FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Parameter Setting in SAT Solver Using Machine Learning Techniques
Student:	Bc. Filip Beskyd
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of winter semester 2021/22

Instructions

CDCL SAT solvers often use many numeric parameters such as restart period, amount of randomness in heuristics, etc. that have a significant impact on the performance. The task in the thesis is to set these parameters specifically according to the given input formula instead of using default values aiming at maximizing the performance. It is assumed that various attributes can be derived from the formula that can be used to predict the solver performance for a particular set of parameters. This thesis will focus on employing machine learning techniques to automate the process of parameter setting. The student will fulfill the following tasks:
1. Study the design of CDCL SAT solvers and their parameters.
2. Suggest attributes that can be derived from the formula and have a significant impact on the performance.
3. Design and implement parameter settings based on selected machine learning techniques using the attributes.
4. Perform experimental evaluation on a relevant set of benchmarks.

References

[1] Gilles Audemard, Laurent Simon: On the Glucose SAT Solver. International Journal on Artificial Intelligence Tools 27(1): 1840001:1-1840001:25 (2018)

[2] Andre Biedenkapp, Marius Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, Holger H. Hoos: Efficient Parameter Importance Analysis via Ablation with Surrogates. AAAI 2017: 773-779

[3] James Bergstra, Daniel Yamins, David D. Cox: Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. ICML (1) 2013: 115-123

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 16, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Parameter Setting in SAT Solver Using Machine Learning Techniques

Bc. Filip Beskyd

Department of Applied Mathematics
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

July 30, 2020

Acknowledgements

In this place, I want to thank my supervisor doc. RNDr. Pavel Surynek, Ph.D. for providing guidance, consultations, his time and valuable advices throughout whole time I was writing this thesis. I also want to thank my family for continuous support during my studies. Thanks to the Metacentrum.cz [1] for providing the computational resources which I extensively used to produce this thesis.

Declaration

Citation of this thesis

Beskyd, Filip. *Parameter Setting in SAT Solver Using Machine Learning Techniques*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

SAT řešiče jsou nezbytné nástroje pro mnohé oblasti počítačové vědy a průmyslu. Obsadily funkci univerzálního nástroje, který uživatelé používají k řešení problémů, jež by v opačném případě museli řešit ad-hoc, což by pravděpodobně nebylo zdaleka tak efektivní jako moderní SAT řešiče.

V posledních dvou a více dekádách spojených s výzkumem SAT řešičů bylo vytvořeno mnoho heuristik. Ty nejefektivnější z nich jsou dnes neodmyslitelnou součástí moderních SAT řešičů, což dále zlepšuje jejich efektivitu v porovnání s jejich předchůdci. Heuristiky mohou být, před samotným provedením prohledávacího procesu konkrétní SAT instance, laděny jedním nebo více numerickými parametry.

V této diplomové práci představuji nástroj, který za pomoci technik strojového učení předpovídá hodnoty těchto parametrů pro heuristiku z podkladové struktury SAT instance s cílem redukce výpočetního času.

Klíčová slova SAT problém, splnitelnost, řešič, grafová sktuktura, strojové učení, heuristika, ladění parametru

Abstract

SAT solvers are essential tools for many domains in computer science and engineering. SAT solvers took a place of a universal tool which their users use when in need for solution of their problems, which would otherwise require ad-hoc solution, which would probably be nowhere near the effectiveness of modern SAT solvers.

Over the course of at least two decades of SAT related research, many heuristics were produced, most effective ones are embedded in SAT solvers of present day, which further increase their effectiveness compared to their predecessors. Heuristics can usually be tuned by single or multiple numerical parameters prior to executing the search process over the concrete SAT instance.

In this thesis I present machine learning approach which predicts the parameter values for heuristic from underlying SAT instance structure in view of reducing computational time.

Keywords SAT problem, satisfiability, solver, graph structure, machine learning, heuristic parameter tuning

Contents

List of Figures

Introduction

SAT problem is one of the fundamental computer science problems. Concretely SAT problem was the first one to be proven to belong to NP-Complete class of problems [3]. Its major use-cases come from industries such as software testing [4], automated planning [5], hardware verification [6] or cryptography [7], as well as many other, more of a theoretical character problems of computer science can be reduced to a SAT problem.

Standard and in practice most used way of solving given problem is to convert it, in some way, to concrete SAT instance which is then given to another program as an input, so called SAT Solver, which solves the instance and answers whether there exists an truth assignment by which it can be satisfied or not, with the concrete proof, that is either variables assignment which satisfy the formula or conflict.

There exist many solvers to SAT problem. Solvers are divided into two major groups, online and offline solvers. This thesis is focused on one offline solver, based on Conflict-driven clause-learning (CDCL) algorithm the Minisat [8]. CDCL SAT solvers have witnessed dramatic improvements in their efficiency over the last 20 years, and consequently have become drivers of progress in many areas of computer science such as formal verification [9]. There is general agreement that these solvers somehow exploit structure inherent in industrial instances. Typically, solvers have many parameters which need to be set prior to solver being executed to find the solution. Depending on how various parameters are set, will be mirrored in running time duration of the solver, thus naturally it makes sense to try to apply machine learning to predict values for these parameters which would hopefully reduce solving time, based on type of given concrete instance. This is logical because previous research shows that many instances are in some way similar thus formula's hidden structure will be similar too. For example structure of an instance which comes from hardware verification industry is vastly different from the structure of the instance which was constructed for pigeon-hole problem [10]. Problems which belong to the same class of problems tend to have similar structure within the class and concrete values of parameters works better for them, than having pre-determined parameter values globally for every instance.

Goals

First of the goals of this thesis is to show in an experimental way that there exist a dependency of SAT solver's parameters on solving time of the instance, and thus it makes great sense to try to tune the parameters to minimize solving time.

Second goal is to explore various input instance features and experiment with automated setting of selected SAT solver parameters to speed up solver's solving time, by extracting features from the instance. These features should be then used in machine learning technique which would be used to learn function of the features and hopefully predict parameter settings which would yield faster solving time than the solver's default parameter values.

Third and last goal is to evaluate how predicted parameters perform on both training set and testing set which is made of new instances which the learning technique did not encounter during its learning.

Contribution

1. Clear visual summary of dependencies of parameters on solving time in form of plots.

2. Extension of usual features extracted by SAT solvers from instances, by computing graph related features additionally on clause graph (CG), and variable–clause graph (VCG), which better capture the underlying structure of the instance.

3. Experimental, not stand–alone python program (Jupyter notebooks [11]) able to extract features, train model and predict parameters.

Structure of the thesis

In chapter 1, I define necessary terminology from logic and graph theory. Chapter 2 focuses on explaining SAT related terminology, principles, and pseudo-codes of 2 main SAT solvers. Chapter 3 explains which heuristics are used in underlying solver, and parameters of MiniSat solver which can tune these heuristics. In chapter 4, I provide brief overview of 3 other works, which are similar to what I am doing in this thesis. Chapters 5, 6 and 7 are core of this thesis, in chapter 5, I experimentally show that there are dependencies of parameters on solving time for selected classes of SAT instances, and therefore show why it is worth a try to use machine learning technique to auto-tune MiniSat's parameters. Chapter 6 shows how I derive features from graph representation of SAT instance, which features I use for machine learning and provide pseudo-codes of my implementation. Final chapter 7 present how predicted parameters compare to MiniSat's default parameters in terms of solving time (number of conflicts).

Chapter **1**

Theoretical background

Purpose of this chapter is to make reader acquaint with basics necessary to understand what I will be doing in this thesis. Thus this chapter is a place where I will define relevant terms from logic and satisfiability theory as well as CNF.

1.1 Basic Boolean logic terminology

This section will provide exact definitions of terms from logic theory that I will be using further in the text.

1.1.1 Boolean variable

Definition 1.1.1. *Boolean variable* is a variable that can only take two possible values, those are `true` and `false` (numerically respectively 1 and 0).

1.1.2 Literal

Definition 1.1.2. A *literal* is either a variable (x) or its negation ($\neg x$), also interpreted as a pair of variable (x) and *sign* (negative meaning $\neg$ and positive meaning absence of $\neg$). Sign is also called *polarity* of a variable in some literature and papers.

Boolean variables and literals are usually represented as small letters of English alphabet such as x, y or z.

1.1.3 Boolean operator

Definition 1.1.3. Basic *Boolean operators* are `and` – conjunction ($\wedge$) , `or` – disjunction ($\vee$) and `not` – negation ($\neg$).

There are more Boolean operators, but to understand the idea of SAT problem they are not necessary here. Operators are also called *connectives* in other literature.

1.1.4 Boolean formula

The language of Boolean formula consists of Boolean variables, Boolean operators and parentheses.

Definition 1.1.4. *Boolean formula* is formed by connecting Boolean variables by Boolean operators, these interconnections are enclosed in parentheses into logical sentences.

Definition 1.1.5. Boolean variable alone is also a Boolean formula.

Definition 1.1.6. A *clause* is a disjunction ($\vee$) of literals.

An example of a Boolean formula: $\phi = ((x \vee \neg z) \wedge (\neg y \wedge z))$

1.1.5 Rules for logical sentence creation

Definition 1.1.7. Let A and B be Boolean formulas. Then $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, are Boolean formulas as well.

Finite application of these rules allow to create complicated formulas.

Note: This definition enforces that every sentence constructed by Boolean operators must be enclosed in parentheses. To improve readability, we can omit most of the parentheses for small cases, if we keep an order of precedence. The order of precedence in propositional logic is (from highest to lowest): $\neg$, $\wedge$, $\vee$.

1.1.6 Partial and complete assignment

Definition 1.1.8. Partial truth assignment for formula ϕ assigns a truth value (`true` or `false`) to some of the variables of ϕ. Special case of partial truth assignment is complete truth assignment for formula ϕ, assigns a truth value to every variable of ϕ. Both are functions from variables to truth values: $V : var(\phi) \rightarrow \{\texttt{true}, \texttt{false}\}$.

An example of an assignment: $v = \{x \mapsto \texttt{true}, y \mapsto \texttt{false}, z \mapsto \texttt{true}\}$

Definition 1.1.9. A clause is *falsified* with respect to an assignment, if all the literals in the clause evaluate to `false`.

1.2 Graph theory

This section contains definitions of graph types used later in the parameter tuning chapter.

1.2.1 Graph

Definition 1.2.1. *Graph* G is a pair or 2 sets V and E, $G = (V, E)$. V is a set of vetrices/nodes, E is a set of unordered pairs of vertices from V, called *edges*, $E \subseteq \{(x, y)|x, y \in V^2\}$.

1.2.2 Bipartite graph

Definition 1.2.2. Graph $G = (\{X \cup Y\}, E)$ is *bipartite* when its set of vertices V, can be partitioned into two disjoint sets X, Y called *partity* such that no two vertices in X are connected with an edge, and no two vertices in Y are connected with an edge.

1.3 Boolean Satisfiability Problem

Definition 1.3.1. The *Boolean Satisfiability Problem*, in practice often abbreviated as "*SAT problem*" or simply "*SAT*", is the problem of finding assignment to the variables of the Boolean formula such that the formula evaluates to `true`.

If this is true, the formula is said to be *satisfiable*. On the other hand, if no such assignment exists, the function expressed by the formula is `false` for all possible variable assignments and the formula is called *unsatisfiable*.

For example, the formula $(A \wedge (\neg B))$ is *satisfiable* because one can find an assignment A = `true` and B = `false`, which yields $(A \wedge (\neg B))$ = `true`. In contrast, $(A \wedge (\neg A))$ is obviously contradictory and whether we let A = `true` or A = `false`, formula can not evaluate to `true` and thus is *unsatisfiable*.

1.4 Conjunctive normal form

While the grammar of propositional logic is suited for manual formula manipulation, a simpler structure is advantageous for algorithmic manipulation. The most common choice for SAT solving is the *conjunctive normal form* (CNF).

The usefulness of the conjunctive normal form for SAT solving was first highlighted by Davis and Putnam in [12]. A formula in conjunctive normal form consists of a conjunction of clauses. This allows a simple representation as a set of sets of literals.

Definition 1.4.1. A *conjunctive normal form* (CNF) of a formula is a conjunction ($\wedge$) of clauses.

Any propositional logic formula can be transformed to CNF. Simplest is to convert each subformula of the original formula to CNF by first replacing all implications and equivalences, then apply De Morgan laws, skolemize, etc., but this naive recursive approach has an exponential growth in space of the formula. There are many efficient ways to do this.

Common way to obtain CNF of the formula is to use Tseitin transformation [13]. For satisfiability testing its not necessary to obtain equal formula, but rather *equisatisfiable* formula. The Tseitin transformation is a linear transformation from an arbitrary propositional formula to CNF, preserving satisfiability.

Definition 1.4.2. Two formulas ϕ and ψ are *equisatisfiable* if and only if ϕ is satisfiable when ψ is satisfiable.

The idea of Tseitin transformation is to introduce new variable for every sub-formula, except for literals, and use these new variables as fresh variables in the formula. Thus the resulting formula is not only on the original variables, but also on new variables. Then conjunct all of the new variables and transform each of them by the logic laws to CNF.

1.4.1 Tseitin transformation example

An example of this transformation can be illustrated on this small formula: Let $\phi = ((x \lor y) \land z) \Rightarrow (\neg w)$.

Sub formulas are:
$\neg w$
$x \lor y$
$(x \lor y) \land z$
$((x \lor y) \land z) \Rightarrow (\neg w)$.

Now new variables a_1, a_2, a_3, a_4 are introduced and assigned for each sub-formula:
$a_1 \Leftrightarrow \neg w$
$a_2 \Leftrightarrow (x \lor y)$
$a_3 \Leftrightarrow (a_2 \land z)$
$a_4 \Leftrightarrow (a_3 \Rightarrow a_1)$.

These are then conjuncted, and on this conjuction, the last step is to replace equivalences and implications by standard Boolean operators rules. Resulting formula is equisatisfiable to ϕ and in CNF.

I will refer the reader to [14] Chapter 2 – "CNF Encodings" for fully detailed information on this topic.

Very obvious, but very important property of CNF is that, the formula in CNF is satisfied when all of its clauses are satisfied, and formula is unsatisfiable when at least one of its clauses is unsatisfiable.

Chapter 2

SAT solving

In this chapter, I will define SAT solver's related terminology and describe the principle of SAT solvers, as well as how SAT solvers work internally including the format of inputs and common heuristics they use today. Additionally, I will provide DPLL [15] (which is base for many modern solvers) and CDCL (version with restarts) [16] pseudo-codes, whose actual implementation is solver MiniSat, which I will be using throughout all experiments, and whose parameters setting is the main goal of this thesis. I will briefly explain their fundamental anatomy such as procedures they use, the principle of how solvers work and how most important heuristics used in these solvers work.

Definition 2.0.1. *SAT solvers* are computer programs that take a Boolean formula as an input (denoted by ϕ) whose goal is to output an assignment satisfying the input formula if such an assignment indeed exists, in other words, if the input Boolean formula is *satisfiable*. Otherwise SAT solver guarantees that there is no such assignment and formula is *unsatisfiable*

Modern SAT solvers expect the input Boolean formula to be in *conjunctive normal form* (CNF). Any Boolean formula can be transformed into CNF with only a linear increase in formula size using the Tseitin transformation [13], described in previous chapter. Hence, requiring CNF as input does not limit the space of allowable Boolean formulas. The advantage of CNF is that the SAT solver only needs to worry about clauses and as will be shown further in the text, modern SAT solvers implement propagation and clause learning techniques that operates efficiently over clauses. [17]

2.1 Solver's principle – resolution rule

Since DPLL algorithm (described in the next paragraph) is based on *resolution* rule, so I will explain what it is first here.

The resolution rule in propositional logic is a inference rule that produces a new clause implied by two clauses containing complementary literals. Two literals are said to be *complements* if one is the negation of the other, an example are c and $\neg c$. The resulting clause contains all the literals that do not have complements.

This rule is formally defined as follows:

$$\frac{a_1 \lor a_2 \lor \cdots \lor c, \quad b_1 \lor b_2 \lor \cdots \lor \neg c}{a_1 \lor a_2 \lor \cdots \lor b_1 \lor b_2 \lor \cdots}$$

or equivalently, and perhaps more transparently re-written as:

$$\frac{(\neg a_1 \land \neg a_2 \land \cdots) \rightarrow c, \quad c \rightarrow (b_1 \lor b_2 \lor \cdots)}{(\neg a_1 \land \neg a_2 \land \cdots) \rightarrow (b_1 \lor b_2 \lor \cdots)}$$

where a_i, b_j, c and $\neg c$ are literals, and fraction line stands for "logical consequence".

Two clauses produce third one, which does not contain variable c. Clauses produced by resolution rule are called the *resolvents*. This rule can be viewed as the principle of consensus applied to clauses rather than terms.

Most trivial example of this rule is fundamental logic rule "Modus ponens":

$$\frac{p \rightarrow q, \quad p}{q} \text{ is equivalent to } \frac{\neg p \lor q, \quad p}{q}.$$

2.2 DPLL

DPLL algorithm [15] is an algorithm to establish whether a CNF formula is satisfiable. DPLL is a complete search over the space of all possible assignments of the formula, while using backtracking. It was introduced in 1962 by Martin Davis, George Logemann and Donald W. Loveland, thus the name DLL after its authors surnames, the "P" in DPLL comes from the fact that DPLL was based on earlier work from 1960 by Martin Davis and Hilary Putnam's who developed DP algorithm [12], which is a resolution-based procedure for checking the validity of a first-order logic formula. Especially in older publications, the Davis–Logemann–Loveland algorithm is often referred to as the "Davis–Putnam method" or the "DP algorithm" instead of today's DPLL. In comparison to DP algorithm, DPLL removed the need for explicit representation of the resolvents.

There are several methods in the literature that form the basis for most SAT solver implementations. The first described satisfiability testing procedure that can be seen as a predecessor to modern SAT solvers is the aforementioned Davis-Putnam procedure (DPP), which also introduced the conjunctive normal form into satisfiability testing. DPP consists of three rules, which modify the set of CNF clauses **without** changing its satisfiability.

The recursive character of DPLL (which can be seen in later paragraph) allows us to define *decision levels* of truth assignments. Decision level n refers to the assignment implied by the decision in the n-th recursive call of DPLL and all assignments deduced from this assignment by unit propagation. When the solver backtracks to level l, all assignments with decision levels higher than l must be reverted back to the assignments of that level. This technique is ”smarter backtrack” referred to as *backjumping*.

2.2.1 Unit propagation rule

Definition 2.2.1. *Unit clause* is a clause which consists only of a single unassigned literal.

When algorithm encounters unit clause, it will *propagate* the consequences of this information to other clauses, hence the name *unit propagation*. To satisfy the formula, unit clause’s literal must be true because the clause is part of conjunction (formula is in CNF). Assuming true for the literal’s value, all clauses containing that literal are satisfied (because the clause is a disjunction by definition) and can be removed. All clauses containing the literal’s negation are modified by removing the negated literal, which cannot make a clause’s value become true, in other words, clause’s evaluation does not depend on the value of this literal.

2.2.2 Pure literal elimination rule

When an unassigned variable x only appears in its positive form or only if in its negative form in the set of undecided clauses, literal is said to be *pure*. That means if literal $\neg x$ doesn’t appear anywhere then it is safe to assume the pure literal’s value is true, and add $x = true$ to the current assignment, as this cannot cause any clauses to become unsatisfied, and as a consequence, satisfy all the clauses containing the literal x, this effectively means that all clauses containing this literal can be removed. similarly if variable x only appears in its negative form, $\neg x$, we can add $x = false$ to the current assignment.

2.2.3 Resolution rule

Resolution rule as defined in previous section can be used to eliminate a variable from the set of clauses. All clauses containing the variable are grouped into the set of literals with a positive occurrence and with a negative occurrence. All those clauses are then replaced with the new one clause instead, formed as disjunction of both sets. When this disjunction is converted back into the conjunctive normal form it becomes the set of all disjunctions formed by taking a clause of either set.

The resolution rule is problematic as it generates a large number of clauses, quickly exhausting the available amount of memory. This prompted Loveland and Logemann to replace it with a *splitting rule* that successively explores the conclusion of assuming a variable to be true or false, thereby creating a recursive algorithm. This is the DPLL procedure which still forms the basis for most recent complete SAT solvers. [18]

Iterative application of these rules reduces the set to either the empty set, if satisfiable, or to a set containing an empty clause (contradiction), if unsatisfiable.

The splitting rule on a variable x is implemented by recursively invoking the DPLL procedure twice, once with the variable assumed true and once assumed false. This is equivalent to adding a unit clause with the literal x or $\neg x$, which will trigger the unit propagation in the recursive call.

2.2.4 Pseudo-code

Algorithm 1 DPLL

```
 1: function DPLL-SAT($\phi$)
 2:     $clauses = \text{CLAUSESOF}(\phi)$
 3:     $variables = \text{VARIABLESOF}(\phi)$
 4:     $v = \emptyset$                                    ▷ Empty assignment
 5:     return DPLL($clauses, vars, v$)
 6: end function
 7:
 8: function DPLL($clauses, vars, v$)
 9:     if $\forall c \in clauses, v^*(c) =$ true then    ▷ $v^*$ denotes current assignment
10:         return true
11:     end if
12:     if $\exists c \in clauses, v^*(c) =$ false then
13:         return false
14:     end if
15:     $v = v \cup \text{UNITPROPAGATION}(clauses, v)$
16:     $v = v \cup \text{PURELITERALELIMINATION}(clauses, v)$
17:     $x \in vars \wedge x \notin v$                      ▷ Select unassigned variable
18:     return DPLL($clauses, vars, v \cup \{v(x) = true\}$) or
                DPLL($clauses, vars, v \cup \{v(x) = false\}$)
19: end function
```

Theorem 2.2.1. DPLL algorithm is sound and complete and always terminates.

It is an depth-first search style of searching through the space of all possible partial assignments. It is worth noting that the algorithm first performs unit propagation and then eliminates pure literals, because unit propagation could cause creating of new pure literals.

2.3 CDCL

The brute force nature of DPLL limits its practical utility. The CDCL [16] paradigm extends DPLL with a series of techniques and heuristics such as conflict-driven branching, clause learning, backjumping, and frequent restarts that dramatically improve its performance over the plain DPLL solver [17], which makes them so useful for solving real-life industrial problems.

2.3.1 Branching

Branching means the operation of selection of an unassigned variable together with selection its value, and then adding it to the current assignment. When variable is assigned by branching process is said to be *decision variable.* A pair of decision variable and its assigned value is called *decision literal.*

The *decision level* of a decision variable is the number of decision variables already in the assignment prior to assignment of selected variable. Decision level of a literal is the decision level of its corresponding variable.

The decision level of a propagated variable x is the number of decision variables in the assignment in the moment variable was being propagated.

For example, let $v = \{x \mapsto \texttt{true}, y \mapsto \texttt{false}\}$ be the current assignment, and lets suppose both x and y were assigned by branching, variables z and s are yet to be assigned, so the branching procedure will select say z and assign value `true`, and add it to v, the decision level of literal z is 2.

2.3.2 Backjumping

When conflict arises, solver needs to revert part of assignment, because it is not possible to further extend the current assignment because of of the clauses is falsified. More efficient way of backtracking was introduced in GRASP solver [16]. Unlike backtrack which only ”jumps up” by one level, backtracking by multiple decision levels is possible, sometimes this technique is called a *non-chronological backtrack.* CDCL solvers use this by backjumping to the smallest decision level at which the newest learnt clause becomes unit clause. For a learnt clause produced by a UIP cut, solver will backjump to highest decision level of all the variables in the learnt clause since it is *asserting clause*, this is the reason why *asserting clauses* are preferred.

2.3.3 Boolean Constraint Propagation (BCP)

Boolean constraint propagation (BCP) in context of CDCL solvers is just like unit propagation in DPLL. When a clause with n literals and assignment with $n - 1$ literals which evaluate to false, remaining literal must be set to true. BCP is first run at the start of the search and it will assign as many variables as possible just by propagation of unit clauses. Every time a variable is propagated, it cuts the current search space in half so its unnecessary for the solver to explore the other half for which this propagated variable would be assigned opposite value. So its natural that we want to trigger BCP often. BCP is triggered only on clauses for which there is only one unassigned literal left, intuitively shorter clauses are better at cutting the search space since they are more likely to propagate. BCP is responsible for maintaining a data structure called the implication graph. Implication graph will be defined in section about conflict analysis.

2.3.4 Conflict driven clause learning

CDCL SAT solvers implement approach of ”conflict driven clause learning” [16], hence the abbreviation CDCL. This means that the solver is dynamically analyzing conflicts during the recursion, and stores/”learns” new clauses which prevent occurrence of future conflicts, they are called *conflict clauses* or sometimes *learnt clauses*. The first implementation of clause learning was solver GRASP [16], simpler and more efficient way of clause learning was introduced by RelSat [19] and further improvements brought by Chaff [20].

Clause learning proceeds by following the normal branching process of DPLL until there is a ”conflict” after unit propagation. If this conflict occurs without any branches, the formula is declared unsatisfiable. Otherwise, the ”conflict graph”/”implication graph” is constructed and analyzed and the ”cause”/”reason” of the conflict is learned in the form of a ”conflict clause”. Then the algorithm backjumps and continue as ordinary DPLL, treating the now learned clause just like initial ones. A clause is said to be known at a stage if it is either an initial clause or has already been learned. The learning process is expected to save us from redoing the same computation when we later have an assignment that causes conflict due in part to the same reason [21].

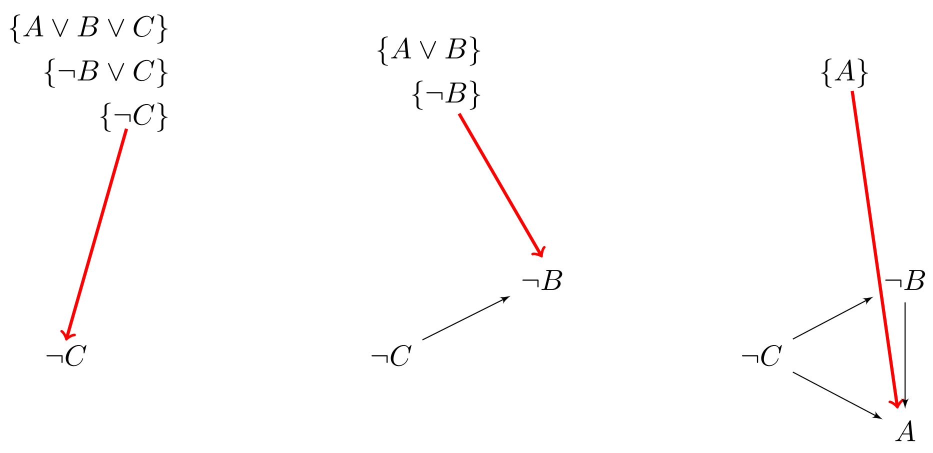
2.3.5 Conflict analysis and clause learning

The conflicts happen for a logical reason, meaning that following the sequence of assignments of some clauses will lead to a conflict (clause is falsified in the current assignment). The solver is ”navigated” by newly learned reasons (learnt clauses) and will avoid conflicts hence the name ”conflict driven”. Learning clauses is crucial because they prune the search space. To derive a clause that will help with avoiding that reason which led to an occurrence of conflict, concept of ”implication graph” is used.

Definition 2.3.1. *Implication graph* is a directed acyclic graph. Vertices are all literals of current assignment. Every time the unit propagation happens, new edges pointing to the literal of the unit clause are added from each negated literal of the original clause. Additionally, falsified clauses have edges from the negation of each of its literal to a special vertex $\perp$.

When the unit propagation deduces an assignment, it is due to a clause, which became unit. This clause is called the *reason* of the assignment. Implication graph can be viewed as chaining of these events.

Example:



From the example it can be seen that assignment of literal A is true, and the reason for that are literals $\neg B, \neg C$.

In more complex case with more clauses, a conflict may arise. The conflict is when it is deduced that a literal's X value should be both true and false. Algorithm will learn a new clause from this conflict by *cutting* the implication graph in two parts, one partition has all the decision assignments on one side (*reason side*) and the conflicting assignments on the other side (*conflict side*).

The idea of cutting the graph is to separate literals which caused conflict, and learn the *conflict clause* from reason side of the cut. A clause containing all negated literals of nodes with an outgoing edge crossing the cut is then constructed. Saving this clause and treating it as if it was part of the original CNF allows algorithm to avoid encountering same conflict. But this technique of course memory demanding and solvers usually offer user to control how much memory can be used for these new clauses. Another approach is that size of the set of learnt clauses is managed dynamically by assigning each clause "timeout", meaning that if that clause was not used for a long time it will be removed by periodical garbage collection service. More about this heuristic will be described in later section about how branching works.

There are usually multiple possible ways of cutting the implication graph, but some cuts are preferred in comparison to other, because they yield better learnt clause.

Depending on the graph cut used, it is necessary to backtrack multiple levels until the new learnt clause becomes unit. The new clause can be unit for multiple levels and in practice backtracking to the lowest level where the clause is unit, known as *backjumping* has been shown to be effective [16] [18].

When learning clauses, special clauses are to be preferred, called *asserting clauses.*

Definition 2.3.2. *Asserting clause* is a clause with exactly one literal from the current decision level.

Why *asserting clauses* are preferred is interconnected with backjumping and will be explained later.

To find cuts, which will lead to asserting clauses, it is necessary to find special vertex called *unique implication point.*

Definition 2.3.3. A *unique implication point (UIP)* is a vertex of literal in the implication graph such that all paths starting from the decision literal with the highest decision level to the vertex $\perp$ must cross the UIP vertex.

There can be many UIPs in the graph, but usually the closest one to the conflict vertex $\perp$ is picked, sometimes referred to as *1-UIP*. In this thesis, by UIP I will refer the 1-UIP.

The learnt clause by cutting at UIP contains exactly one literal with the highest decision level, that being the UIP literal itself and thus it precisely follows definition of an *asserting clause*. So the clause produced by cutting implication graph at UIP gives a clause that is preferred.

The clause learning technique implemented by CDCL SAT solvers is essentially applying the resolution rule. That is, we can apply the resolution rule to the clauses on the conflict side of the implication graph to construct the exact same learnt clause as a CDCL SAT solver. A CDCL SAT solver returns unsatisfiable when a conflict occurs at decision level zero, and this is equivalent to inferring an empty clause as a resolvent of the resolution rule. By this perspective, a CDCL solver can be viewed as a resolution engine constructing a proof of the empty clause for an unsatisfiable CNF input. [17]

2.3.6 Restarts

Restart means that the solver will discard the current assignment completely, delete the implication graph as well and then will start the search again. Restart strategy was introduced in [22]. It can be seen as backjumping back to decision level 0. Restarts can be triggered in the moment when BCP does not result in conflict. The decision whether and when to perform a restart is up to the restart heuristic. This strategy is used with great success in practice. It might appear like solver is getting rid of all what it was working for but the important thing to note is, its only deleting the assignment, but keeping the clause database that it built in the previous run.

2.3.7 Pseudo-code

Algorithm 2 CDCL with restarts

```
function CDCL($\phi$)
    $v = \emptyset$                                        ▷ Empty assignment
    loop
        $v = \text{BCP}(\phi, v)$                          ▷ Try solve by unit propagation
        if $\exists conflictingClause \in \phi$ such that
          $\text{Falsified}(conflictingClause, v)$ then
            if $\text{CurrentDecisionLevel}(v) = 0$ then
                return false
            end if
            $learntClause = \text{ConflictAnalysis}(conflictingClause, v)$
            $\phi = \phi \cup learntClause$                ▷ Clause learning
            $v = \text{BTtoLevel}(GetBTLevel(learntClause), v)$
        else if $|v| = |\text{variablesOf}(\phi)|$ then
            return true
        else if $RestartCondition = \texttt{true}$ then
            $v = \text{BTtoLevel}(0, v)$                   ▷ Restart
        else
            OptionalClauseDeletion( )                      ▷ Remove not-so-useful clauses
            $var = \text{BranchingHeuristic}( )$           ▷ Select next variable
            $value = \text{PolarityHeuristic}(var)$        ▷ Pick its value
            $v = v \cup \{var \to value\}$                 ▷ Branch
        end if
    end loop
end function
```

Theorem 2.3.1. CDCL algorithm is sound and complete and always terminates.

Proof of soundness and completeness is not trivial and can be found in [16].

Chapter 3

MiniSat's heuristics and parameters

This short chapter will introduce and briefly explain few heuristics which are implemented in MiniSat solver, and I will state which of these heuristics am I going to use for experimenting, and eventually which will be tuned in terms of predicting parameter's value which will be passed to the solver, which is controlling heuristic's effect, such as probability of choosing a new variable by decision heuristic, base restart interval and more.

3.1 Tuned parameters

In this thesis I will tune the following heuristics settings of MiniSat solver:

`-var-decay`	the VSIDS's decay factor
`-cla-decay`	the clause decay factor
`-rfirst`	base restart interval
`-rinc`	restart interval increase factor

3.2 Decision heuristic

Duty of decision/branching heuristic is to pick a variable which is yet unassigned and assign it value. Decision heuristic is divided into two parts: the variable selection heuristic (typically VSIDS [20]) is responsible for selecting which variable to branch on and the polarity heuristic (typically phasesaving [23]) is responsible for selecting the value [17]. For all the experiments and measurements I always used phasesaving. Few manual tests has led me to this decision because whenever I turned off phasesaving in Minisat solver I have obtained only worse results, meaning the solving time was always longer than with phasesaving turned off.

3.2.1 VSIDS

VSIDS is an abbreviation of *variable state independent decaying sum.* Many different branching heuristics were developed over the course of years, but VSIDS has gained dominant place in SAT community, and has become a standard choice for many popular SAT solvers, such as Minisat [8] which I employed as default solver for my thesis.

The main idea of VSIDS heuristic is to associate each variable with floating point number, an *activity* factor, which signifies a variable's frequency of appearing in recent conflicts via the mechanism of *bump* and *decay.*

Bump is a number which is incremented by 1 every time this variable appears in conflict (doesn't matter if it is on the conflict side or reason side). There is slightly modified version to this, which only increment bump if variable appears in learnt clause, but Minisat does not use this variant.

Decay factor $0 < \alpha < 1$ is a number by which each of the variable's activity is multiplied after each conflict and thus decreased. A naive implementation of decay takes linear time to perform the multiplication for each variable activity. MiniSat [8] introduced a clever implementation of decay that reduces the cost to amortized constant time. [17]

3.3 Clause decay

In MiniSat, similar principle as in VSIDS is used and applied to clauses. When a learnt clause is used in the analysis process of a conflict, its activity is incremented. Inactive clauses are periodically removed from the *learnt clauses database* [8]. This strategy can be viewed as attempting to satisfy the clause involved in a conflict, but particularly attempting to to satisfy the most recent clauses involved in a conflict. In fact, the decision heuristic of MiniSat involves decaying the activity of clauses more often than the standard VSIDS heuristic. Benchmarks have shown that this schema responds faster to changes and avoid branching on out-dated variables.

Using this heuristic, the clauses with the highest activity values represent the clauses most actively involved in recent conflicts. Since a set of unsatisfiable clauses generates many conflicts, and therefore many conflict clauses, the high activity of a clause can be seen as a potential sign of unsatisfiability. [24]

3.4 Random frequency

The frequency with which the MiniSat choose a random variable rather than the one determined by heuristic. It is a probability parameter and thus allowed values are $0 \leq \epsilon \leq 1$.

3.5 Restart frequency

Frequency of restarts in MiniSat is determined by two parameters, the *base restart interval* and *restart interval increase factor*. One round of search will take as long until the search encounters given number of conflicts. For example, `minisat(120)` will be searching space of assignments as long as it reaches count of conflicts equal to 120. After that, the algorithm will pause, determine new number of needed conflicts to force next restart, and continue searching.

Number of needed conflicts to restart L is determined as follows

$$L = \texttt{restart\_base\_interval} \cdot \texttt{restart\_inc\_factor}^{\#restarts}.$$

Chapter 4

Related work

In this chapter I will provide an overview of some of the related works, which faced similar challenges as this thesis, or whose method has directly or indirectly affected my work.

4.1 Portfolio solver: SatZilla

SatZilla [25] is a portfolio solver, which won many awards in SAT Competition [26]. It introduces new approach of using many other solvers (portfolio) in the background. The solvers are used as-is, and SatZilla does not have any control over their execution.

Machine learning was previously shown to be an effective way to predict the runtime of SAT solvers, and SatZilla exploits this. It uses machine learning to predict hardness of the input instance, and then based on this prediction select a solver from its portfolio which will be assigned to solve the problem. Machine learning technique for predicting is ridge regression.

This works because different solvers are better for different types of instances, there is no universal solver which would perform very good on every category of instances, thus the key is to predict what kind of input instance it is and determine which solver would suit best to solve it.

First step is to identify one or more solvers to use for pre-solving instances. These pre-solvers are then ran for a short amount of time before features are computed, in order to ensure good performance on very easy instances and to allow the hardness predicting models to choose solver exclusively on harder instances.

Predicting hardness of an instance is done by first extracting various features from the input. To be usable effectively for automated algorithm selection, these features must be relatively cheap to compute.

To train the model, SatZilla will first compute some features on training set of problem instances and run each algorithm in the portfolio to determine its running times. If feature computation cannot be completed for some reason (error or timeout), backup solver will be determined and used for solving this instance.

In summary, when the model for predicting the runtime of the solvers has been already trained, SatZilla is ready to be used in practice. When the new input instance comes, it computes its features. These are then used as input for predictive model which predicts the best solver to be used. That particular solver is then used to solve instance.

4.2 Parameter tuning: AvatarSat

AvatarSat [27] is a modified version of Minisat 2.0. AvatarSat introduced two key novelties.

First one is that it used machine learning to determine the best parameter settings for each SAT formula. The machine learning technique used is Support Vector Machine.

Second novelty in AvatarSat is the ”course correction” as it dynamically ”corrects” the direction in which solver is searching. Modern SAT solvers store new learnt clauses and drop input clauses during the search, which can change the structure of the problem considerably. AvatarSat’s argument is that the optimal parameter settings for this modified problem may be significantly different from the original input problem.

AvatarSat therefore first selects values of parameters for MiniSat to use during the initial part of the search. When the number of new clauses accumulated during the search crosses a threshold, the ”course correction” procedure examines the new clauses to select a new set of parameters for MiniSat to use after the restart. This is the principle of AvatarSat which dynamically adapts the parameter settings to the potentially changing characteristics of the SAT problem.

Input SAT instances are classified using features extracted from the instance. Each class corresponds to the best configuration for the SAT-formulas belonging to the class. The approach is similar as I applied in this thesis.

AvatarSat uses 58 different features of SAT formulas such as ratio between variables and clauses, number of variables, number of clauses, positive and negative literal occurrences etc. Features used are very similar to those of SatZilla, which were originally designed to measure hardness of the instance and not its structure.

The classifier is trained by running a number of configurations of Minisat on a number of sat instances, then each instance is paired with the optimal configuration and that data is used to train the classifier with Radial Basis Function (RBF). When the classifier is trained, Minisat can be configured for any SAT formula by classifying the formula and then retrieving the configuration associated with that class.

AvatarSat is tuning only two parameters, `-var-decay` and `-rinc`, Nine values for the first one and three for the second one, so the number of examined configurations is 27.

4.3 Iterated local search

The key idea underlying iterated local search is to focus the search not on the full space of all candidate solutions but on the solutions that are returned by some underlying algorithm, typically a local search heuristic. [28]

Iterated local search is a local search algorithm that optimalizes parameters but only one dimension at a time, it is a one-dimensional variant of hill climbing. It checks configurations differing in one single change from the current configuration until it finds a better one. Iterative first improvement are repeated until no improvement can be made by a single change. This is most likely to be a local extreme. To escape the local extreme a fixed number s of random changes are made, and the process of making iterative first improvements is resumed until a new local maximum is reached and the process of perturbation and iterative first improvements are repeated until some termination criterion is reached. [29]

In [29] author has used this algorithm for MiniSat's parameter tuning. There was not feature extraction approach as in AvatarSat, SatZilla and this thesis, but it was an attempt to tune SAT solvers parameters and therefore I mention his solution in this chapter. Results were measured only on cryptographic instances, concretely factorization problem instances, but this approach performed surprisingly well. As author has mentioned, it is an open question how the tuned solver would perform on bigger instances.

Chapter 5

Impact of parameters

The term “structure”, due to its vagueness, leaves much room for interpretation, though, and it remains unclear how this structure manifests itself and how exactly it should be exploited. [30]

However, research has advanced since then, and nowdays the structure of some instances can be exploited.

Base idea of my thesis comes from paper [31], where it was shown that industrial instances exhibit ”hidden structures” based on which solver is learning clauses during search. In [23] researchers have shown that formulas with good community structure tend to be easier to solve.

Variables form logical relationships and we hypothesize that VSIDS exploit these relationships to find the variables that are most ”constrained” in the formula. The logical relationship between variables are concretized as some variation of the variable incidence graph (VIG). [17]

My idea is to exploit this fact, so I will construct a graph which is representing each instance, compute various properties of this graph which will be used as features of instances for machine learning, in addition to standard features of the instance like number of variables, clauses, their ratios etc.

In this chapter I will present results of my initial data exploration. I performed several observations on four different classes of problems. On these classes I observe how the solver’s parameters affect then number of conflicts, and thus solving time.

5.1 Classes of SAT instances selected

For this thesis I limited myself to four SAT problem classes for which I will use machine learning to set solvers parameters. I have chosen these:

- Random SAT/UNSAT
- Pigeonhole problem
- Planning
  - $n^2 - 1$ problem
  - Hanoi towers
- Factorization

I have selected these because they are **structurally diverse**, and I expect them to have different demands on parameters.

Random SAT/UNSAT are instances of 3-SAT problem, generated randomly. This class is also part of SAT category in which various solvers are compared by efficiency they achieve. In SAT competition context the category is simply called *RANDOM*.

Pigeonhole problem involves showing that it is impossible to put $n + 1$ pigeons into n holes if each pigeon must go into a distinct hole. It is well known that for this combinatorial problem there is no polynomial-sized proof of the unsatisfiability. [32] Combinatorial problems are also part of SAT competition category, known as *crafted.* I chose pigeonhole problem as one representative, because it can be generated easily with random starting positions with fixed size.

Planning problems representatives I chose planning problem known as $n^2 - 1$ *problem*, or in its fixed size form *Lloyd's fifteen.* I generate numbered tiles order randomly, and aggregate these instances always within fixed size, for example, I never combine problem of size 4x4 with 5x5. Another representative of planning is Hanoi towers problem, this problem is not possible to "randomize" because initial state is always given, this problem I added out of curiosity to observe if it will exhibit some similarities to $n^2 - 1$ planning problem.

Factorization problem is problem of determining whether a big number is a prime or otherwise it can be factored, is an example of typical *INDUSTRIAL* instance from SAT competition. I chose factorization because its easy to generate random instances, by simply generate one big random number and build SAT formula. I will observe for MiniSat deals with satisfiable and unsatisfiable instances. When factorization instance is satisfiable, it means that the number is not prime.

5.2 Correlation of number of conflicts and solving time

For the initial dataset building process, it is necessary to find close to optimal solver parameters for which the solving time is lowest possible.

Since some of the harder instances take several tens of seconds to solve, it would be unfeasible to generate dataset from solving each of these instances by brute force search on grid of parameter values, thus I decided to build this dataset from small instances from SATLIB. These instances are usually solved within fraction of second by Minisat solver, but this approach has a downside, it is hard to capture real solving time, because for these small instances overhead usually outweigh useful computation time.

Solving time varies a bit with every run and solving time captured by MiniSat also includes several system-originated factors which are not desired to take part in the dataset. However, computation is deterministic with fixed initial random seed, and number of conflicts for concrete parameters stays always the same for every run, so it is natural to use number of conflicts as other metric instead of actual solving time.

The following scatter plot shows strong correlation between number of conflicts and solving time on randomly selected instance from SATLIB, thus it is correct to use conflict count as measurement of performance of the parameters.

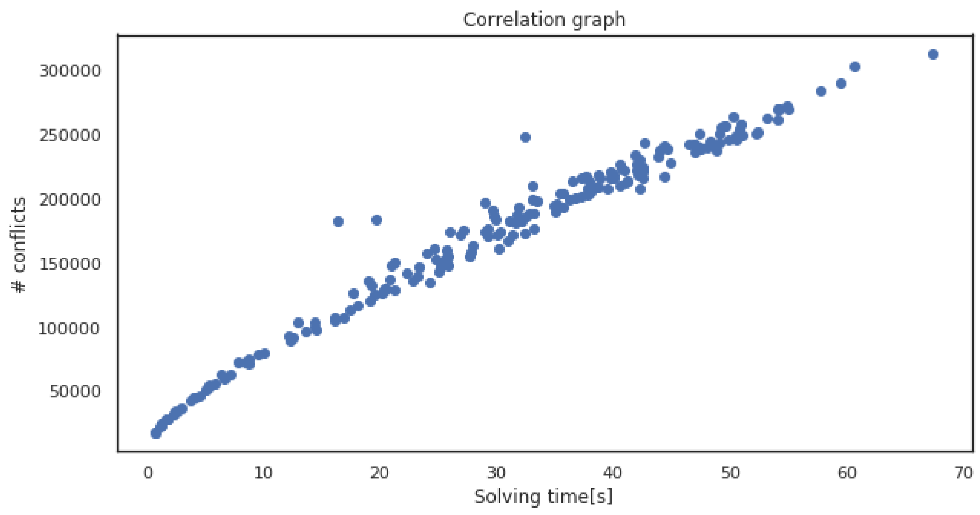


Figure 5.1: Correlation between number of conflicts and solving time

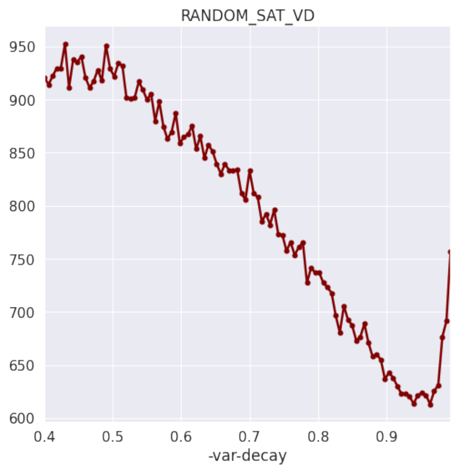
The implication from this observation for this thesis is, that I decided to measure conflicts instead of time, in this chapter and also in the final chapter about experiments.

5.3 Observed parameters

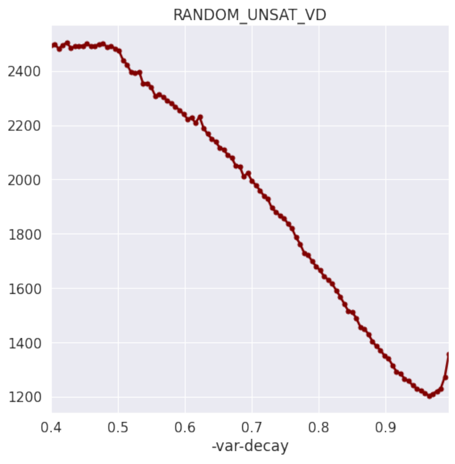
Each of the subsections shows how the parameter's value depend on number of conflicts for each of aforementioned problem classes. Note that axes do not have same values in each of the examples. Before I made these plots I first analyzed at what intervals should I chose to discretize. For example I observed that *variable decay* parameter for values in $(0, 0.4)$ always gave bad results so I avoided those and only examined $[0.4, 1)$. For space-saving purposes I have omitted label of vertical axis, and it will always be **number of conflicts**.

Following plots are results of several measurements of impact of one parameter at the time. For each class of problems I have gathered multiple instances, either downloaded or generated by myself. I ran MiniSat on every instance, with concrete parameter value and recorded number of conflicts encountered during the search with that value. Then I repeated the same with different initial random seed and computed mean of number of conflicts for each instance. Lastly I aggregated results from previous step for each class of problems by computing mean of conflicts for every parameter's value.

5.3.1 Variable decay



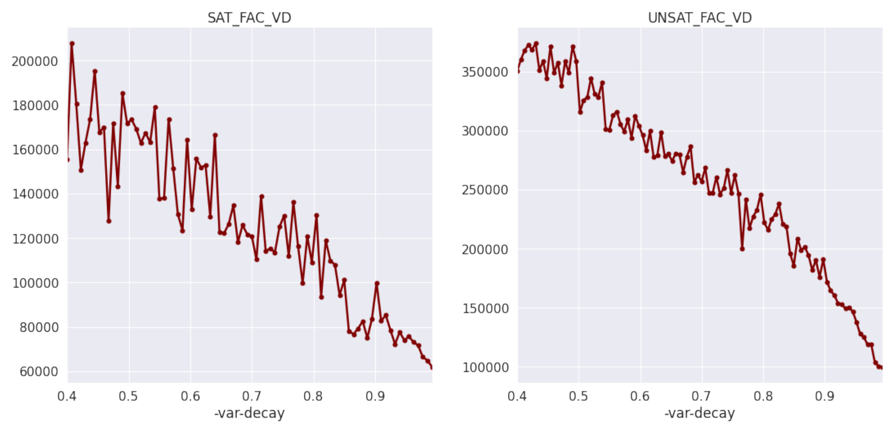
(a) sat



(b) unsat

Figure 5.2: Variable decay on random SAT instances

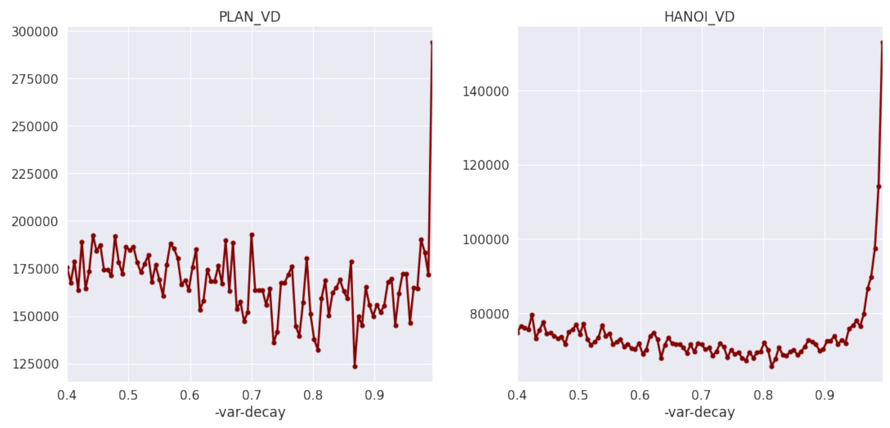
For random SAT instances in (a) and (b) it is obvious that variable decay around 0.95 gives best results in terms of least conflicts. For unsatisfiable instances the plot line is also ”smoother”, this is probably because the solver has to search entire space so there are not many backjumps which would cut ”heavy” branches.



(a) not prime (b) prime

Figure 5.3: Variable decay on factorization instances

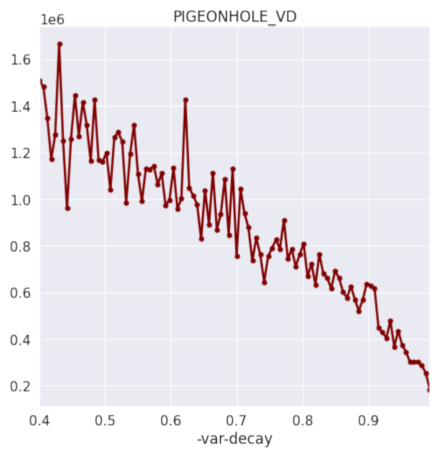
Factorization instances seem to require `-var-decay` close to 1 for fastest solving. Again the plot for unsatisfiable instances – prime numbers (b) is smoother.



(a) $n^2 - 1$ (b) hanoi

Figure 5.4: Variable decay on planning instances

Obviously plots for planning instances are very different from previous instances. As `-var-decay` increases from 0.9 higher number of conflicts rises rapidly, this might mean that planning instances have variables which are more-less independent, because setting variable decay factor close to value of 1, effectively means algorithm will decay activity of variables very slowly.

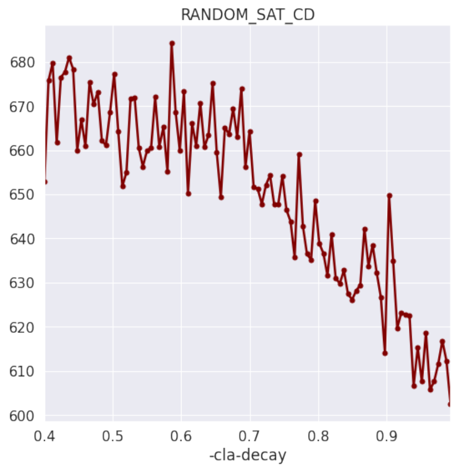


(a) pigeonhole

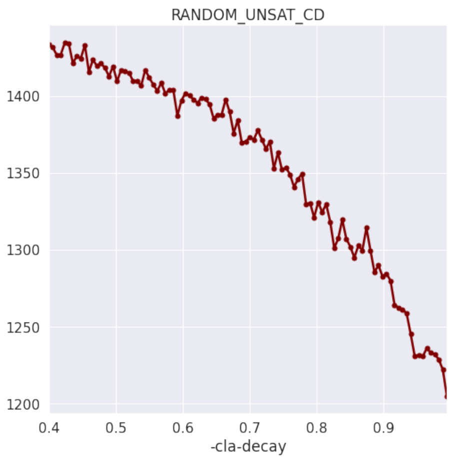
Figure 5.5: Variable decay on pigeonhole instances

Behavior of pigeonhole problem instances seems to be similar to factorization.

5.3.2 Clause decay



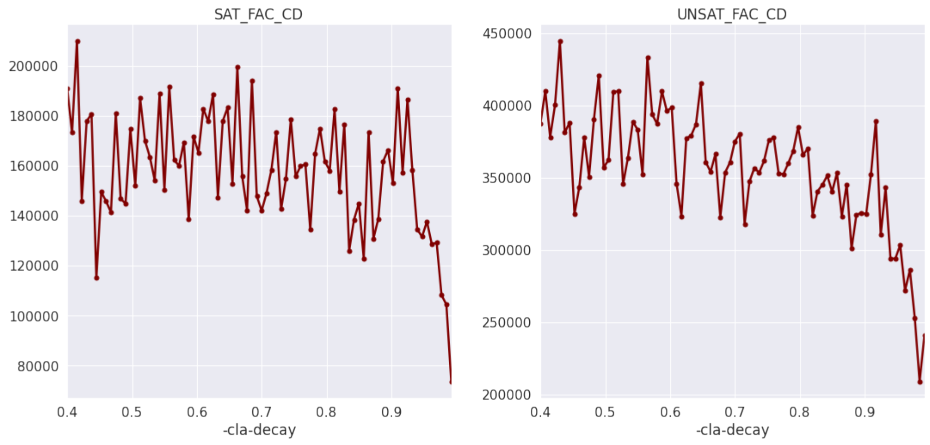
(a) sat



(b) unsat

Figure 5.6: Clause decay on random SAT instances

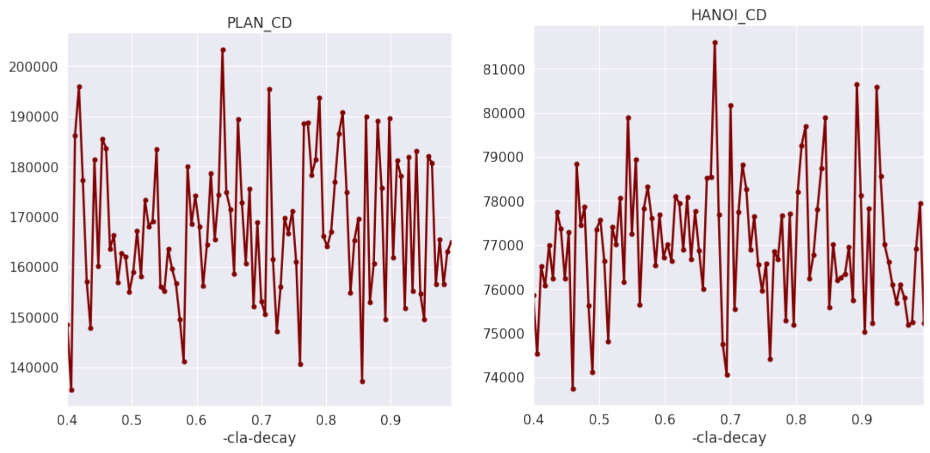
Plots show that for random instances the fastest solving time is when parameter is set to value very close to 1.



(a) not prime (b) prime

Figure 5.7: Clause decay on factorization instances

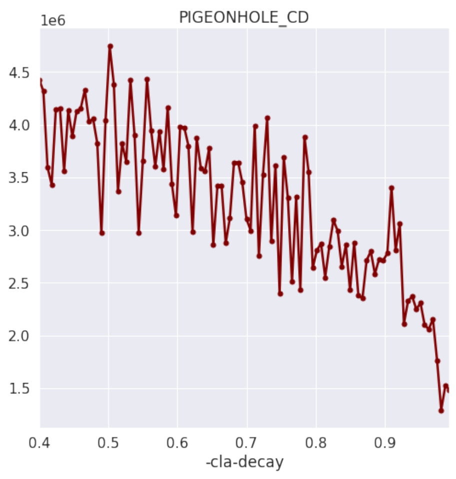
Similar results as for random instances can be seen here, but the trend starts to decrease at approximately value of 0.93.



(a) $n^2 - 1$ (b) hanoi

Figure 5.8: Clause decay on planning instances

These rather chaotic plots suggest that `-cla-decay` parameter does not matter very much for planning instances.

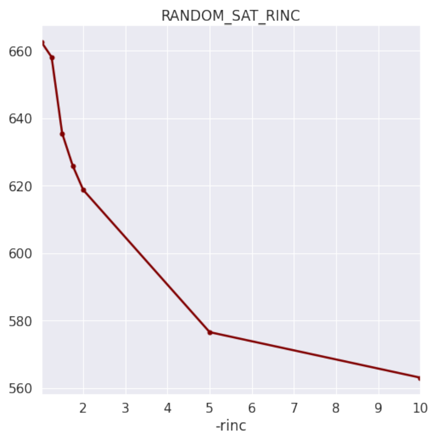


(a) pigeonhole

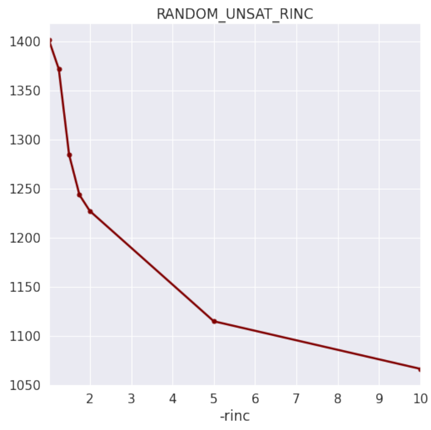
Figure 5.9: Clause decay on pigeonhole instances

From the plot it can be seen that there is a decreasing trend from clause decay. This is similar result as for random and factorization instances.

5.3.3 Restart interval increase factor



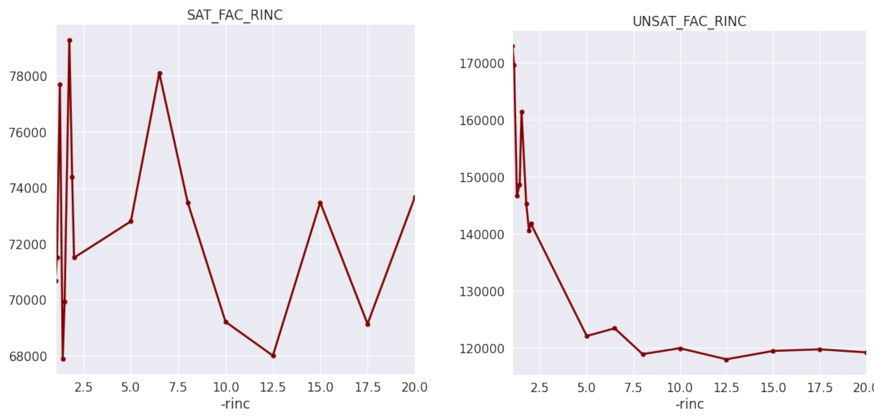
(a) sat



(b) unsat

Figure 5.10: Restart interval increase factor on random SAT instances

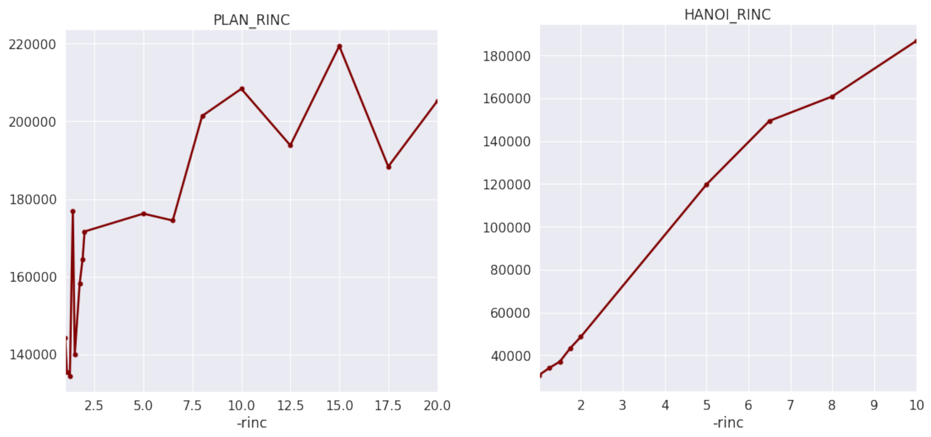
Higher value seems to be better for random instance, but important note is that the instances on which I performed these aggregations are of smaller size than in SAT competitions [26]. For those, these plots could look very different. I hypothesize that for big instances smaller value of this parameter would be better, because if the value is too high, it might mean that longer the solver is running, the interval until next restart will be too high, so the much needed restart would not happen in very long time.



(a) not prime (b) prime

Figure 5.11: Restart interval increase factor on factorization instances

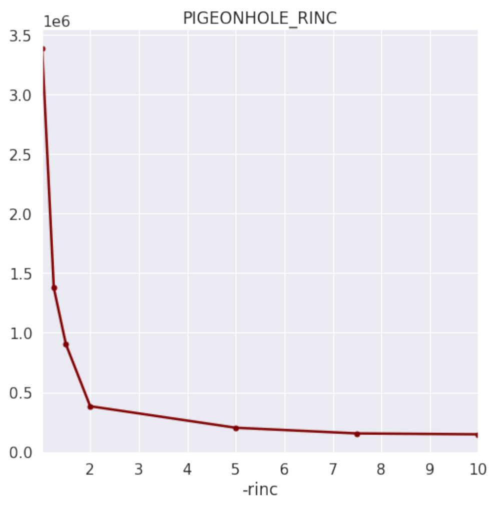
For not prime instances its hard to say which value could be suitable, prime instances show that values over 10 seems like fastest.



(a) $n^2 - 1$ (b) hanoi

Figure 5.12: Restart interval increase factor on planning instances

For planning instances it is clear that lower values are faster that higher, (b) graph is clear, and on (a) there is slightly raising trend. But again I think this is dependent on instance size. Note that x axis have different values on (a) and (b). Lower value means more frequent restarts, that is suggesting that solver is often in some local optima, which eventually will not lead to solution.

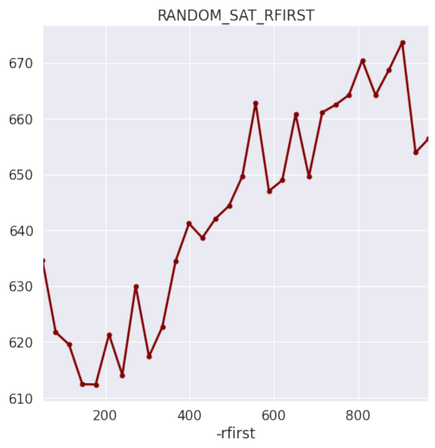


(a) pigeonhole

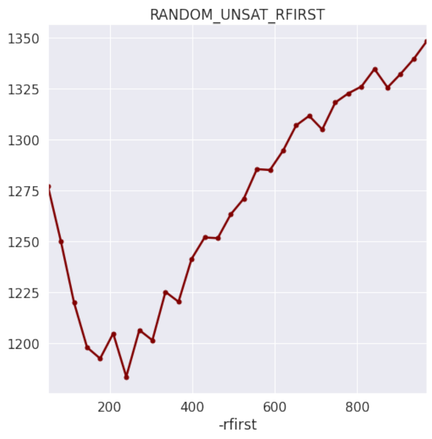
Figure 5.13: Restart interval increase factor on pigeonhole instances

Plot shows that for value 5 and higher impact of this parameter does not show very significant improvement. Higher values are preferred for pigeonhole problem, this means that the problem demands less restarts because it is doing some useful work, in other words, the path to solution is narrow so there are not many branches which lead to nowhere.

5.3.4 Restart interval base



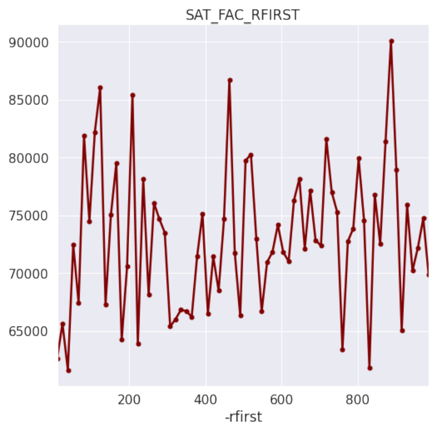
(a) sat



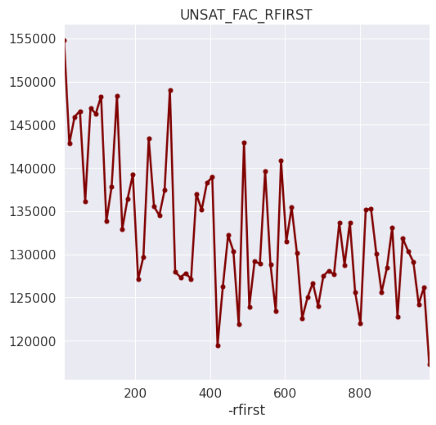
(b) unsat

Figure 5.14: Restart interval base on random SAT instances

Initial restart interval around 200 conflicts seems best for random instances.



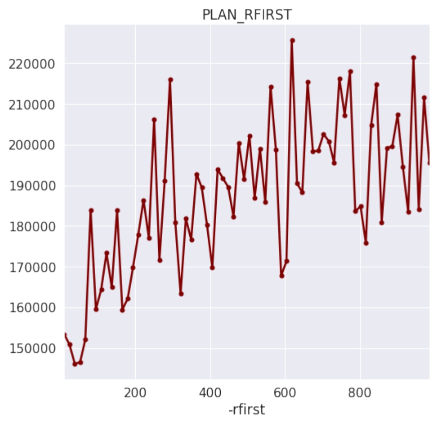
(a) not prime



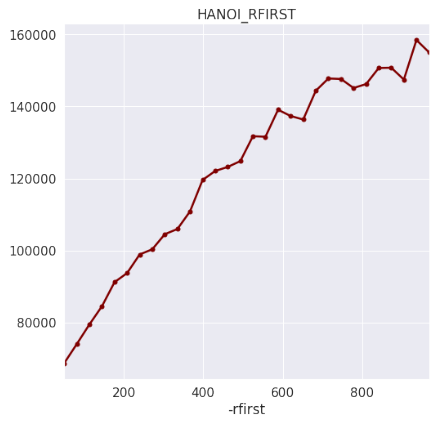
(b) prime

Figure 5.15: Restart interval base on factorization instances

Nothing can be really drawn from this, apart from that this parameter does not matter for factoring instances.



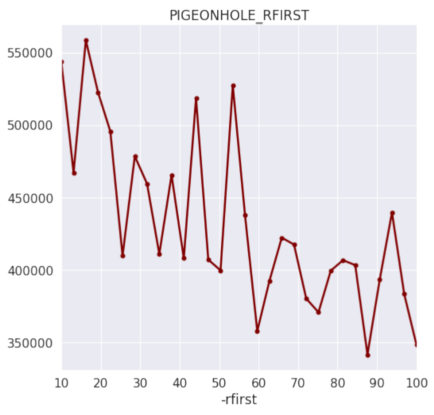
(a) $n^2 - 1$



(b) hanoi

Figure 5.16: Restart interval base on planning instances

On both (a) and (b) there is increasing trend, so for planning instances smaller initial interval is needed, and thus more frequent restarts.

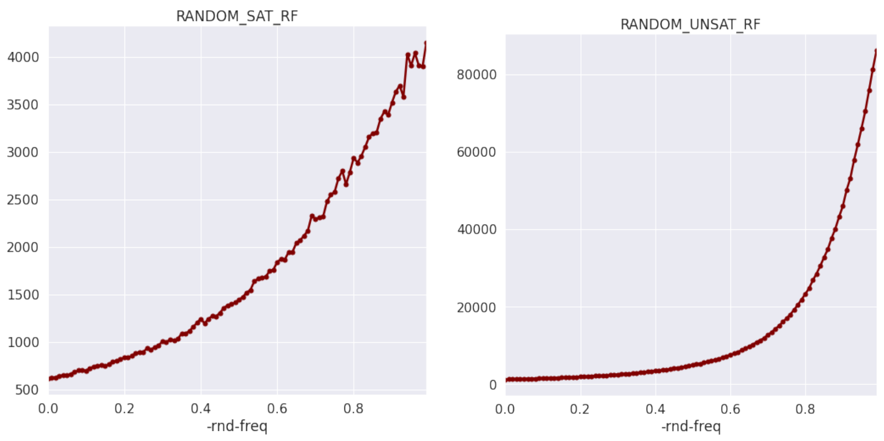


(a) pigeonhole

Figure 5.17: Restart interval base on pigeonhole instances

Decreasing trend suggests that less frequent intervals perform better for pigeonhole instances, this further confirms the hypothesis made with restart increase factor in previous section.

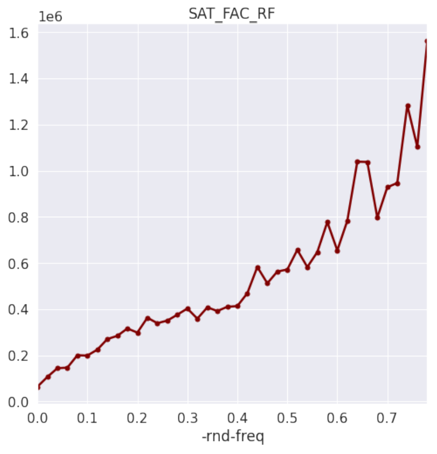
5.3.5 Random variable frequency



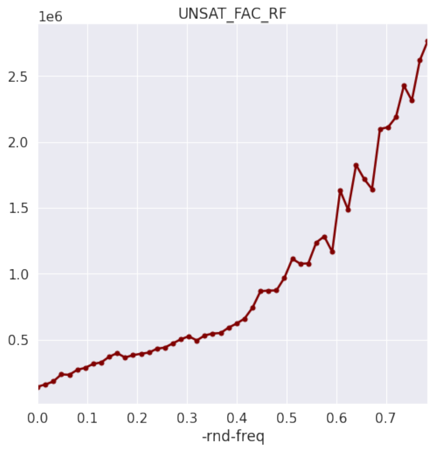
(a) (b)

Figure 5.18: Random variable frequency on random SAT instances

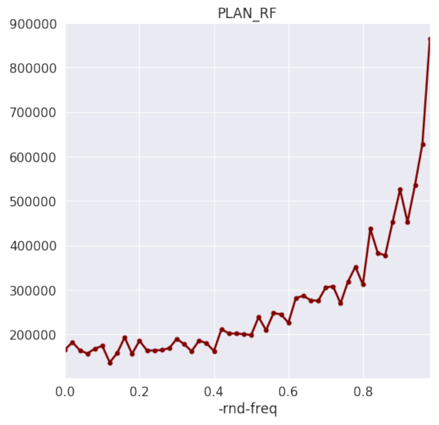
It is evident that that higher the probability the solver chooses random variable, the worse. The same is observed for all of the other classes of instances, so I will only provide plots without further comments.



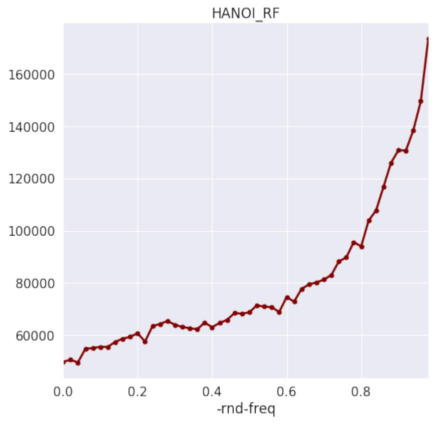
(a)



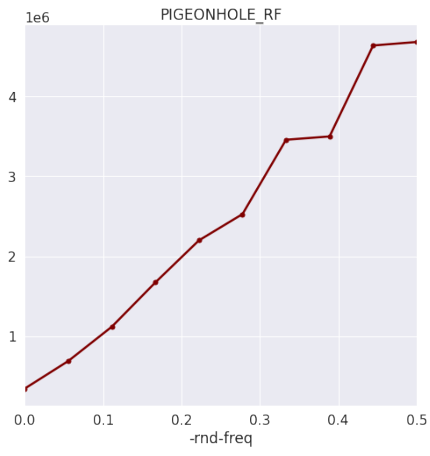
(b)



(c)



(d)



(e)

Figure 5.19: Random variable frequency on factorization, planning and pigeonhole instances

5.4 Implications for parameter tuning

The results show that there are some dependencies among parameters and solving time, thus it makes sense to try and implement machine learning system to set these parameters automatically depending on input instance.

It is debatable whether I should include parameter `-cla-decay` in the list of parameters which will be learned by machine learning technique, since for all classes of instances value close to 1 was best. I included this parameter nevertheless.

The parameter `-rnd-freq` I will not include, because it is clear from the smooth lines on the presented graphs in previous section, that in every instance higher value yielded only worse results, and thus I take it as good suggestion that value of 0 of this parameter is best.

Later when creating dataset for machine learning technique, for each class of problems I will use grid-search to find optimal parameters. For each instance of one of the classes, I will set up the grid of parameters to be searched. The values in this grid will be those, for which I observed to be successful individually. For example, for random SAT instances I will search the space of `-var-decay` $\in [0.8, 0.95]$ and `-rfirst` $\in [50, 400]$, because I saw that values from these intervals showed the best results.

Chapter 6

Parameter tuning

In this chapter I will describe each of the components of the process of parameter tuning for MiniSat solver.

Starting with overview of features extracted from SAT instances which try to describe the structure of the instance as closely as possible.

Next stage is to prepare a dataset for machine learning technique.

In comparison to related works which I listed in chapter 3, I took a slightly different path in stage of actual learning, instead of treating this problem as classification task, where features are used for classifying each instance into class of the problem it most likely belong to, and only then set parameter values, which are predetermined for each class, I will directly predict values. Thus the dataset constructed will have n features, where n is number of extracted features, and 4 target features (parameters: `-var-decay`, `-cla-decay`, `-rinc`, `-rfirst`). Thus, the approach I implemented is doing an multi-output regression.

6.1 SAT instance features

6.1.1 Basic formula features

By basic features I mean characteristics of the instance which were used in SatZilla's [25] feature extractor which I have used with an option `-base`.

These features can be classified into three categories:

1. problem size
2. balance
3. proximity to Horn formula

Problem size features are number of clauses, number of variables, their ratio. Balance features are ratio of positive and negative literals in each clause, ratio of positive and negative occurrences of each variable, fraction of binary and ternary clauses. Horn statistics are fraction of *Horn clauses* [33] , number of occurrences in a Horn clause for each variable.

This is just an short overview of what kind of features are extracted by SatZilla. Detailed description of these features is available in [25].

6.1.2 Structural features

To extract structural features of a SAT problem instance, I have decided to use three common types of graph representations of a formula as defined next.

Definition 6.1.1. *Variable graph (VG)* has a vertex for each variable and an edge between variables that occur together in at least one clause.

Definition 6.1.2. *Clause graph (CG)* has vertices representing clauses and an edge between two clauses whenever they share a negated literal.

Definition 6.1.3. *Variable-clause graph (VCG)* is a bipartite graph with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs in a clause.

From the input instance I construct each VG, CG and VCG. All of these graphs correspond to constraint graphs for the associated *constraint satisfaction problem (CSP)* [34]. Thus, they encode the problem's combinatorial structure.

For these three graphs I use basic node degree statistics from [25]. Feature extractor computes degree for every node in each of three types of graphs, so there are three separate arrays of numbers, from these arrays five statistics are computed: minimum, maximum, entropy of the array, variance and mean, that adds up to 15 features. Since SatZilla's extractor is standalone application, I decided not to append to their source and implement my own.

I constructed same 3 types of graphs (VG, CG and VCG), and additionally computed several graph properties which I thought could help describe instance's structure closely, and at the same time are not too much time expensive.

- Variable graph features
  - diameter
  - clustering coefficient
  - size of maximal independent set (approx.)
  - node redundancy coefficient
  - number of greedy modularity communities
- Clause graph
  - clustering coefficient

  - size of maximal independent set (approx.)

- Variable-clause graph
  - latapy clustering coefficient
  - size of maximal independent set (approx.)
  - node redundancy coefficient
  - number of greedy modularity communities

The VG is usually smallest in terms of number of nodes, thus I could compute more properties on this graph, such as modularity communities and diameter.

In contract CG is the biggest graph (there are more clauses than variables) and thus I limited the number of features extracted from this graph to only two, relatively easy–to–compute features.

VCG has the highest number of nodes among these three types of graphs ($|Vars| + |Clauses|$), but as defined earlier, it is a bipartite graph and some of the features are easier to compute on bipartite graph than on standard graph.

6.1.3 Clustering coefficient

Clustering coefficient can be computed for every node. It measures how close its neighbors are to being in a clique (neighbors form a complete graph).

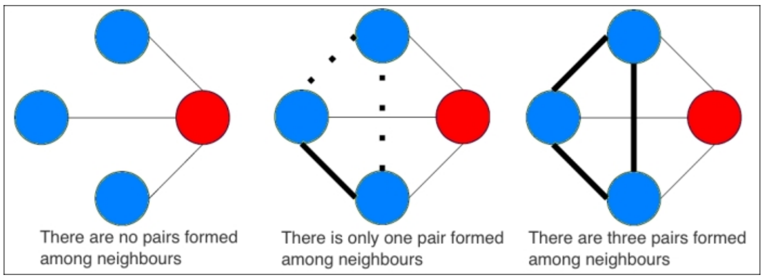

There are no pairs formed among neighbours
There is only one pair formed among neighbours
There are three pairs formed among neighbours

Figure 6.1: Clustering coefficient example [2]

On the left graph the clustering coefficient of a red node is 0, because none of his neighbors share an edge, middle graph has only 1 edge so the coefficient would be $C_{red} = 1/3$, right graph has all 3 edges in the neighborhood of red node so $C_{red} = 3/3 = 1$.

For undirected graphs is it formally defined as:

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}$$

Where N_i is a set of neighbors of a vertex v_i, k_i is a number of neighbors of a vertex v_i. Computing clustering coefficient, if implemented properly, has the complexity of $\mathrm{O}(n^2 \log n)$.

Because VCG graph is bipartite, I need to compute clustering coefficient there too, but for bipartite network this formula would not make much sense as the coefficient would be always 0 because neighbor of a vertex is from other partity and by definition there is no edge between nodes in same partity. For bipartite graphs there exist a redefinition called "Latapy clustering coefficient" [35].

6.1.4 Maximal independent set

Definition 6.1.4. An *independent set* sometimes called *anticlique* is a set of vertices in a graph, no two of which are adjacent.

Definition 6.1.5. *Maximal independent set (MIS)* is an independent set that is not a subset of any other independent set.

The problem of finding maximum independent set is called the *maximum independent set problem* and is an NP-hard optimization problem itself. However its approximation can be computed in $\mathrm{O}(|V|/\log^2|V|)$ [36].

6.1.5 Greedy modularity communities

I used Clauset–Newman–Moore algorithm [37] for finding communities in a graph.

Greedy modularity maximization begins with each node in its own community and joins the pair of communities that most increases modularity until no such pair exists.

The principle of this algorithm is similar to hierarchical agglomeration algorithm for detecting community structure which is faster than many competing algorithms.

Its running time on a network with n vertices and m edges is $\mathrm{O}(md \log n)$ where d is the depth of the dendrogram describing the community structure.

Many real-world networks are sparse and hierarchical, with $m \sim n$ and $d \sim \log n$, in which case the algorithm runs in essentially linear time, $\mathrm{O}(n \log^2 n)$.

6.2 Constructed dataset

Constructing training dataset for learning consists of few steps.

For each instance I ran SatZilla's feature extractor, then ran my extractor and combined the computed features into one sample.

I determined the optimal values for every parameter of MiniSat for the given instance, using the brute-force *grid search.*

Final step of compiling single sample for the dataset was to add the corresponding optimal parameters to the data row of extracted features.

6.3 Pseudo-code of extracting features

Algorithm 3 Extract features

```
    Input: CNF formula $\phi$
    Output: An array of numerical values, the features
function ARRAY_STATS($arr$)
    $min$ = MIN($arr$)
    $max$ = MAX($arr$)
    $vc$ = VARIANCE_COEFFICIENT($arr$)
    $mean$ = MEAN($arr$)
    return {min,max,vc,mean}
end function

function EXTRACT_FEATURES($\phi$)
    $features = \emptyset$
    $VG$ = BUILDVG($\phi$)
    $CG$ = BUILDCG($\phi$)
    $VCG$ = BUILDVCG($\phi$)
    $vg\_ccoef$ = CLUSTERING_COEFFICIENT($VG$)
    $vg\_mis$ = MAXIMAL_INDEPENDENT_SET($VG$)
    $vg\_nrdcy$ = NODE_REDUNDANCY($VG$)
    $vg\_mod\_centr$ = GREEDY_MODULARITY_COMMUNITIES($VG$)
    $cg\_ccoef$ = CLUSTERING_COEFFICIENT($CG$)
    $cg\_mis$ = MAXIMAL_INDEPENDENT_SET($CG$)
    $vcg\_ccoef$ = CLUSTERING_COEFFICIENT($VCG$)
    $vcg\_mis$ = MAXIMAL_INDEPENDENT_SET($VCG$)
    $vcg\_nrdcy$ = NODE_REDUNDANCY($VCG$)
    $vcg\_mod_centr$ = GREEDY_MODULARITY_COMMUNITIE($VCG$)
    $features$ = {ARRAY_STATS($x$)|$\forall x \in$ {$vg\_ccoef, vg\_nrdcy,$
                    $vg\_mod\_centr, cg\_ccoef, vcg\_ccoef, vcg\_nrdcy,$
                    $vcg\_mod_centr$}}
    $features = features \cup \{vg\_mis, cg\_mis, vcg\_mis\}$
    return $features$
end function
```

Function ARRAY_STATS is a helper function that computes statistics from passed input array. Function EXTRACT_FEATURES takes input CNF formula. On lines 11–13 VG, CG, VCG graphs are constructed. Lines 14–22 compute various features on each graph. Line 24 apply ARRAY_STATS on variables which are currently per–node statistics number arrays, from these arrays I compute statistics minimum, maximum, variance coefficient, mean. Line 25 appends features vector by 3 values which numbers.

6.3.1 Algorithm complexity

ARRAY_STATS function is obviously O(n), with the size of input array. Building variable graph BUILDVG is $O(n^2)$, it iterates over every clause and then over every literal in that clause. Building variable clause graph has the same complexity as VG. Building Clause graph is the most expensive task to do, for every pair of clauses, that is $O(n^2)$ operation, it checks for intersection of literals. Intersection of two sets is quadratic in worst case., thus the complexity of BUILDCG is $O(n^4)$.

6.4 Learning

As underlying machine learning technique I have chosen random forest, as this is multi-output regression and also data are from 4 distinctive classes which have different optimal parameter demands, and I believe random forest suits best for this task.

Another alternative was to use artificial neural net, but since I do not have large dataset which size would be roughly in thousands of instances.

6.5 Pseudo-code of learning process

Algorithm 4 Prepare dataset and learning

```
    Input: Set $\Phi$ of CNF formulas
    Output: Model capable of predicting parameters
 1: function COMPILE_DATASET($\Phi$)
 2:     $rows = \emptyset$
 3:     for each $\phi \in \Phi$ do
 4:         $SZ\_feat$ = SATZILLAFEATURES($\phi$)
 5:         $my\_feat$ = EXTRACT_FEATURES($\phi$)
 6:         $features = my\_feat \cup SZ\_feat$
 7:         $targets$ = OPTIMAL_PARAMS($\phi$)      ▷ precomputed from grid-search
 8:         $sample = (features, targets)$                              ▷ pair
 9:         $rows = rows \cup \{sample\}$
10:     end for
11:     return $rows$
12: end function
13:
14: function TRAIN($\Phi$)
15:     $\mathcal{D}$ = COMPILE_DATATSET($\Phi$)
16:     $model$ = RANDOMFORREST($\mathcal{D}$)
17:     return $model$
18: end function
```

Loop on line 3 goes over every instance in training set. Line 4 calls external program which computes features featured in SatZilla. Line 5 calls my extractor defined earlier. On line 6 I combine these two sets of features into one. Line 7 append corresponding optimal parameters which I precomputed, this lookup is $O(1)$ because it is ordered in same order as instances in Φ. Line 8 creates a tuple of features and corresponding targets, this is one row for the dataset. Line 9 appends this row to the end of the dataset.

Chapter **7**

Evaluation

In this final chapter I will present the results achieved, in the form of plots from which it is evident that the tuned parameters outperform MiniSat defaults.

All graphs in this chapter only show pure solving time. Time spent computing features is not included.

All instances are pre-processed by SatELite instance pre-processor [38] which is very fast and the time spent preprocessing can be neglected in any evaluations.

In the following plots there are two columns for each instance next to each other. Blue columns are performances on tuned parameters, green ones on the default MiniSat's parameters. Instances are sorted by number of conflicts yielded by default parameter.

7.1 Performance on training instances

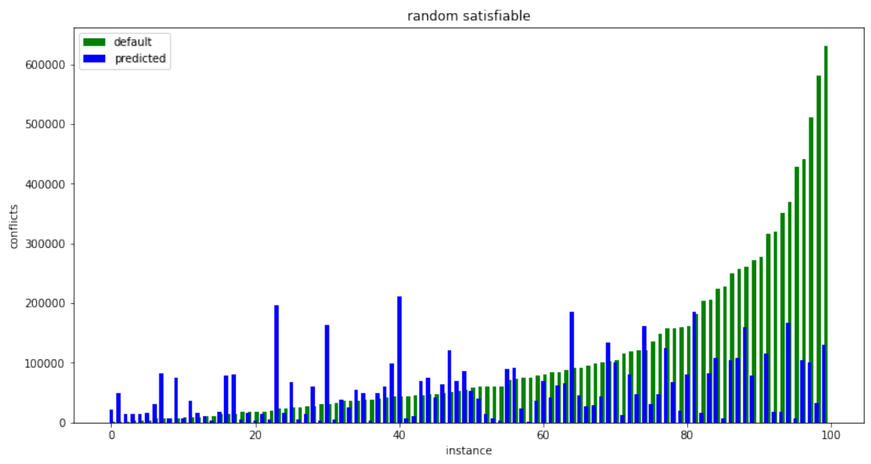


Figure 7.1: 100 training instances, random, satisfiable

Instances used for training are from SATLIB, they have constant number of variables, 250 before preprocessing.

In this plot it can be seen that tuned parameters (blue), are faster for some of the instances but in fact slower for those instances that can be solved very fast with default parameters, those are instances which are solved within single digit number of restarts.

This is probably because the model tends to choose wider restart interval (in comparison to default's value of 2, which is quite low), because it was also trained on the factorization instances, which require less frequent restarts. On those random instances which take considerable time to solve by default parameters, the efficiency rises dramatically, and thus tuned parameters should be used on random instances which have larger number of variables, because for small instances default parameters perform better.

This could be fixed by including random instances of different sizes in the training set, so the model could adapt to the size of the instance better, for example, for small instances restart frequency should be also much smaller.

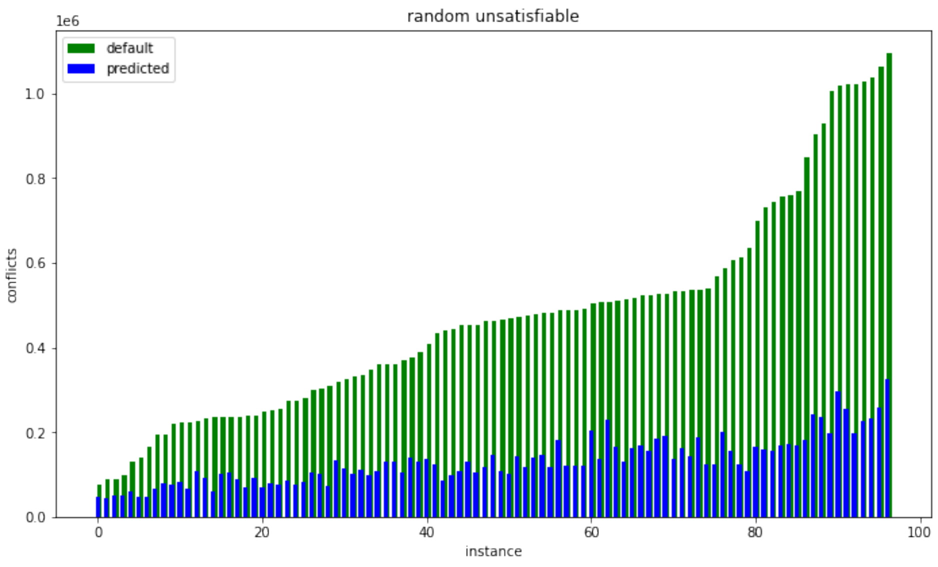


Figure 7.2: 97 training instances, random, unsatisfiable

For every single random unstatisfiable instance from training set (SATLIB) the tuned parameters are much faster.

This plot proves that it is worth tuning solver's parameters in particular for unsatisfiable instances. There is a slight correlation between heights of green and blue bars on the graph, unlike for random satisfiable instances.

The hardest instances are from so called *phase transition* which is a ratio of clauses to variables around value 4.26, so roughly 4x more clauses than variables.

The computation of features is very fast for random instances as they have balanced ratio of clauses and variables.

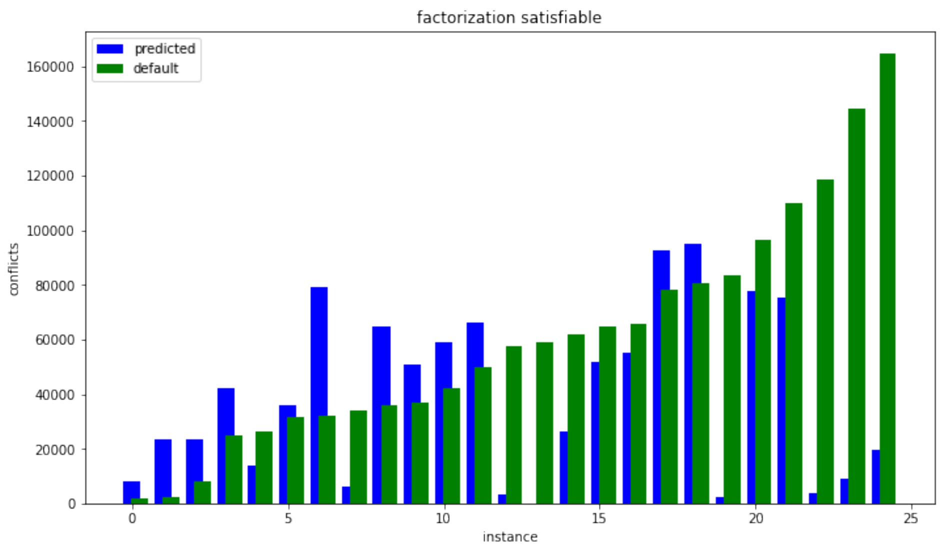


Figure 7.3: 25 training instances, factorization, satisfiable (not prime numbers)

Twelve out of twenty-five satisfiable instances were actually slower with tuned parameters but ten of them were easier instances. This is a bit disturbing result as first sight, my hypothesis is that the cause of it, is lack of easier instances of this type of problem in the training dataset as first half of the graph shows. On the second half of the graph it can be observed that for harder instances, only two instances are slightly slower. I would say, for harder instances these results are positive.

The model does not distinguish well between hard and easier instances, and as a result it is predicting restart frequency parameters similarly for both harder and easier instances.

Another possibility can be that there is no information to be captured from the graph structure about how hard the instance will be.

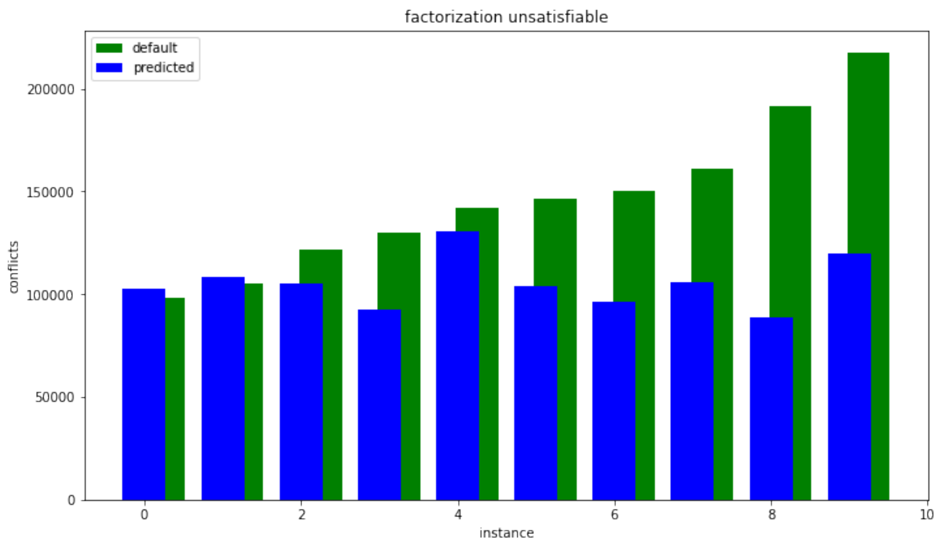


Figure 7.4: 10 training instances, factorization, unsatisfiable (prime numbers)

Majority of eight out of ten instances are favoring tuned parameters, for two easiest instances the default parameters perform better but only by a tiny bit of 500-1000 conflicts less which is small enough count to be neglected.

The improvement is only moderate, nowhere near the improvement observed on random unsatisfiable instances.

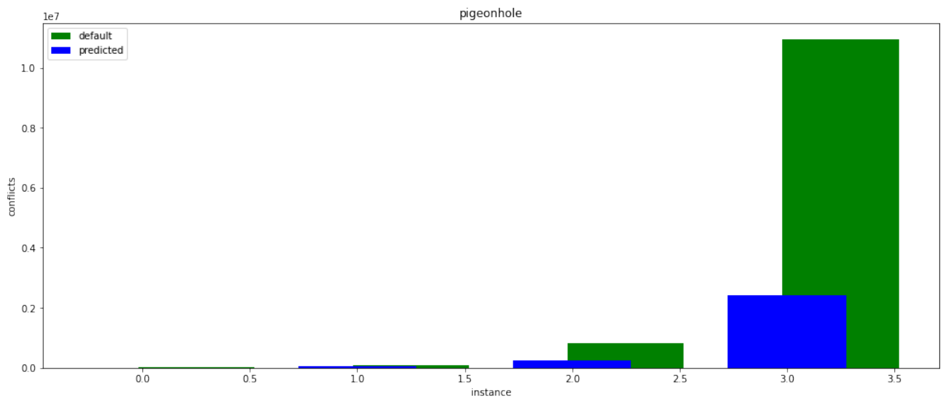


Figure 7.5: Training instances, pigeonhole problem

First two bars are insignificant, but on remaining the big improvement can be seen. Even though the training dataset contained only these four instances of pigeonhole problem (because higher order of this problem is very difficult and it was infeasible to perform gridsearch on many parameters values), the model was able to predict values correctly.

This might mean that the structure of this instance is vastly different from all the other instances from classes.

7.2 Performance on testing instances

As a testing set for random instances I generated instances randomly but with 300 variables in compparison to SATLIB's 250, to observe whether the model will be able to predict values correctly also for instances which are much harder than the ones it was trained on.

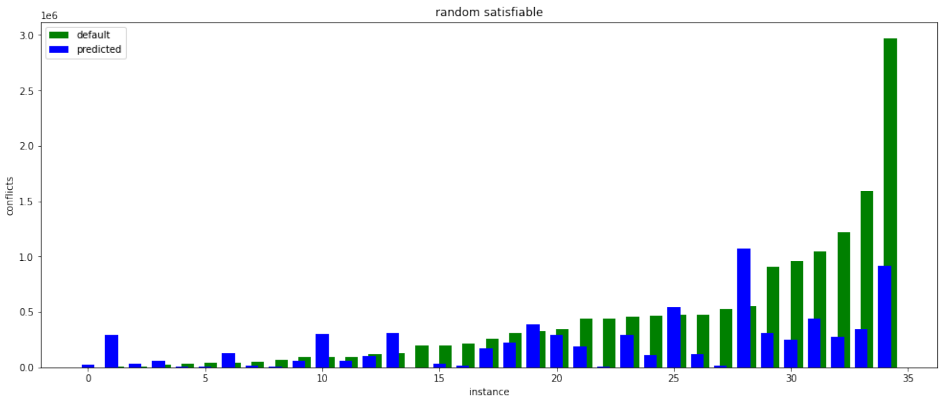


Figure 7.6: 36 testing instances, random, satisfiable

Plot shows very good results, so this verify my hypothesis, that even model trained on smaller instances can perform good on bigger.

For harder instances (second half of the plot), there is only one instance which takes almost twice as much with tuned parameters as with the default ones.

This is probably because the structures of the random instances are homogeneous regardless of their size.

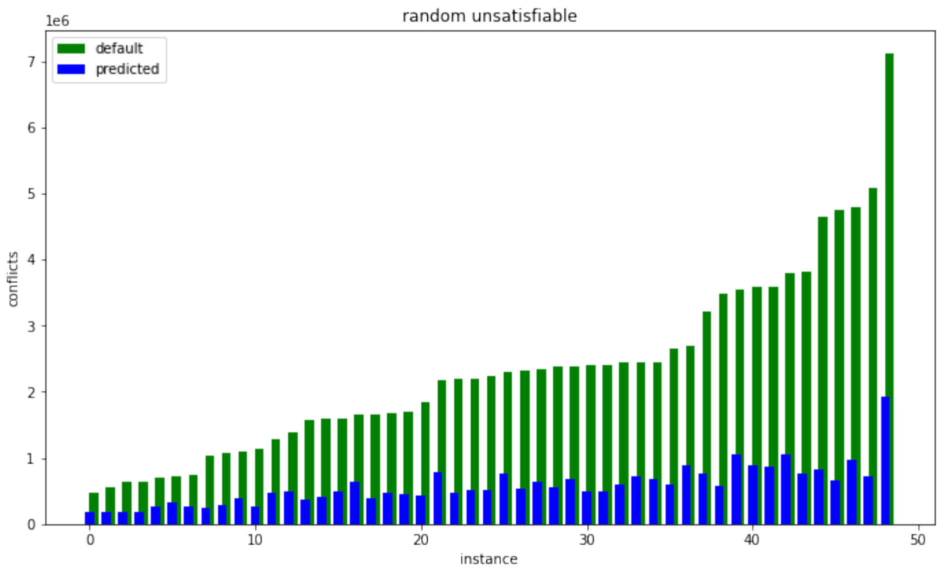


Figure 7.7: 60 testing instances, random, unsatisfiable

Observed results are remarkable, all instances are faster on tuned parameters by at least 2x, on some instances, mostly harder ones, 3x faster.

The key takeaway is that the parameter tuning is very effective way to improve SAT solvers performance on unsatisfiable instances.

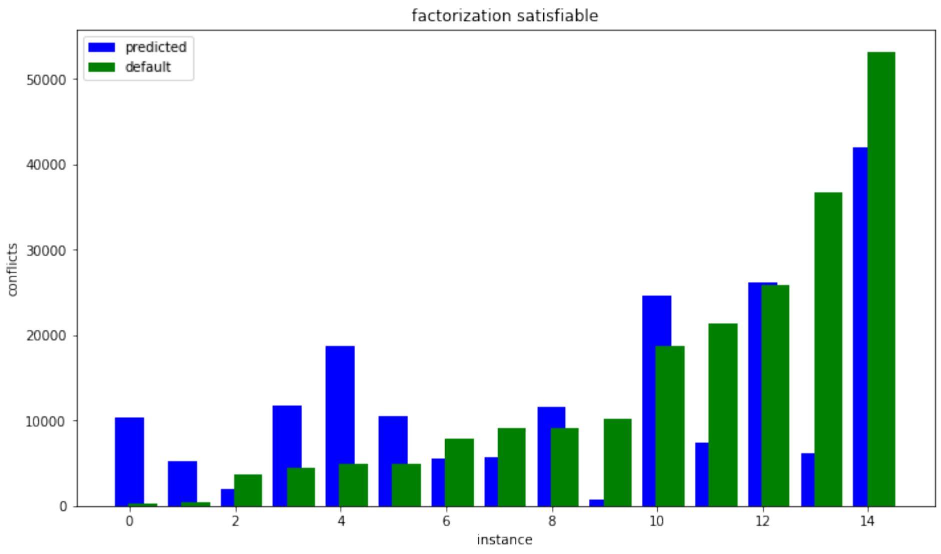


Figure 7.8: 15 testing instances, factorization, satisfiable

Similar results as on training set can be seen here for factorization, satisfiable instances. Performance is better on harder instances, from harder instances only one is outperformed by default settings. Easier instances are solved faster by default settings, most likely because of low base restart interval.

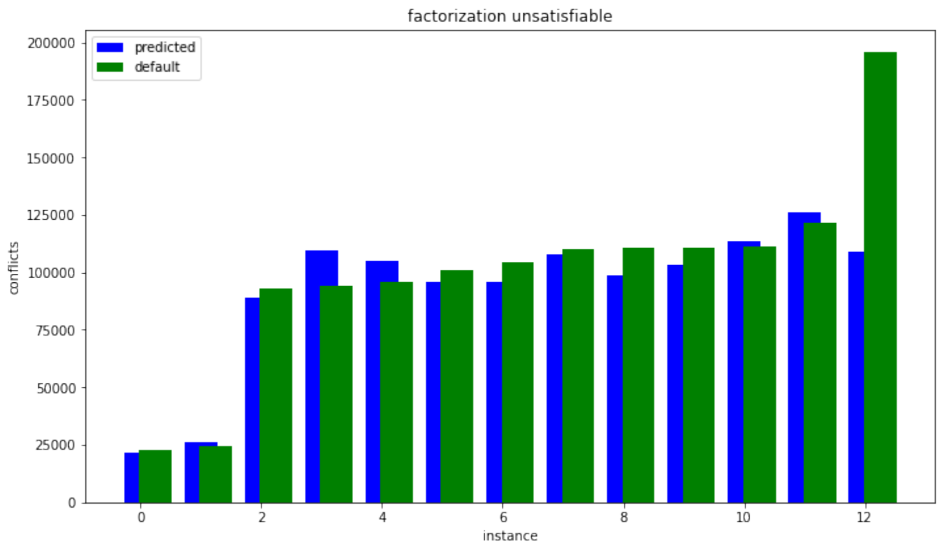


Figure 7.9: 13 testing instances, factorization, unsatisfiable

For testing set I picked 2 easy instances, which can be seen as first 2 bars, then a few average hard instances and one very hard, the last bar.

For easy instances there is no improvement, for medium instances predicted parameters give steadily 100000 conflicts, worth noting is that, the same is for instances in the training set. For hard instance number of conflicts is also close to 100000, and almost 2x speedup can be seen. Tuned parameters perform more less the same as default ones but I hypothesize that, that as hardness of the instances increase the speedup would get more significant with it.

7.3 Planning instances

It is unfortunate that I was unable to train the model for planning problems. This is due to computational burden that I discovered later, while trying to extract features from the planning instances.

Constructing clause graph and computing features on it for planning instances is usually very big due to the nature of planning instances, which have very large number of clauses.

If I were to start over, I would not include clause graph for planning instances, and focus more on graph properties of corresponding VG and VCG graphs.

Conclusion

One of the goals of this thesis was to experimentally show that there is a dependency of SAT solver's parameters on solving time. This was fully illustrated in chapter 5 about the impact of parameters, on set of four, structurally diverse, SAT instances of: 1st random 3SAT instances, 2nd combinatorial problem – the pigeonhole problem, 3rd planning problem and 4th factorization SAT instances.

The dependencies were measured on five of the MiniSat's parameters regarding heuristics (variable decay, clause decay, random variable selection frequency) and regarding restarts (restart interval increase factor, and base restart interval).

Most visually evident dependencies were observed on random SAT instances which exhibited clear dependency on all five parameters, Apart from planning instances, every class showed dependency on clause decaying factor. Unsurprisingly, each of the studied classes were dependent on parameters regarding restart frequency.

Second goal was to explore various input instance features and experiment with automated setting of MiniSat's parameters. This was achieved successfully in chapter 6 about parameter tuning where I provided two pseudo-codes which I also implemented and used to evaluate predicted parameters in chapter 7.

Last goal was to evaluate how predicted parameters perform on both training set and testing set. This was partially successfully done, it is quite a disappointment that I was not able to evaluate the performance on planning instances as they are most connected to the real-life application of SAT solvers, due to their nature of having large number of clauses, which are multiplied by the complexity of building CG and computing features on it. However for other classes of instances I accomplished my goal.

The most positive achievement was tuning parameters for unsatisfiable random SAT instances, where for handful number of instances tested I achieved even 3x speedup.

Overall conclusion of the thesis is that I fulfilled all goals which I set in the introduction chapter.

As a suggestion for a future work I would try to focus more on computing features of VG and VCG and leave CG out as it is very computationally expensive and often causes feature extractor execution time to outweighs actual solving time.

Bibliography

[1] MetaCentrum – National Grid Infrastructure, CESNET department. Available from: `https://www.metacentrum.cz`

[2] Clustering coefficient example [image]. [Cited 2020-25-7]. Available from: `https://www.oreilly.com/library/view/mastering-python-data/9781783988327/ch07s02.html`

[3] Cook, S. A. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, New York, NY, USA: Association for Computing Machinery, 1971, ISBN 9781450374644, p. 151–158, doi:10.1145/800157.805047. Available from: `https://doi.org/10.1145/800157.805047`

[4] Dennis, G.; Chang, F. S.-H.; et al. Modular Verification of Code with SAT. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, New York, NY, USA: Association for Computing Machinery, 2006, ISBN 1595932631, p. 109–120, doi: 10.1145/1146238.1146251. Available from: `https://doi.org/10.1145/1146238.1146251`

[5] Kautz, H.; Selman, B. Planning as Satisfiability. 01 1992, pp. 359–363.

[6] Gupta, A.; Ganai, M. K.; et al. SAT-Based Verification Methods and Applications in Hardware Verification. In *Formal Methods for Hardware Verification*, edited by M. Bernardo; A. Cimatti, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-34305-9, pp. 108–143.

[7] Soos, M.; Nohl, K.; et al. Extending SAT Solvers to Cryptographic Problems. In *Theory and Applications of Satisfiability Testing - SAT 2009*, edited by O. Kullmann, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ISBN 978-3-642-02777-2, pp. 244–257.

[8] Niklas Eén, N. S. Theory and Applications of Satisfiability Testing: 6th International Conference, SAT, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers, chapter An Extensible SAT-solver. *Springer Berlin Heidelberg*, 2004: pp. 502–518.

[9] Newsham, Z.; Lindsay, W.; et al. SATGraf: Visualizing the Evolution of SAT Formula Structure in Solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Springer, Austin, USA: Springer, 62 - 70 2015, doi:10.1007/978-3-319-24318-4_6.

[10] Weber, T. Pigeonhole SAT instance generator [online]. 2007, [Cited 2020-15-6]. Available from: `http://user.it.uu.se/~tjawe125/software/pigeonhole/`

[11] Fernando Pérez, B. G. JuPYteR Notebook, a web-based interactive computational environment for creating Jupyter notebook documents. Available from: `https://jupyter.org/`

[12] Davis, M.; Putnam, H. A computing procedure for quantification theory. In *J. ACM, vol. 7, no. 3*, 1960, pp. 201–215.

[13] Tseitin, G. S. On the Complexity of Derivation in Propositional Calculus. In *Springer Berlin Heidelberg*, 1983, pp. 466–483.

[14] Biere, A.; Biere, A.; et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* NLD: IOS Press, 2009, ISBN 1586039296.

[15] Moskewicz, M. W.; Madigan, C. F.; et al. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[16] Marques Silva, J. P.; Sakallah, K. A. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 220–227.

[17] Liang, J. H. *Machine Learning for SAT Solvers.* Dissertation thesis, 2018, [Cited 2020-19-6].

[18] Harder, J. Development of SAT solver. 2014, [Cited 2020-2-7]. Available from: `https://www.isp.uni-luebeck.de/sites/default/files/jannis-harder-bsc-thesis.pdf`

[19] Bayardo, R. J.; Schrag, R. C. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, AAAI Press, 1997, ISBN 0262510952, p. 203–208.

[20] Moskewicz, M. W.; Madigan, C. F.; et al. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[21] Beanie, P.; Kautz, H.; et al. Understanding the Power of Clause Learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, p. 1194–1201.

[22] Gomes, C. P.; Selman, B.; et al. Boosting Combinatorial Search through Randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, USA: American Association for Artificial Intelligence, 1998, ISBN 0262510987, p. 431–437.

[23] Pipatsrisawat, K.; Darwiche, A. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2007*, edited by J. Marques-Silva; K. A. Sakallah, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN 978-3-540-72788-0, pp. 294–299.

[24] D'Ippolito, N.; Frias, M.; et al. Alloy+HotCore: A Fast Approximation to Unsat Core. 02 2010, pp. 160–173, doi:10.1007/978-3-642-11811-1_13.

[25] Xu, L.; Hutter, F.; et al. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.*, volume 32, 2008: pp. 565–606.

[26] SAT competition website. Available from: `http://www.satcompetition.org/`

[27] Ganesh, V.; Singh, R.; et al. AvatarSAT: An Auto-tuning Boolean SAT Solver. 08 2009.

[28] Lourenço, H.; Martin, O.; et al. *Iterated Local Search: Framework and Applications*, volume 146. 09 2010, ISBN 978-1-4419-1663-1, pp. 363–397, doi:10.1007/978-1-4419-1665-5_12.

[29] Pintjuk, D. Boosting SAT-solver Performance on FACT Instances with Automatic Parameter Tuning. 2015, [Cited 2020-15-7]. Available from: `https://www.diva-portal.org/smash/get/diva2:811289/FULLTEXT01.pdf`

[30] Sinz, C.; Dieringer, E.-M. DPvis – A Tool to Visualize the Structure of SAT Instances. In *Theory and Applications of Satisfiability Testing*, edited by F. Bacchus; T. Walsh, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN 978-3-540-31679-4, pp. 257–268.

[31] Ansótegui, C.; Giráldez-Cru, J.; et al. The Community Structure of SAT Formulas. In *Theory and Applications of Satisfiability Testing – SAT 2012*, edited by A. Cimatti; R. Sebastiani, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31612-8, pp. 410–423.

[32] Haken, A. *The Intractability of Resolution (Complexity)*. Dissertation thesis, USA, 1984, aAI8422073.

[33] Horn, A. On Sentences Which are True of Direct Unions of Algebras. *J. Symbolic Logic*, volume 16, no. 1, 03 1951: pp. 14–21. Available from: `https://projecteuclid.org:443/euclid.jsl/1183731038`

[34] Bennaceur, H. A comparison between SAT and CSP techniques. *Constraints*, volume 9, 04 2004: pp. 123–138, doi:10.1023/B:CONS.0000024048.03454.c0.

[35] Latapy, M.; Magnien, C.; et al. Basic Notions for the Analysis of Large Two-mode Networks. *Social Networks*, volume 30, 01 2008: pp. 31–48, doi:10.1016/j.socnet.2007.04.006.

[36] Boppana, R.; Halldórsson, M. Approximating Maximum Independent Sets by Excluding Subgraphs. 01 2006, pp. 13–25, doi:10.1007/3-540-52846-6_74.

[37] Clauset, A.; Newman, M.; et al. Finding community structure in very large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, volume 70, 01 2005: p. 066111, doi:10.1103/PhysRevE.70.066111.

[38] Eén, N.; Biere, A. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Theory and Applications of Satisfiability Testing*, edited by F. Bacchus; T. Walsh, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 61–75.

[39] van Rossum, G. Python, programming language. Available from: `https://www.python.org/`

[40] Oliphant, T. NumPy, a library for the Python programming language. Available from: `https://www.python.org/`

[41] Travis Oliphant, E. J., Pearu Peterson. SciPy, an open-source library used for scientific computing and technical computing. Available from: `https://www.scipy.org/`

[42] McKinney, W. Pandas, library for for data manipulation and analysis. Available from: `https://pandas.pydata.org/`

[43] Cournapeau, D. Scikit-learn, a free machine learning library. Available from: `https://scikit-learn.org/stable/`

[44] Hunter, J. D. Matplotlib, a plotting library for Python and its extension NumPy. Available from: `https://matplotlib.org/`

[45] Aric Hagberg, D. S., Pieter Swart. NetrokX, library for studying graphs and networks. Available from: `https://networkx.github.io/`

Appendix A

Acronyms

CDCL Conflict-driven clause-learning

CG Clause graph

CNF Conjunctive normal form

CSP Constraint satisfaction problem

DP Davis–Putnam algorithm

DPLL Davis–Putnam–Logemann–Loveland algorithm

SAT Boolean satisfiability problem

SVM Support vector machine

VCG Variable-clause graph

VG Variable graph

VSIDS Variable state independent decaying sum

Appendix B

Software description

Software source codes accompanying this thesis which I implemented during writing this thesis is experimental, meaning that the main purpose was to produce necessary graphs, plots, statistics etc.

It is not stand-alone application ready to be used in practice as that was not part of my assignment.

Source code is written in Python, specifically across multiple Jupyter notebooks.

The following section I will mention frameworks, languages and tools which were used in this thesis.

B.1 Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

For implementation I chose programming language Python [39], because I am most familiar with it and it libraries, and also because there was expected a lot of interactive data science, scientific computing and manipulation with the data which is comfortably done in Jupyter Notebook.

B.2 JuPYteR Notebook

The JuPYteR Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. [11]

All the code I wrote is in Jupyter notebooks, it is easy to execute, easy to make changes and immediately see the results without need for switching between IDE and multiple terminals.

B.3 Numpy and Scipy

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. [40]

SciPy is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering. [41]

I used both libraries when manipulating data, preparing data for plotting, generating instances SAT for testing, and much more.

B.4 Pandas

In computer programming, pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license. [42]

Pandas was used for concatenating multiple tabular data, storing and sorting results.

B.5 Scikit-learn

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. [43]

I used scikit's implementation of Random forest algorithm for training the predictive model. I also used it for generating parameter's grid for grid-search.

B.6 Matplotlib

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+. [44]

Every data plot in this thesis was rendered by matplotlib.

B.7 NetworkX library

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. [45]

NetworkX's graph implementation was used for constructing VG, CG, and VCG. Four of graph algorithms were used to compute SAT instance's features.

Appendix C

Contents of enclosed CD

readme.txt	detailed CD contents description
src	source codes
text_src	thesis latex source codes and images
2020_Beskyd_Filip_thesis.pdf	thesis PDF