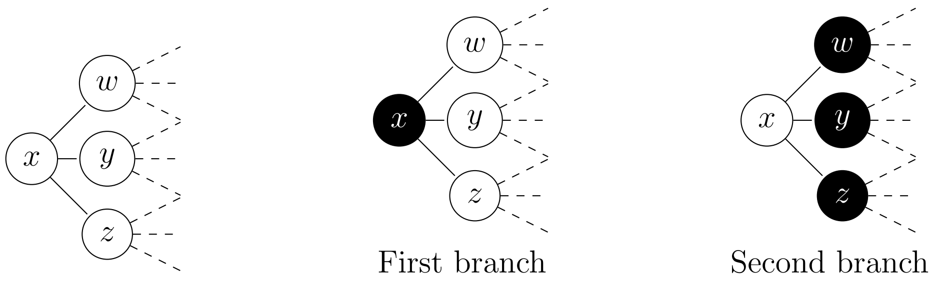


Figure 4.5.: Rule S3

also clear that the rules S2 and S3 influence the parameter k differently. For example, the first branch of S2 has $k' = k - 2$ while the second branch has $k' < k - 5$, since the neighbours of v must have degree at least 2. This means that we can avoid expanding a branch if, in that branch, k' would become smaller than zero. Taking from the previous example, if we have to apply S2 and $k = 4$, then we can avoid exploring the second branch since $k' < 4 - 5$ would be negative.

Considerations on the implementation

We tried to keep our implementation simple and extensible, but we think that there might be interesting possibilities to explore in the future:

Integrating rule S2 and P3. Rule P3 tells us how to handle vertices of degree 2 that have adjacent neighbours. This would allow us to avoid branching in rule S2 when the vertex meets this condition. We would still need rule S2 to handle the cases in which the neighbour nodes are not adjacent, but we might avoid a whole subtree.

Heuristics on branching. In our implementation we explore the branches in the order explained here; however, there might be a heuristic suggesting which branch to prefer at each step.

Interleaving with kernelization. During our search, we constantly modify the graph. Therefore, it might be the case that we can detect unsatisfiability by kernelization. Interleaving kernelization and search might improve our runtime. However, we decided not to implement this technique because it complicates the backtracking of the algorithm.

We keep the graph in memory as an adjacency list, but use an additional structure to perform backtracking more quickly. A support graph (SG) is a structure containing a vector with the count of the degree of each vertex in the graph and the information of whether the vertex was *removed*. Additionally, the SG keeps a priority queue with a list of nodes of degree up to a given value N. This structure requires only $O(|V|)$ space to be stored, but allows us to perform removal and reinsertion of a node in linear time. Moreover, since the bounded search algorithms for vertex cover relay on the degree of the vertices (as shown by rules S1-S3), by having a priority queue, we can pick a branching node in constant time.[4]

4.3.3. Reduction to SAT

We considered also a third alternative to compute vertex cover: a reduction to SAT. In particular, the hard task during our experiment (see next section) was to show that the vertex cover was optimal. In fact, the local search algorithm cannot give us this

[4] There is a trade-off between the time required to find a branching node and the overhead of keeping the priority queue. While our initial tests confirm that the priority queue provides a slightly better performance, a more detailed study would be valuable.

guarantee, and the FPT algorithm needs exponential time to show unsatisfiability for a given size k of the vertex cover. We decided to try to see whether a SAT solver would help. Not only SAT solvers are highly optimized, but they also perform more elaborate reasoning than our FPT algorithm. In particular, clause learning should reduce the search space of our SAT instance.

The encoding of vertex cover into SAT is quite straightforward. First, each edge of the graph is represented as a binary clause; second, we add a constraint stating that at-most k variables can be set to true. Let's consider the graph $G = (V, E)$, we associate each vertex v with a variable in our formula, e.g. $v \in var(F)$. Therefore, for each edge $(v_i, v_j) \in E$ we need to state that at least one among v_i and v_j must be true (i.e. in the cover):

$$\bigwedge_{(v_i,v_j)\in E} (v_i \vee v_j).$$

Additionally, we state that at most k vertices can be in a cover of size k. Now the real issue is how to encode the *at most* constraint. We used the encoding suggested by Sinz ([40]) that allows us to encode the constraint by using $O(k*n)$ clauses and $O(k*n)$ new variables.[5] The idea of this encoding is to have a series of circuits performing addition. At each stage the sum is increased by the value of the variable v_i. If the sum overflows, the variable x_i is set to true. Since the overflow occurs only if there are more then k variables v_i that are true, we can use these overflow variables to detect the violation of the at-most constraint.

[5]Robert Stelzmann kindly provided us with an implementation of this algorithm

4.4. Experimental results

In our experimental study we try to understand how hard it is to solve strong Horn-backdoor detection on a benchmark of SAT problems. As discussed in Section 3.5, most of the studies in this area try to understand how big backdoors are. Since they usually (e.g. [30],[25]) apply local search algorithm to answer this question, it is impossible to guarantee the minimality of the solution. Moreover, we are not aware of studies that discuss the time required to extract this information from the instances.

4.4.1. Goals and methodology

In our experiments we try, for each instance, to:

- find the smallest strong Horn-backdoor,
- test the quality of the solution obtained by the local search algorithm,
- present the correlation between smallest strong Horn-backdoor size and other features of the instance.

In order to do so, we proceeded as follows:

1. We composed a benchmark with 3239 instances from various sources.
2. We generated the associated vertex cover instances.
3. We ran a modified version of COVER to obtain an upper-bound on the size of the backdoor.
4. For instances with small backdoors (up to $k = 150$) we verified the minimality of the backdoor.
5. For instances with bigger backdoors we verified a lower-bound bigger than 150.
6. Considering only the instances for which we have the exact value of the smallest backdoor, we compute the quality of the solution provided by a fast version of COVER.

A complete list of the instances with lower- and upper-bound is available online.[6]

Configuration A big part of the experiment was conducted on the cluster of the Technische Universität Dresden. For all the experiments in which we were interested in timing, the results refer to an Intel Centrino 1,7Ghz with 1GB RAM.

The first time we ran COVER, we allowed the iterative version to improve up to a 10% from the original solution, performing 10^5 iterations at each step. For example, if the first solution found by COVER had size 500, we would let COVER run for all

[6] The complete dataset is available for download from `http://marco.gario.org/work/master`

values of k up to 450. Instead, when we were interested in the quality of the solution, we decided to run a faster version of the algorithm. We would let COVER run until it failed to find a solution for a given k, and reduced the number of iterations at each step to 10^4. The reason for doing so, is that we are interested in evaluating the quality of the solution of COVER in a situation in which we allow it to run for few seconds.

To verify the lower-bounds on our solutions, we used both our own FPT algorithms and CryptoMinisat 2.9 ([41]) on the SAT reduction. Before giving up the search, we let these solvers run with a time-out of 90 minutes for each instance.

Statistical information In the following sections we will present some statistical measures about our benchmark. We use the following derived values in our discussion: minimum, maximum, average, standard deviation and correlation coefficient. Figure 4.6 describes the formulas used, where N is the total number of elements in our data series, x_i is the i-th element and $\overline{x}$ is the average of the series.

$$\text{Average: } \overline{x} = \frac{1}{N}\sum_{i=0}^{N} x_i$$

$$\text{Standard deviation: } s_x = \sqrt{\sigma^2} = \sqrt{\sum_{i=0}^{N} \frac{(\overline{x} - x_i)^2}{N-1}}$$

$$\text{Correlation: } r = \frac{\sum_{i=0}^{N} x_i y_i - \overline{x}\,\overline{y}}{(N-1)s_x s_y}$$

Figure 4.6.: Formulas used for statistical analysis

4.4.2. Benchmark statistics

Before presenting the results obtained in our experiments, we are going to provide some information about the benchmark we used.

We took 3239 instances from SATLIB, SAT'11 competition and Car Configuration ([18],[1] and [24]). Following the methodology of the SAT competition, we can classify these instances as: 1097 Random, 604 Industrial and 1518 Crafted. 20 instances were not classified in any of these categories. Note that the Industrial group is mainly composed by the Car Configuration instances (588 out of 604). These instances were added to the original dataset because it was lacking industrial instances.

Figure 4.7 provides an overview on some statistical information about the instances in our benchmark divided by type. The value *avgClause size* is already an aggregated value over the single instances. For each instance we compute the average clause size, and then we aggregate these values. This summary tells us the number of variables per type, and shows us that most of the instances have an average clause size close to 3. The

distribution of the size of the instances as number of variables and number of clauses is depicted in Figure 4.8 and 4.9. Note that the clause count distribution is represented in logarithmic base. We can see that many instances have roughly 1000 clauses, and few instances have more than 10^4 clauses. Moreover, most of the instances have less than 300 variables, with roughly 600 instances in the range 1500-2000.

From this overview, we can see that the dataset we worked on is composed by instances that are relatively small.

	Variable Count			Clause Count			avgClause Size			
	min	max	$\overline{x}$	min	max	$\overline{x}$	min	max	$\overline{x}$	s_x
Industrial	170	9523	1734,61	588	34238	6720,79	1,41	44,44	3,52	2,28
Random	20	1024	114,15	76	36000	1026,22	2,98	10	3,25	1,03
Crafted	12	6384	197,83	32	2162160	3225,64	2,09	12	2,44	1,11
Other	48	20490	6056	261	123329	36262,75	2,21	2,77	2,50	0,22
ALL	12	20490	492,24	32	2162160	3336,49	1,4	44,43	2,91	3

Figure 4.7.: Benchmark Statistics

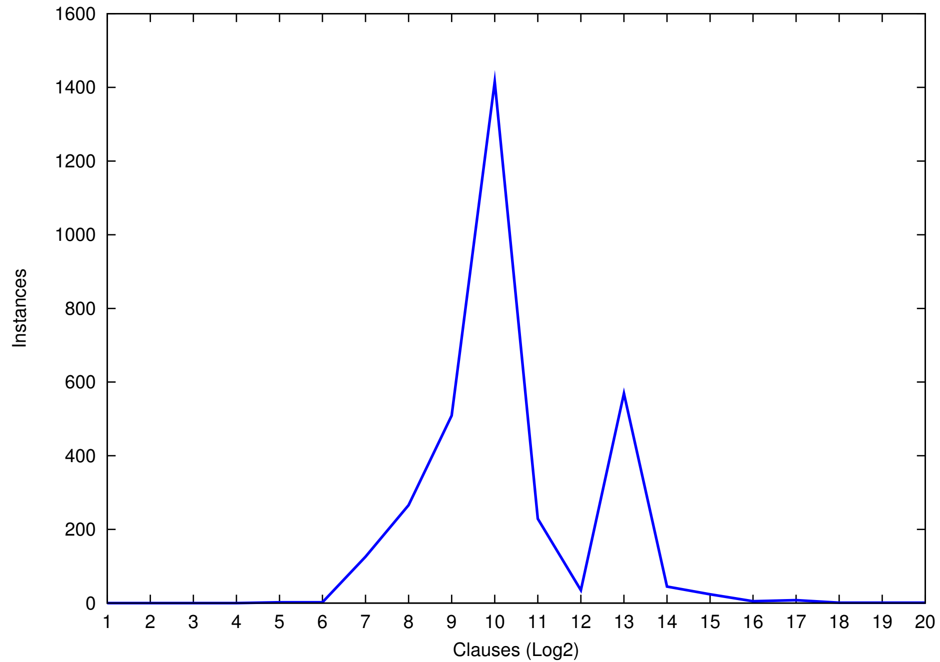


Figure 4.8.: Clause count distribution

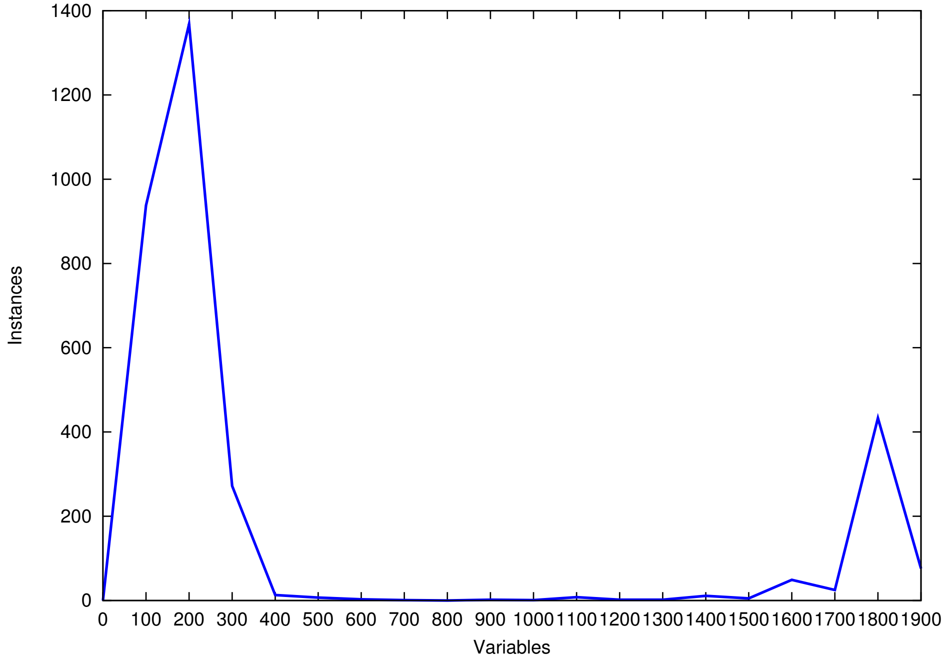


Figure 4.9.: Variable count distribution

4.4.3. Upper-bounds

We computed the upper-bound on the smallest strong Horn-backdoor variables with COVER with the configuration explained in Section 4.4.1. For brevity, we call this value mvc-ub, since it is the upperbound on the minimum vertex cover. Figure 4.10 shows the distribution of this value among the instances. We can see that there are two spikes, but roughly a third of the instances have mvc-ub between 90 and 105. Moreover, 2417 instances have mvc-ub less or equal 150. These are the instances for which we verified the lower-bound. The other concentration point (360-375) includes 332 instances, mainly from the Car Configuration domain.

4.4.4. Lower-bounds

We divided the computation of the lower-bound in three phases. First, we tried to compute the lower-bound for all instances whose upper-bound was up to 150. We tried to compute the lower-bound using the FPT algorithms. In the few cases where this approach failed, we generated the corresponding SAT problem and tried to solve it instead. We gave up on solving, and considered the run failed, after 150 minutes. To compute the lower-bound we took the upper-bound k of each instance and tried to solve it for $k-1$. If k is a minimal value, then $k-1$ will give us an unsatisfiable instance. By applying the kernelization and the bounded search, we were able to verify the lower-bound for most of our instances.

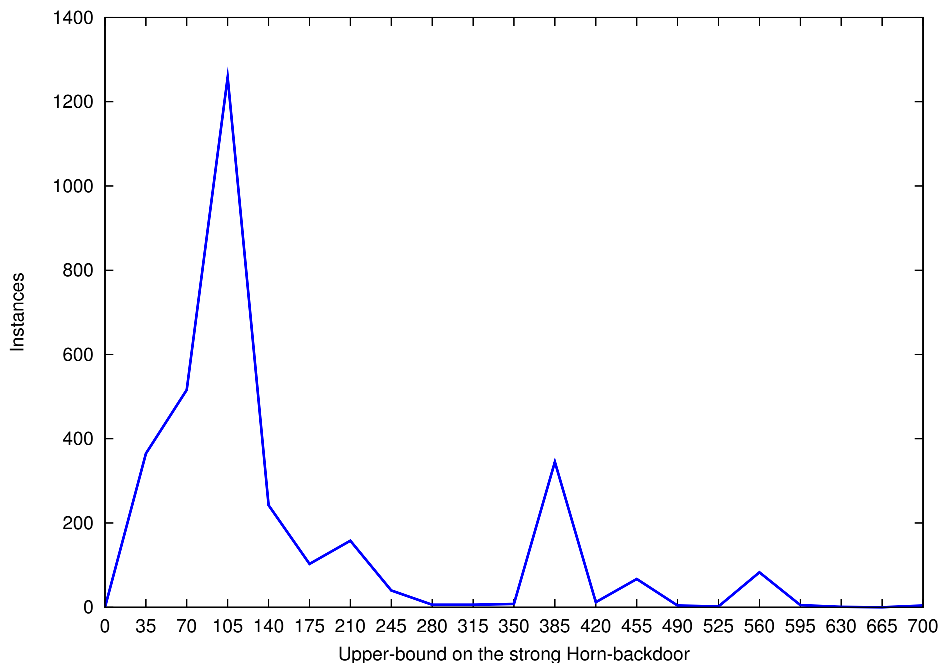


Figure 4.10.: Distribution of the upper-bound on the strong Horn-backdoor

Finally, we applied the FPT algorithms to all the other instances to see which ones had a solution of size less than 150 but were not detected by the local search algorithm.

Up to 150 Out of 2418 instances that had a strong Horn-backdoor of size up to 150, we solved 2357 (97.5%) with the FPT algorithms. On the 61 unsolved instances, we performed the SAT reduction and tried to solve them with CryptoMinisat. By doing so we solved 8 instances more, but 53 remain unsolved: we couldn't prove their lower-bound. It is interesting to note that CryptoMinisat performed a quite different search w.r.t. our FPT algorithm. Figure 4.11 shows the runtime of CryptoMinisat on these 8 instances. We can see that 7 instances were solved quite quickly, and only one took slightly more than 30 minutes.

There is a huge gap in runtime between solved and unsolved instances. With 90 minutes time-out for the FPT algorithm, we were surprised to find out that the average solving time was 25 seconds. 87% (2122) of the instances were solved in less than 5 seconds, 93% in under a minute. This means that the generated vertex cover instances were in most of the cases easy, but the hard ones were really hard: i.e. the 2% that we couldn't solve at all.

Kernelization played an important role in achieving a good runtime. In fact, 51.5% of the instances were solved by kernelization. Recall that a kernelization technique can perform early rejection in polynomial time (Section 2.2.3). After applying kernelization, we might have a smaller parameter for our new instance. Since we are working on unsatisfiable instances, a smaller parameter can provide an exponential improvement

Instance	k	Runtime (seconds)
sgen1-unsat-85-100.cnf	64	1852.16
sgen1-unsat-97-100.cnf	73	67.59
ais8.cnf	98	15.44
hole10.cnf	99	11.95
C171_FR.cnf	129	133.60
3blocks.cnf	138	56.21
C250_FV.cnf	139	69.90
C250_FW.cnf	139	71.44

Figure 4.11.: CryptoMinisat runtime for "hard" unsatisfiable vertex cover instances

on our runtime. It is interesting, therefore, to see how the parameter changes. On the instances that were not solved by kernelization (1172), we got a parameter reduction of 17.8% on average. If we break down this average, we can see that roughly 20% of the instances were not reduced, and 5% were reduced by more than 50%. For example, `aim-200-2_0-yes1-4.cnf` was reduced from $k = 109$ to $k' = 6$.

Finally, we note that on the "hard" instances that we didn't manage to solve, the average parameter reduction was of 18%. Therefore, the hardness of these instances is not given by the size of the parameter.

150-up There are 821 instances with an upper-bound on the backdoor size between 151 and 5307. Of these instances, 121 (14.7%) were solved by kernelization.

For the remaining instances, we tried to compute a lower-bound by means of the FPT algorithms or reduction to SAT. We managed to verify that 602 instances had a lower bound of at least 151. For the remaining instances we couldn't compute the lower-bound of 151 neither with the FPT algorithms nor with Cryptominisat within a 90 minutes timeout. By simply applying kernelization we obtained a lower-bound on the solution for all these instances, but unfortunately it was too small to be meaningful.

4.4.5. Local search quality

We focus in this section on studying the results obtained by the local search algorithm on the instances for which we know the exact value of the smallest strong Horn-backdoor (2486 instances). Most of the instances have a solution of size (k) less than 150. However, we also managed to prove some bigger lower-bounds for a few instances by kernelization.

Once we know the exact size of the smallest strong Horn-backdoor, we can run the local search algorithm and verify whether the solution we obtain is optimal or not.

In the first run of COVER, we used the configuration described in Section 4.4.1. Performing this computation takes between 30 and 90 minute but we obtain the optimal solution for all our instances with only one run (solution quality 1-0-0). This result is interesting since the runtime of the local search algorithm depends only polynomially

on the size of the instance and parameter. However, the configuration we used performs a quite extensive and expensive search. We are interested in trying to reduce the computation time of COVER. We define a new configuration that combines small iterations (MAX_ITERATIONS=10^4) with a quick fail (i.e. the search is concluded if we cannot find a solution for a given k within MAX_ITERATIONS). We run COVER with this new configuration 100 times and we measure run-time and solution quality.

For 2449 instances we obtained a solution quality of 100-0-0. For the remaining 37 instances the solution quality is detailed in Figure 4.13. Overall, we can see that the algorithm performs well and almost always provides a solution that is at most 1 value from the optimum; recall that the solution quality *a*-*b*-*c* indicates how many times we obtain the optimum (a), one more than the optimum (b) or 2 or more than the optimum (c). Note also that the last four instances in Figure 4.13 (`par32` family) have a good solution quality even if the solution size (i.e. the size of the cover) is around 1200.

The runtime of this configuration of COVER is on average of 115 ms, with a standard deviation of 387 ms. If we focus only on instances with solution size up-to 150, we obtain an average runtime of 97 ms and a standard deviation of 35 ms. Figure 4.12 shows the runtime distribution of this configuration of cover.

The average error among all the instances is 0.11%. This value, together with the solution quality and the runtime results suggest that the use of a local search algorithm like COVER is a good method to compute strong Horn-backdoors.

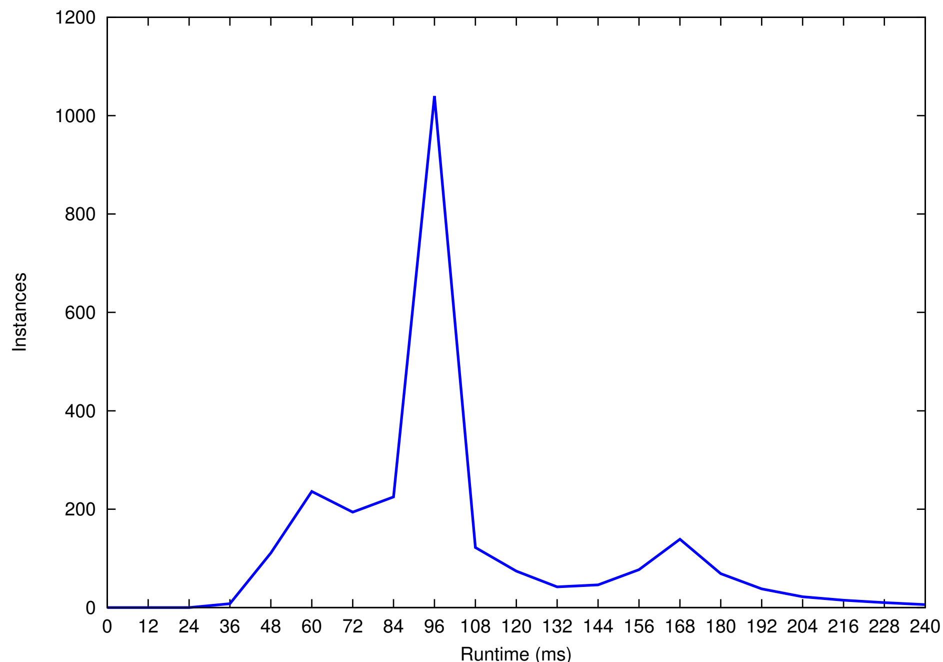


Figure 4.12.: COVER's runtime ($k \leq 150$)

Instance	Solution quality	Instance	Solution quality
uuf225-029.cnf	35-68-0	uf200-033.cnf	99-1-0
uuf125-032.cnf	83-17-0	uf200-046.cnf	99-1-0
uuf175-030.cnf	88-12-0	uf200-08.cnf	99-1-0
x1_48.shuffled-as.sat03-1592.cnf	96-3-1	uuf150-02.cnf	99-1-0
uuf175-046.cnf	96-4-0	uf200-019.cnf	99-1-0
uuf250-023.cnf	96-4-0	uf200-030.cnf	99-1-0
uf200-025.cnf	97-3-0	uf200-031.cnf	99-1-0
aim-200-3_4-yes1-2.cnf	98-1-1	uf200-035.cnf	99-1-0
uuf175-036.cnf	98-1-1	uf200-036.cnf	99-1-0
uuf250-021.cnf	98-2-0	uf200-045.cnf	99-1-0
uuf200-032.cnf	98-2-0	uf200-050.cnf	99-1-0
uf200-032.cnf	99-1-0	uf200-06.cnf	99-1-0
uf200-048.cnf	99-1-0	uf200-07.cnf	99-1-0
uf200-05.cnf	99-1-0	uuf150-043.cnf	99-1-0
uf200-047.cnf	99-1-0	uuf175-011.cnf	99-1-0
uf200-04.cnf	99-1-0	uuf200-027.cnf	99-1-0
uf200-029.cnf	99-1-0		
par32-2.cnf	99-1-0 (1232)	par32-3.cnf	99-1-0 (1258)
par32-5.cnf	99-1-0 (1257)	par32-4.cnf	97-3-0 (1269)

Figure 4.13.: Solution quality

4.4.6. Strong Horn-backdoor and features

In instances belonging to the same family, it happens that the upper-bound on the smallest strong Horn-backdoor is almost constant or it is related to some simple feature of the formula.

Flat Noteworthy is the `flat` family of instances. This family of instances encodes the problem of graph colouring on flat graphs ([8]). This is an artificial class of graphs designed to be hard to be solved by some algorithms. The interesting result we obtained here is that the upper-bound on the size of the strong Horn-backdoor (mvc-ub) is always twice the number of vertices in the original graph. For example, the family `flat-30` is obtained by encoding into SAT the problem of colouring flat graphs with 30 vertices. This family, i.e. *all instances*, has mvc-ub 60; similar results are obtained for `flat-50` and `flat-100`. Another characteristic of this family is to have 3 variables for each vertex. Therefore, there is an excellent correlation between the number of variables and the size of the mvc-ub of these instances. This motivated us to verify whether other families had the same behaviour.

Random instances The family of uniform random formulas (uf/uuf) shows a strong correlation between the number of variables and the size of the smallest strong Horn-backdoor. The correlation coefficient is 0.99 and this can be seen easily from Figure

4.14. This behaviour seems to be common among all random instances. We obtain a correlation coefficient of 0.81 for all others random instances, excluding the (u)uf family and the hardnm instances (for which we don't have a lower-bound on the backdoor size).

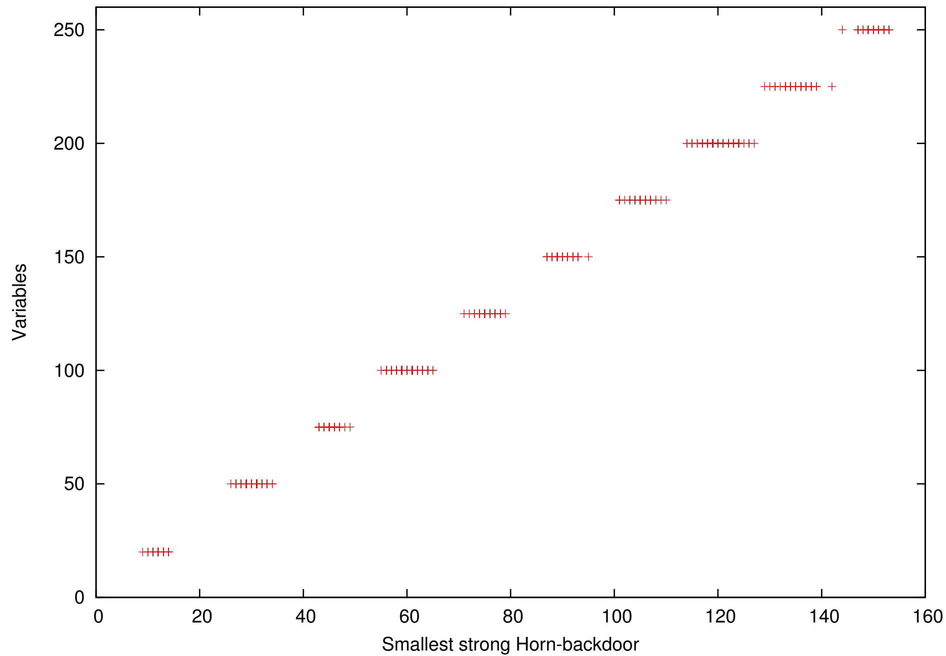


Figure 4.14.: Correlation between Variables and smallest strong Horn-backdoor in the (u)uf family

Car Configuration The results for the Car Configuration industrial family refer only to the *upper-bound* on the size of the smallest backdoor. In fact, for this family, the average smallest backdoor size is 379. This family is composed by instances that are named to indicate a configuration (expressed as a number) and a test. If we consider all the instances together, we obtain a correlation of 0.65 between number of variables and the upper-bound on the size of the strong Horn-backdoor. However, if we consider all the instances related to the *same* configuration, we obtain much higher correlations. Figure 4.15 presents a summary table of the correlation coefficients together with the number of instances for each configuration.[7]

Concluding remarks

The results presented in this section are just examples of the advantage of having a dataset with information concerning backdoors. Moreover, we believe that backdoors are strongly related to the domain of application. These 3 examples are meant to motivate further study of backdoors in relation to features of the instances. This could

[7]We consider only configurations with more than 5 instances when computing the correlation

Base configuration	Correlation (r)	# Instances
C168	0.99	58
C170	0.99	6
C202	0.83	23
C208	0.99	16
C210	0.89	32
C220	0.95	348
C638	0.73	84

Figure 4.15.: Correlation coefficients for the Car Configuration family

allow, eventually, to build models to predict backdoor size for a given instance. These models would then be particularly useful to decide whether to solve the instance by exploiting its backdoors.

What we couldn't consider in this work is the relation between the single backdoor sets. It might be the case that similar instances share similar backdoor variables, and therefore it would be possible to devise search heuristics to find these variables.

4.5. Chapter summary

In this chapter we explored the connection between strong Horn-backdoor detection and Vertex Cover. We presented the original proof of deletion/strong Horn-backdoor detection being FPT (Section 4.1), and the reduction to Vertex Cover (Section 4.2). Later on, we presented three methods to solve Vertex Cover: local search, FPT algorithms and reduction to SAT (Section 4.3). Armed with these tools we built a dataset of SAT instances with additional information on their smallest strong Horn-backdoor. After presenting the methodology and challenges faced in building the dataset (Section 4.4), we use it to show how local search algorithms can be used to obtain good solutions really fast. We concluded the chapter by presenting three examples of studies linking backdoor size to properties of the instance; these examples are meant to motivate further research in the creation of good quality datasets of SAT instances.

Our goal for the experimental part was to study the relation between strong Horn-backdoor detection and vertex cover. We think that this study shows two important results:

- Kernelization is fundamental to be able to solve many instances. In fact, even if we implemented a really simple kernelization, we manage to solve many instances without search.
- Local search, and in particular COVER, allows us to solve quickly and with an excellent quality strong Horn-backdoor detection

5. Conclusion

In this work we explored the field of backdoors for SAT. While preparing the overview on the field, we faced many interesting questions. However, we had to focus on a smaller set of objectives in order to complete this master thesis. We present now some ideas for future work that build upon the results presented here.

5.1. Future directions for backdoors

The idea of backdoor was created to try to explain the good behaviour of SAT solvers on some instances. Little work has been done on trying to exploit backdoors to improve solvers. While it is not obvious how to apply the backdoor information to DPLL SAT solvers, we think that backdoors should be used to explore new solving techniques. For example, we should try to implement efficient Horn solvers and use the backdoor detection as a pre-processing step in order to generate formulas that are solvable in polynomial time. The vast majority of research on complete solvers has been done on DPLL/CDCL algorithms. Therefore, the first step would be to reconsider the experiment performed by Paris et al. ([30]) and see whether restricting a CDCL solver on branching only on strong Horn-backdoors can provide an improvement in solving runtime *even* if we include the time spent in finding such backdoors. More in general, the study of backdoors might open new opportunities to study alternative solving approaches. In this direction the classical definition of backdoor allows us to work only with polynomial time subsolvers. Studying more in detail classes of backdoors that are NP might still be valuable. The work on pseudo backdoors only scratches the surface of this problem. More work on particular classes and complexity of the relative detection problem would be valuable.

It would also be quite interesting to understand the influence of various SAT pre-processing techniques on the backdoor size for different classes. For example, pre-processing techniques that remove clauses might induce a growth of the size of the backdoors for all the classes that are not closed under clause removal. Since SAT solvers relay heavily on preprocessing, the interaction between these techniques and backdoors might provide interesting results.

Finally, we presented some simple results on the correlation between features of an instance and the size of the smallest strong Horn-backdoor. Continuing in this direction, one could try to define models to predict the size of this backdoors and, eventually, heuristics to be able to detect variables belonging to a particular backdoor set. The interaction between machine learning and backdoors is an unexplored area that might provide interesting results. We hope to have contributed in this direction by presenting

methods to efficiently build datasets of strong Horn-backdoors.

5.2. Future directions for parameterized complexity

Most of the experimental work of this thesis was focused on applying results from the parameterized complexity community. Our contribution in this sense is a set of publicly available tools to solve Vertex Cover. We were surprised by the lack of public implementations of many important FPT algorithms, and we believe that making implementations available is a good way of attracting new researches in the field.

Our implementation of vertex cover is quite simple. There are many more *theoretically* better algorithms that we could have implemented. Since we devised our implementation in a flexible and extensible way, it would be interesting to implement other algorithms and then compare their *empirical* performances: e.g. crown reduction ([2]), Chen's vertex cover ([6]) and Abu-Khzam's ([3]) parallel vertex cover.

While studying different algorithms to solve vertex cover, we found many similarities with SAT solving. However, an interesting feature missing in all the vertex cover algorithms is *learning*. Seeing how learning drastically improved SAT solving, we are left wondering whether it would be possible to apply a similar idea in order to solve vertex cover.

Finally, we would like to stress out that this work was focused only on Horn-backdoors and vertex cover. However, as we show in Section 3.4.3 there are many other classes for which backdoor detection is FPT (e.g. RHorn and Cluster) and we are not aware of any work trying to solve backdoor detection for these classes using FPT algorithms.

5.3. Summary of this work

In this work we explored the field of backdoors for SAT. We started by providing an overview of the state of the art on backdoors in Chapter 3. We presented theoretical definitions, complexity analysis and some experimental results. We also provided an introduction to parameterized complexity, and we tried to show how important this field is in the study of backdoors. In particular, we investigated the relation between strong Horn-backdoor detection and Vertex Cover by trying to understand what are the possible approaches to solve these problems. At the end of Chapter 4 we provided positive results suggesting that kernelization and local search algorithms are good ways to solve strong Horn-backdoor detection. We also provided a few examples on how to use a dataset containing information about backdoors to study the relation between the domain of the instance and its backdoors. Finally, we concluded this chapter with some idea on future work.

Bibliography

[1] *SAT'11 Competition.* http://www.satcompetition.org/, 2011.

[2] F. N. Abu-Khzam, R. Collins, M. Fellows, M. Langston, W. Suters, and C. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 62–69.

[3] F. N. Abu-Khzam, M. a. Langston, P. Shanbhag, and C. T. Symons. Scalable Parallel Algorithms for FPT Problems. *Algorithmica*, 45(3):269–284, Apr. 2006.

[4] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating Satisfiable Problem Instances. *Proceedings of the national conference on artificial intelligence*, 2000.

[5] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability.* IOS Press, 2009.

[6] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, Sept. 2010.

[7] Y. Crama, O. Ekin, and P. Hammer. Variable and term removal from Boolean formulae. *Discrete Applied Mathematics*, 75(3):217–230, 1997.

[8] J. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, 26:245, 1996.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] B. Dilkina, C. Gomes, and A. Sabharwal. Tradeoffs in backdoors: Inconsistency detection, dynamic simplification, and preprocessing. In *ISAIM-08: 10th International Symposium on Artificial Intelligence and Mathematics*, 2007.

[11] B. Dilkina, C. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. *Principles and Practice of Constraint Programming–CP 2007*, pages 256–270, 2007.

[12] B. Dilkina, C. Gomes, and A. Sabharwal. Backdoors in the Context of Learning. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 73–79, 2009.

[13] R. G. Downey. Parameterized complexity for the skeptic. *18th IEEE Annual Conference on Computational Complexity, 2003. Proceedings.*, pages 147–168, 2003.

[14] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.

[15] J. Fichte and S. Szeider. Backdoors to tractable answer-set programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 863–868. AAAI Press/IJCAI, 2011.

[16] J. Flum and M. Grohe. *Parameterized complexity theory.* Springer-Verlag, 2006.

[17] P. Gregory, M. Fox, and D. Long. A new empirical study of weak backdoors. In *Principles and Practice of Constraint Programming*, pages 618–623. Springer, 2008.

[18] H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In I. Gent, H. Maaren, and T. Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000.

[19] F. Hüffner, R. Niedermeier, and S. Wernicke. Techniques for Practical Fixed-Parameter Algorithms. *The Computer Journal*, 51(1):7–25, Mar. 2007.

[20] S. Inform. On 2-SAT and Renamable Horn. *Annals of Mathematics*, 1996.

[21] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1368, 2005.

[22] H. Kleine Buning and Z. Xishun. Satisfiable formulas closed under replacement. *Electronic Notes in Discrete Mathematics*, 9:48–58, 2001.

[23] S. Kottler, M. Kaufmann, and C. Sinz. A New Bound for an NP-Hard Subclass of 3-SAT Using Backdoors. In *Proceedings of the 11th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2008)*, pages 161–167, 2008.

[24] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1):145–163, 2000.

[25] Z. Li and P. van Beek. Finding Small Backdoors in SAT Instances. *Proceedings of the 24th Canadian Conference on Artificial Intelligence*, pages 269–280, 2011.

[26] C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001*, pages 279–285.

[27] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535. IEEE, 2001.

[28] N. Nishimura and P. Ragde. Solving #SAT using vertex covers. *Acta Informatica*, 44(7):509–523, 2007.

[29] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[30] L. Paris, R. Ostrowski, P. Siegel, and L. Sais. Computing Horn Strong Backdoor Sets Thanks to Local Search. *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 139–143, Nov. 2006.

[31] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. *Solver description, SAT competition*, 54:9–10, 2007.

[32] I. Razgon and B. O'Sullivan. Almost 2-SAT is fixed-parameter tractable. *Journal of Computer and System Sciences*, 75(8):435–450, 2009.

[33] B. Reed. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, July 2004.

[34] S. Richter, M. Helmert, and C. Gretton. A stochastic local search approach to vertex cover. *KI 2007: Advances in Artificial Intelligence*, pages 412–426, 2007.

[35] R. Rossi, S. Prestwich, S. Tarim, and B. Hnich. Generalizing Backdoors. In *Proceedings of the 5th International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS 2008)*, number 03, 2008.

[36] Y. Ruan, H. Kautz, and E. Horvitz. The backdoor key: A path to understanding problem hardness. *AAAI'04 Proceedings of the 19th national conference on Artifical intelligence*, pages 118–123, 2004.

[37] M. Samer and S. Szeider. Backdoor sets of quantified Boolean formulas. In *Proceedings of the 10th international conference on Theory and applications of satisfiability testing*, pages 230–243. Springer-Verlag, 2007.

[38] M. Samer and S. Szeider. Backdoor trees. *Proceedings of the 23rd Conference on Artificial*, pages 363–368, 2008.

[39] J. Silva and K. Sakallah. GRASP—A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997.

[40] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. *Theory and Practice of Constraint Programming-CP 2005*, pages 1–5, 2005.

[41] M. Soos. CryptoMiniSat 2.5. 0. *SAT Race competitive event booklet*, 2010.

[42] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, Dec. 2004.

[43] S. Szeider. Backdoor sets for DLL subsolvers. *SAT 2005*, pages 73–88, 2006.

[44] S. Szeider. Matched formulas and backdoor sets. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 94–99, 2007.

[45] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceeding of IJCAI-03*, volume 18, pages 1173–1178, 2003.

A. Instances with unverified lower-bound below 150

Instance name	Upper-bound
289-sat-11x4.cnf	132
289-sat-4x8.cnf	96
289-sat-5x8.cnf	120
289-sat-6x8.cnf	144
289-sat-7x6.cnf	126
2bitmax_6.cnf	134
aim-200-6_0-yes1-2.cnf	139
aim-200-6_0-yes1-3.cnf	137
aim-200-6_0-yes1-4.cnf	136
counting-easier-php-012-010.sat05-1172.reshuffled-07.cnf	98
dubois50.cnf	100
eulcbip-7-UNSAT.shuffled-as.sat05-3936.cnf	101
eulcbip-8-UNSAT.sat05-3937.reshuffled-07.cnf	121
genurq6Sat.shuffled-as.sat03-1512.cnf	113
genurq7Sat.shuffled-as.sat03-1513.cnf	136
marg6x6.shuffled-as.sat03-1456.cnf	120
mod2c-3cage-unsat-10-3.sat05-2568.reshuffled-07.cnf	105
mod2c-rand3bip-unsat-120-3.sat05-2340.reshuffled-07.cnf	120
mod2-rand3bip-sat-210-2.sat05-2159.reshuffled-07.cnf	145
pmg-11-UNSAT.sat05-3939.reshuffled-07.cnf	112
pmg-12-UNSAT.sat05-3940.reshuffled-07.cnf	127
pret150_25.cnf	100
pret150_40.cnf	100
pret150_60.cnf	100
pret150_75.cnf	100
s113-100.cnf	85
sgen1-sat-140-100.cnf	112
sgen1-sat-160-100.cnf	128
sgen1-sat-180-100.cnf	144
sgen1-unsat-103-100.cnf	79
sgen1-unsat-109-100.cnf	82
sgen1-unsat-115-100.cnf	88
sgen1-unsat-121-100.cnf	91
sgen1-unsat-127-100.cnf	97
sgen1-unsat-133-100.cnf	100
sgen1-unsat-139-100.cnf	106

Instance name	Upper-bound
sgen1-unsat-145-100.cnf	109
sgen1-unsat-151-100.cnf	115
unsat-set-b-fclqcolor-10-07-09.sat05-1282.reshuffled-07.cnf	141
urqh1c5x5.shuffled-as.sat03-1468.cnf	104
urqh5x5.shuffled-as.sat03-1481.cnf	128
urquhart3_25.shuffled-as.sat03-1553.cnf	102
urquhart4_25bis.shuffled-as.sat03-1554.cnf	128
Urquhart-s5-b3.shuffled-as.sat03-1572.cnf	96
x1.1_56.shuffled-as.sat03-1584.cnf	111
x1.1_64.shuffled-as.sat03-1585.cnf	126
x1.1_72.shuffled-as.sat03-1586.cnf	143
x1_56.shuffled-as.sat03-1593.cnf	110
x1_64.shuffled-as.sat03-1594.cnf	126
x1_72.shuffled-as.sat03-1595.cnf	142
x2_56.shuffled-as.sat03-1602.cnf	111
x2_64.shuffled-as.sat03-1603.cnf	126
x2_72.shuffled-as.sat03-1604.cnf	142