Project

Search Space Partitioning with Lookahead

Ahmed Irfan

12 November 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Künstliche Intelligenz
Professur für Knowledge Representation and Reasoning

Supervised by:
Prof. Dr. rer. nat. Steffen Hölldobler
Dipl.-Inf. Norbert Manthey

Ahmed Irfan
Search Space Partitioning with Lookahead
Project, Fakultät für Informatik
Technische Universität Dresden, November 2012

Abstract. With technological shift from single core CPU to multi-core CPUs, parallel SAT solving requires more attention than before. Portfolio and search space partitioning are the two main approaches in parallel SAT solving. Search space partitioning is further divided into plain search space partitioning and iterative search space partitioning. In this work, I discuss the lookahead techniques for iterative search space partitioning. I extend the work of [HM12b] by implementing the lookahead techniques for partitioning and show that the lookahead for iterative partitioning is better approach than the VSIDS. The iterative partitioning with lookahead can solve application instances faster and 20 additional instances are solved in a given time limit of 3600 seconds. The results are shown with the help of traditional statistical measures as well as with a novel ranking method [VG11].

Contents

1. Introduction

The satisfiability problem (often abbreviated as SAT problem) is one of the most researched NP-complete problem in theoretical computer science. The importance of the satisfiability problem can be realized by the fact that it is the best known example of NP-complete problem [Coo71] and from the still open *P vs NP* problem from the Millennium Prize Problems. The SAT problem in principle is hard to solve, yet modern state-of-the-art SAT solvers solve real life instances (problems translated to the SAT problem) quite efficiently. For example, SAT solvers are used in planning [KS92], scheduling [GHM+12] [CP89], vehicle routing [Goe10], hardware and software verification [BCCZ99] [DKW08], bioinformatics [LMS06] and configuration [ABL+10].

Over the last two decades, a lot of improvements have been made to solve the SAT problem and can be seen in the yearly SAT competitions. This success was more evident in the sequential SAT solvers (using one CPU) and quite less in parallel ones (using more than one CPU). A number of attempts have been made for parallel SAT solving; to have an overview of these approaches, see [HMN+11]. With the technological shift from single core CPU to muti-core CPUs, parallel SAT solving requires more attention than before. Among the approaches for parallel SAT solving so far, the most dominant approach is to run several different SAT solvers in parallel for the same problem and wait for the first answer [Rou12]. This approach is called *portfolio* approach. Another promising approach to parallel SAT solving is *iterative search space partitioning* [HJN10]. The idea is to partition the original SAT problem into sub-problems and try to solve them in parallel with some sort of limit on the solve time. If the sub-problem is not solved in that limit, then it is partitioned into more sub-problems and each sub-problem is tried to solve (this makes the approach iterative). Based on the idea of iterative search space partitioning, an effort has been made in [HM12a] and the parallel SAT solver *splitter* [HM12b] was implemented which used variable state independent decay sum (VSIDS) heuristic for partitioning.

Apart from the main results of [HJN10], lookahead is also mentioned as good heuristic to partition the search space and suggests further investigation. This gives the motivation of this work to investigate lookahead techniques for iterative search space partitioning by extending the solver *splitter*. The particular interest is to find if *splitter* with lookahead is faster (better median runtime) and better (solves more instances) than the *splitter* with VSIDS. The idea of using lookahead for partitioning the search space is the basis of the parallel SAT solver *cube & conquer* [HKWB12] and also its successor *concurrent cube & conquer* [vdTHB12]. The SAT solver *cube & conquer* is able to solve some hard instances much faster than other sequential or parallel SAT solvers and its successor *concurrent cube & conquer* performed very well in the SAT challenge 2012. The major difference between this work and *cube & conquer* approach is that this work investigates the lookahead techniques for iterative search space partitioning (in simple words, partitioning is done more than once) while *cube & conquer* uses plain search space partitioning (partitioning is done once).

A recent paper [HW12] has put together seven challenges in parallel SAT solving, which need the attention of the researchers for the same. This work can also be seen as an attempt to solve one part of the challenge $\# 2$ which is about designing a dynamic search space partitioning that is efficiently computable and results in partitions enabling solvers

to consistently outperform currently known methods.

The author of [HJN10] has compared decision heuristics VSIDS with lookahead and concluded that VSIDS for search space partitioning is better than lookahead. He has used simple lookahead decision heuristic which counts the number of assigned literals, with detection of failed literals as the only local learning options (discussed in Section 3.3). I believe that the performance of splitter with lookahead could be improved by including some other decision heuristics and local learning options. It is shown in Section 5.3 that splitter with lookahead meets the expectations and surprisingly it also outperforms the splitter with VSIDS.

The structure of this report is as follows. Next section (Section 2) gives required background of propositional logic. Section 3 briefly introduces the dominant approaches in SAT solving and also gives a detailed account on the lookahead techniques. Section 4 talks about search space partitioning where scattering is also introduced. Section 5 discusses the specification of the splitter using lookahead techniques, some of the parameter tuning tests performed on the training set and evaluations of the final tests on the full test set. In the end, conclusion with future direction of this work is given in Section 6.

2. Preliminaries and Notation

In this section, some basic notions of propositional logic and the notation used in this document are given.

2.1. Propositional Logic

As most SAT solvers take formulas only in conjunctive normal form (CNF), the description will be limited to this form. Before defining CNF formula (formula in CNF), the syntax of propositional logic is given.

Definition 2.1. The set of *atomic propositions* AP is a countably infinite set of symbols. Atomic propositions are also called *atoms* or *propositional variables* or simply *variables*.

Propositional variables can be assigned the truth value either *true* or *false*, represented by $\top$ and $\bot$ respectively. They can be combined with connectives *negation*, *disjunction* and *conjunction*. These connectives are denoted by $\neg, \vee, \wedge$ respectively, where $\neg$ is unary and $\vee, \wedge$ are binary connectives.

Definition 2.2. A *literal* L is a propositional variable A or its negation $\neg A$.

Definition 2.3. A *polarity* of a literal L is said to be *positive* if $L = A$, *negative* if $L = \neg A$ where $A \in AP$.

Definition 2.4. Given a literal L, its *complement* $\overline{L}$ is defined as:

$$\overline{L} := \begin{cases} \neg A & \text{, if } L = A \in AP \\ A & \text{, if } L = \neg A, \text{ where } A \in AP \end{cases}$$

A pair L, $\overline{L}$ of literals is said to be complementary.

Definition 2.5. A *clause* is a disjunction of literals. It is represented by a finite set of literals. A clause is a valid clause if it contains a complementary pair of literals.

Remark. For simplicity, valid clauses are not considered, because any truth assignment would satisfy the clause (satisfiability of clause is defined later).

Definition 2.6. A *CNF Formula* F is conjunction of clauses and is represented by a set of clauses.

Clauses can be categorized on the basis of size, which is defined as followed:

Definition 2.7. The *size* of a clause C, denoted by $|C|$ is defined as the number of literals in C. A clause C is called *unit clause*, *binary clause*, *ternary clause* iff $|C| = 1$, $|C| = 2$, $|C| = 3$, respectively.

Definition 2.8. A *cube* is a conjunction of literals. It is represented as a set of unit clauses which is a CNF formula containing only unit clauses.

To define what the SAT problem is, some important definitions like partial assignment, reduct and some other functions are required.

Definition 2.9. Given a set S of literals, the *complement* of S, represented by $\overline{S}$, is defined as:

$$\overline{S} = \{\overline{L} \mid L \in S\}$$

Definition 2.10. Given a literal L, a function $atom$ returns its underlying propositional variable, i.e.

$$atom(L) = \begin{cases} A & \text{, if } L = A \in AP \\ A & \text{, if } L = \neg A \text{, where } A \in AP \end{cases}$$

The function $atom$ can be overloaded to clauses, i.e. given a clause C, $atom$ will a return set of propositional variables

$$atom(C) = \begin{cases} \emptyset & \text{, if } C = \{\,\} \\ atom(C') \cup \{atom(L)\} & \text{, if } C = C' \cup \{L\} \end{cases}$$

Likewise, overloading of $atom$ for CNF formula, i.e. given a CNF formula F then $atom$ returns the set of atoms present in the formula

$$atom(F) = \begin{cases} \emptyset & \text{, if } F = \{\,\} \\ atom(F') \cup atom(C) & \text{, if } F = F' \cup \{C\} \end{cases}$$

Definition 2.11. A *partial assignment* is a sequence of literals which does not contain complementary pairs.

Definition 2.12. Given a clause C and a partial assignment J, let $s(J)$ be the set of all the elements in J. Then, the *reduct* of C w.r.t. J, denoted $C|_J$, is defined as:

$$C|_J = \begin{cases} \top & \text{, if } C \cap s(J) \neq \emptyset \\ C \setminus \overline{s(J)} & \text{, otherwise} \end{cases}$$

Definition 2.13. A partial assignment J *satisfies* a clause C, in symbols $J \models C$, iff $C|_J = \top$.

Definition 2.14. Given a partial assignment J and a CNF formula F, the *reduct* $F|_J$ of F w.r.t. J is the CNF formula:

$$F|_J = \{C|_J \mid C \in F \text{ and } C|_J \neq \top\}$$

The next example clarifies the notion of reduct for a CNF formula.

Example 2.15. Let $F = \{\{1, \neg 2\}, \{3\}, \{4, \neg 5\}\}$, and $J = (1, \neg 3, 5)$. Then $F|_J = \{\{\,\}, \{4\}\}$.

Definition 2.16. Let F be a CNF formula and J be a partial assignment. Then J is a *model* for F, in symbols $J \models F$, iff $F|_J = \{\,\}$. J *falsifies* F, in symbols $J \not\models F$, iff $\{\,\} \in F|_J$. In the latter case, J is called *conflict* for F.

Lemma 2.17. *Let F be a CNF formula, and J a partial assignment. Then $J \models F$ iff J satisfies every clause in F.*

Next is the definition of the SAT problem and after that the notions of semantic consequence, semantic equivalence and resolvent are given.

Definition 2.18. Given a CNF formula F, *propositional satisfiability problem* or briefly *SAT* is the problem to decide whether F is satisfiable, i.e. if there exist a partial assignment for F such that it satisfies F.

Definition 2.19. Let F, G be CNF formulas. Then G is a *semantic consequence* of F, denoted as $F \models G$, iff for every partial assignment J:

$$J \models F \Rightarrow J \models G$$

Definition 2.20. Let F, G be CNF formulas. Then F and G are *semantically equivalent*, in symbols $F \equiv G$, iff:

$$F \models G \text{ and } G \models F$$

Clauses are represented as sets which means clauses do not have duplicate literals. No duplicate literals in a clause is an assumption for the following definition.

Definition 2.21. Let C_1 be the clause containing literal L and C_2 be the clause containing literal $\overline{L}$, then the *(propositional) resolvent* of C_1 and C_2 with respect to L is defined as:

$$\{C_1 \setminus \{L\}\} \cup \{C_2 \setminus \{\overline{L}\}\}$$

A clause C is said to be resolvent to C_1 and C_2 iff there exists a literal L such that C is the resolvent of C_1 and C_2 with respect to L.

2.2. Notation

It is helpful to fix the notation for the rest of the document, which is:

- F and F_i denote CNF formulas, where $i \in \mathbb{N}$
- C and C_i denote clauses, where $i \in \mathbb{N}$
- A and A_i denote variables, where $i \in \mathbb{N}$
- L and L_i denote literals, where $i \in \mathbb{N}$
- J and J_i denote partial interpretations, where $i \in \mathbb{N}$

I will use above mentioned notion through out the document.

3. SAT Solving

This section discusses the techniques used by most of the current state-of-the-art SAT solvers for the SAT problem. These techniques can be better explained by an abstract reduction system. An abstract reduction system is defined as:

Definition 3.1. Let R be a set and $\rightarrow$ be a binary relation $R \times R$, then the pair $(R, \rightarrow)$ is defined to be an *abstract reduction system* (ARS). The binary relation $\rightarrow$ is called reduction and every pair $(x, y) \in \rightarrow$ is written as $x \rightarrow y$.
$\rightarrow^+$ represent the transitive closure of $\rightarrow$.
$\rightarrow^*$ represent the reflexive and transitive closure of $\rightarrow$.

3.1. Davis Putnam Logemann Loveland

The *Davis Putnam Logemann Loveland* (DPLL) algorithm [DLL62] is the basis for most of the current state-of-the-art SAT solvers. It is an improvement to the *Davis Putnam* (DP) algorithm [DP60]. The DPLL ARS is defined as:

Definition 3.2. Let $CNFF$ be a set of CNF-formulas and PA be a set of partial assignments over the set of propositional variables AP, then

$$R \subseteq (CNFF \times PA) \cup \{\mathsf{SAT}, \mathsf{UNSAT}\}$$

For better representation, $(F, J) \in R$ is written as $F :: J$. The evaluation of $F :: J$ is the reduct $F|_J$. The reduction rules are given in Figure 1. The literal with a dot on top i.e. $\dot{L}$ represents a decision literal, otherwise a propagated literal. P contains only propagated literals.

The definition of $level$ used in DPLL ARS is as follows:

Definition 3.3. The $level$ of J, in symbols $level(J)$, is defined as the total number of decisions literals in J.

The reduction rules $\rightsquigarrow_{\mathsf{SAT}}$ and $\rightsquigarrow_{\mathsf{UNSAT}}$ are simple rules for satisfiable and unsatisfiable results respectively. They are also the termination rules which stop the application of any further reduction rules. The reduction rule $\rightsquigarrow_{\mathsf{DECIDE}}$ chooses a free variable and its polarity in $F :: J$ and concatenates to J. The definition of free variable is:

Definition 3.4. A variable $A \in atom(F)$ is a *free variable* iff $A \in atom(F|_J)$, otherwise it is said to be *assigned*.

$F :: J$	$\rightsquigarrow_{\mathsf{SAT}}$	SAT	iff	$F\|_J = \emptyset$
$F :: J$	$\rightsquigarrow_{\mathsf{UNSAT}}$	UNSAT	iff	$\{\ \} \in F\|_J$ and $level(J) = 0$
$F :: J$	$\rightsquigarrow_{\mathsf{DECIDE}}$	$F :: J, \dot{L}$	iff	$L \in atom(F\|_J) \cup \overline{atom(F\|_J)}$
$F :: J, \dot{L}, P$	$\rightsquigarrow_{\mathsf{NB}}$	$F :: J, \overline{L}$	iff	$\{\ \} \in F\|_{J,\dot{L},P}$
$F :: J$	$\rightsquigarrow_{\mathsf{UNIT}}$	$F :: J, L$	iff	$\{L\} \in F\|_J$

Figure 1: DPLL ARS

The reduction rule $\rightsquigarrow_{\mathsf{NB}}$, where NB stands for naive backtrack, removes the last decision literal and the following propagated literals from J when a conflict (an empty clause) occurs in the reduct. The heart of the DPLL ARS is the $\rightsquigarrow_{\mathsf{UNIT}}$ reduction rule. It concatenates the literals in unit clauses of $F|_J$, i.e. $C = \{L\} \in F$ to $J := J, L$ and these literals are called *propagated literals*. The unit rule is also called *unit propagation*. Here is an example to explain how DPLL ARS works.

Example 3.5. Let $F = \{\{1, 2\}, \{2, \overline{3}\}, \{\overline{2}, \overline{3}, 4\}, \{\overline{1}, 3\}, \{\overline{4}\}\}$, then the execution in DPLL ARS is:

$$
\begin{array}{lll}
 & F :: () & \\
\rightsquigarrow_{\mathsf{UNIT}} & F :: (4) & F|_{(4)} = \{\{1, 2\}, \{2, \overline{3}\}, \{\overline{2}, \overline{3}\}, \{\overline{1}, 3\}\} \\
\rightsquigarrow_{\mathsf{DECIDE}} & F :: (4, \dot{1}) & F|_{(4,\dot{1})} = \{\{2, \overline{3}\}, \{\overline{2}, \overline{3}\}, \{3\}\} \\
\rightsquigarrow_{\mathsf{UNIT}} & F :: (4, \dot{1}, 3) & F|_{(4,\dot{1},3)} = \{\{2\}, \{\overline{2}\}\} \\
\rightsquigarrow_{\mathsf{UNIT}} & F :: (4, \dot{1}, 3, 2) & F|_{(4,\dot{1},3,2)} = \{\{\,\}\} \\
\rightsquigarrow_{\mathsf{NB}} & F :: (4, \overline{1}) & F|_{(4,\overline{1})} = \{\{2\}, \{2, \overline{3}\}, \{\overline{2}, \overline{3}\}\} \\
\rightsquigarrow_{\mathsf{UNIT}} & F :: (4, \overline{1}, 2) & F|_{(4,\overline{1},2)} = \{\{\overline{3}\}\} \\
\rightsquigarrow_{\mathsf{UNIT}} & F :: (4, \overline{1}, 2, \overline{3}) & F|_{(4,\overline{1},2,\overline{3})} = \{\,\} \\
\rightsquigarrow_{\mathsf{SAT}} & \mathsf{SAT} &
\end{array}
$$

The execution of DPLL ARS starts from $F :: ()$ (with empty partial assignment). The unit rule is applied because the unit clause $\{4\}$ in $F|_{()}$. No other rule applies except decide rule. The decide rule chooses the literal 1 as a decision. The unit rule is twice because of the unit clause $\{3\}$ in $F|_{(4,\dot{1})}$ and $\{2\}$ in $F|_{(4,\dot{1},3)}$. This leads to an empty clause in $F|_{(4,\dot{1},3,2)}$. The rule naive backtrack is applied to correct the last decision and change the last decision literal to propagated literal. A clause becomes a unit now in $F|_{(4,\overline{1})}$. Applying the unit rule results in $F|_{(4,\overline{1},2)}$ and another unit clause. The unit rule is applied again which returns $F|_{(4,\overline{1},2,\overline{3})}$. Now there are no clauses left in $F|_{(4,\overline{1},2,\overline{3})}$ and the SAT rule is applicable. The final result is SAT, i.e. the given formula F is satisfiable.

3.2. Conflict-Driven Clause Learning

The conflict-driven clause learning (CDCL) algorithm [SS96] is based on the DPLL algorithm. The CDCL algorithm gives the power to perform non-chronological backtracking by adding clauses to the CNF-Formula which are semantic consequences of the CNF-Formula. DPLL-CDCL ARS is obtained by removing the reduction rule $\rightsquigarrow_{\mathsf{NB}}$ from the DPLL ARS and adding the reduction rule $\rightsquigarrow_{\mathsf{CDBL}}$. To explain the reduction rule $\rightsquigarrow_{\mathsf{CDBL}}$, the notion of *relevant clause* and *linear resolution derivation* is required.

Definition 3.6. A clause C is *relevant* in F w.r.t J iff $C \in F$ and there exists a partial assignment J_1, L, J_2 such that

1. $J_1, L, J_2 = J$ and
2. $C|_{J_1} = \{L\}$

The set $relevant(F :: J) = \{C \in F \mid C \text{ is relevant in } F :: J\}$ is called the *set of relevant clauses* of F w.r.t. the partial assignment J.

Definition 3.7. Given C and F, a *linear resolution derivation* from C w.r.t. F is a sequence $S = (C_i \mid i \geq 0)$ of clauses defined inductively as follows:

- $C_0 = C$ and
- C_i is the resolvent of C_{i-1} and for some clause $E \in F$

If S is finite and C_n is the last element of the sequence, then it is called a *linear resolution derivation* from C to C_n w.r.t. F.

Definition 3.8. The *conflict driven backtrack learning* (CDBL) reduction rule $\rightsquigarrow_{\mathsf{CDBL}}$ is:
$F :: J_1, \dot{L}, J_2 \rightsquigarrow_{\mathsf{CDBL}} F \cup \{C_1\} :: J_1, L_1$
iff there exists $C \in F$ such that $C|_{J_1,\dot{L},J_2} = \{\,\}$ and there is a linear resolution derivation from C to C_1 w.r.t. $relevant(F :: J_1, \dot{L}, J_2)$ and $C_1|_{J_1} = \{L_1\}$
The clause C_1 is called *learnt clause*.

The reduction rule $\rightsquigarrow_{\mathsf{DECIDE}}$ is an important reduction rule in DPLL ARS, see Figure 1. Many heuristics have been explored in the last decade, but the one that stands out among these in the CDCL SAT solvers is the *variable state independent decay sum* (VSIDS) decision heuristic. The VSIDS decision heuristic was presented in the Chaff paper [MMZ$^+$01]. The idea of the VSIDS decision heuristic is to assign *activity* score to each variable based on its frequency in the formula. This activity score of a variable is increased with usage of the variable in the linear resolution derivation of the learnt clause by the $\rightsquigarrow_{\mathsf{CDBL}}$ reduction rule. Then the VSIDS picks the variable with the highest activity score. Due to the use of acitivity score, the VSIDS decision heuristic is also called *activity heuristic*.

3.3. Lookahead

The lookahead SAT solvers are also based on the DPLL algorithm. The major difference of lookahead SAT solver from CDCL SAT solvers is that they are heavily driven by expensive heuristics. Before going deep into the lookahead procedure, first look at some basic definitions.

Definition 3.9. The *lookahead* on $F :: J$ with respect to L, in symbols $lookahead(F :: J, \dot{L})$, is defined as:

$$lookahead(F :: J, \dot{L}) = \begin{cases} \top & \text{, if } \{\,\} \in F|_{J,\dot{L},P} \text{ and } F :: J, L \rightsquigarrow^*_{\mathsf{UNIT}} F :: J, \dot{L}, P \\ \bot & \text{, if otherwise} \end{cases}$$

Definition 3.10. A literal L is called a *failed literal* in $F :: J$ if $lookahead(F :: J, \dot{L}) = \top$.

The main idea of lookahead SAT solvers is to perform $lookahead$ on some interesting free variables, find failed literals and choose a literal according to heuristics (explained later in this section). This process can be called as $lookaheadDecide$. The procedure $lookaheadDecide$ is performed every time before actually deciding a literal. To utilize the time efficiently, it performs some reasoning techniques (also explained later in this section).

$F :: J$	$\rightsquigarrow_{\mathsf{SAT}}$	$F ::$ SAT	iff	$F\|_J = \{\}$
$F :: J$	$\rightsquigarrow_{\mathsf{UNSAT}}$	$F ::$ UNSAT	iff	$\{\} \in F\|_J$ and $level(J) = 0$
$F :: J$	$\rightsquigarrow_{\mathsf{DECIDE}}$	$F :: J, \dot{L}$	iff	$L \in atom(F\|_J) \cup \overline{atom(F\|_J)}$
$F :: J$	$\rightsquigarrow_{\mathsf{UNIT}}$	$F :: J, L$	iff	$\{L\} \in F\|_J$
$F^0, \ldots, F^l :: J, \dot{L}, P$	$\rightsquigarrow_{\mathsf{NB}}$	$F^0, \ldots, F^{l-1} :: J, \overline{L}$	iff	$\{\} \in (F^0 \cup \ldots \cup F^l)\|_{J,\dot{L},P}$
$F :: J, \dot{L}, P$	$\rightsquigarrow_{\mathsf{LA\_BACK}}$	$F :: J$	iff	$\{\} \notin F\|_{J,\dot{L},P}$
$F :: J$	$\rightsquigarrow_{\mathsf{LR}}$	$F, F^l :: J$	iff	$F\|_J \models F^l\|_J$ and $level(J) = l$

Figure 2: Lookahead ARS

Definition 3.11. To explain the DPLL-Lookahead in terms of ARS, some modifications in the ARS $(R, \rightsquigarrow)$ are required. Now each clause of F has a number in the superscript, which denotes its level. The reduction rules of *DPLL-Lookahead ARS* are shown in Figure 2.

The reduction rule $\rightsquigarrow_{\mathsf{LR}}$, where LR denotes for *local reasoning*, adds clauses which are locally valid at level l (means they are valid as long as $level(J) \geq l$) and these clauses are called *locally learnt clauses*. All these locally learnt clauses at level l need to be removed while applying the reduction rule $\rightsquigarrow_{\mathsf{NB}}$ at level l. The reduction rule $\rightsquigarrow_{\mathsf{NB}}$ finds the failed literal L. Another reduction rule which is not in DPLL ARS is $\rightsquigarrow_{\mathsf{LA\_BACK}}$, which is only used in the $lookaheadDecide$ phase while computing the decision heuristic. The reduction rule $\rightsquigarrow_{\mathsf{LA\_BACK}}$ gives the power to go back one level, same as $\rightsquigarrow_{\mathsf{NB}}$ except it can be applied even if a failed literal is not detected in the $lookaheadDecide$. Rest of the reduction rules works same as described in DPLL ARS, in Figure 1.

Remark. For simplicity, the clauses of same level are grouped together into a CNF-Formula. So, F^0 denotes clauses of original problem and clauses learnt at level 0 while F^2 denotes the clauses learnt at level 2.

3.3.1. Decision Heuristic

The most simple and popular branching heuristic which is still used in lookahead SAT solvers, is given by Freeman in [Fre95]. It is based on the *simplification hypothesis* by Hooker and Vinay [HV95]. Before stating what the simplification hypothesis is, the notion of simpler problem is required. A possible definition taken from [Fre95] is given next.

Definition 3.12. A *simpler problem* is the one with fewer and shorter clauses.

Definition 3.13. Given $F_1 :: J_1$ and $F_2 :: J_2$, then $F_2 :: J_2$ is a *sub-problem* of $F_1 :: J_1$, if $level(J_2) > level(J_1)$ and $F_1 :: J_1 \rightsquigarrow^+ F_2 :: J_2$.

Hypothesis 3.14. Simplification hypothesis*: Other things equal, a decision heuristic works better when it creates simpler sub-problems.*

Freeman [Fre95] defines the decision heuristic as choosing the variable which gives the most simplest sub-problems. One way for choosing such a variable is to first calculate the *difference* (in short $diff$) each polarity of that variable makes to original problem by performing one-step lookahead, and then combining the $diff$ score of each polarity. This combined $diff$ score is called $mixdiff$ [HvM09]. Given a CNF-Formula F and a literal $L = A \in atom(F)$, mathematically $mixdiff$ can be written as:

$$\begin{aligned} mixdiff(F :: J, A) =& 1024 * diff(F :: J, \dot{L}) * diff(F :: J, \dot{\overline{L}}) + diff(F :: J, \dot{L}) \\ &+ diff(F :: J, \dot{\overline{L}}) \end{aligned}$$

So, the decision heuristic is given by:

$$lookaheadDecide(F :: J) = arg\ max_{A \in atom(F|_J)} mixdiff(F :: J, A)$$

There are different variations of $diff$ based on how it is calculated. One given by Freeman is to count the number of assigned variables. Given F and L, $diff1$ calculates the number of assigned variables by performing $lookahead$ on F with respect to L, if L is not a failed literal. Mathematically it is defined as:

$$diff1(F :: J, \dot{L}) = \begin{cases} |atom(F) - atom(F|_{J,\dot{L},P})| & \text{, if } F :: J, \dot{L} \rightsquigarrow^*_{\mathsf{UNIT}} F :: J, \dot{L}, P \text{ and} \\ & \quad lookahead(F :: J, \dot{L}) = \bot \\ 0 & \text{, if } lookahead(F :: J, \dot{L}) = \top \end{cases}$$

Another variation is to calculate the number of new binary clauses, denoted by $diff2$. Mathematically it is defined as:

$$diff2(F :: J, \dot{L}) = \begin{cases} |\{C \text{ such that } |C| = 2 \text{ and} \\ \quad C \notin F|_J \text{ and} \\ \quad C \in F|_{J,\dot{L},P}\}| & \text{, if } lookahead(F, \dot{L}) = \bot \text{ and} \\ & \quad F :: J, \dot{L} \rightsquigarrow^*_{\mathsf{UNIT}} F :: J, \dot{L}, P \\ 0 & \text{, if } lookahead(F, \dot{L}) = \top \end{cases}$$

A generalized decision heuristic model called *recursive weighted heuristic*, for CNF formulas of maximum clause size 3, is presented in [MdWH10] which is extended for CNF formulas of arbitrary size in [AF10]. Recursive weighted heuristic is an iterative model and accuracy of the heuristic increases with the number of iterations performed. The heuristic value $h_i(L)$ means the tendency of the literal L being an element of the model for a given $F :: J$.
For each $A \in atom(F|_J)$:

$$h_0(A) = h_0(\neg A) = 1$$

For each $L \in atom(F|_J) \cup \overline{atom(F|_J)}$:

$$h_{i+1}(L) = \sum_{C \in F|_J} \left(\frac{\gamma^{k-|C|}}{\mu_i^{|C|-1}} \prod_{L_1 \in C \setminus \{L\}} h_i(\overline{L_1})\right)$$

$$\mu_i = \frac{1}{2 * |atom(F)|} \sum_{A \in atom(F)} (h_i(A) + h_i(\neg A))$$

where k is the maximum clause size, γ is the importance constant which is set to 5 by [AF10] and μ_i is the average heuristic value in iteration i. The importance constant γ is used to give more weight to shorter clauses and the average heuristic value μ_i to normalize the scores. The heuristic value for a literal is treated as $diff$ score and $mixdiff$ is used to calculate the value of a variable in F.

3.3.2. Polarity Heuristic

After deciding the variable with the decision heuristic, the next step is to choose the polarity of the variable to be examined first. There are different strategies to do so and different solvers go for different ones.

The lookahead SAT solver *satz* [LA97] always chooses positive polarity. The lookahead SAT solver *kcnfs* [DD04] chooses the polarity which has higher frequency of the chosen decision variable in the formula. The lookahead SAT solver *march* [HvM06] chooses the polarity for which the $diff$ score is lower. The reason for choosing polarity with lower $diff$ score is that the lower $diff$ score assigns less number of variables than the the higher $diff$ score does, thus has lower probability of making mistake and higher probability of leading to a satisfiable solution. On the other hand, polarity with higher $diff$ score can lead to less computation than polarity with lower $diff$ score. It is a good idea to check polarity with higher $diff$ score if the complementary polarity pair $(L, \overline{L})$ $diff$ scores are not comparable. *Comparable* is defined as

$$c \leq \frac{diff(L)}{diff(\overline{L})} \leq \frac{1}{c}$$

where c is a parameter and its value lies between 0 and 1. This strategy of choosing polarity is called *adaptive polarity heuristic* and it has been introduced in the later version of the SAT solver *march* [Heu08]. The value for c used in *march* is 0.1.

3.3.3. Pre-selection Heuristic

Pre-selecting small number of variables to be used by the decision heuristic can reduce the computational cost of the decision heuristic. On the other hand, this pre-selection of the variables can also degrade the overall performance of the SAT solver if the pre-selected variables do not contain the optimal variable (the variable which would have been chosen by the decision heuristic without performing the pre-selection of variables). Due to these reasons, pre-selection of the variables is a very crucial step. Some of the heuristics for pre-selection of variables are discussed here.

The SAT solver *satz* [LA97] uses $prop_z$ heuristic, which pre-selects the variables based on their occurrence in the binary clauses.

$$prop_z(F) = \{A \mid A \in atom(C) \text{ and } C \in F \text{ and } |C| = 2\}$$

The SAT solver *kcnfs* [DD04] also uses $prop_z$. The pre-selection heuristic used by the

SAT solver *march* [HvM06] is based on the approximated number of newly created clauses, called *clause reduction approximation*. To explain what CRA is, first it is helpful to have a function $freq_{>2}$. Consider a literal L, a clause C and a CNF formula F, then the function $freq_{>2}(F, L)$ returns the number of clauses in which L is present and whose size is greater than 2.

$$freq_{>2}(F, L) = |\{C \mid C \in F \text{ , } L \in C \text{ and } |C| > 2\}|$$

$$CRA(F, A) = (\sum_{\{A, L_1\} \in F} freq_{>2}(F, \overline{L_1})) * (\sum_{\{\neg A, L_2\} \in F} freq_{>2}(F, \overline{L_2}))$$

The heuristic CRA only considers the variables in binary clauses and their impact on the formula. The heuristic CRA is an approximation because it only counts the number of newly created clauses and does not cater the fact that some of these newly created clauses might be satisfied. The free variables are sorted in descending order on the score returned by CRA and the pre-selection heuristic gives some top percentage of these sorted variables. The SAT solver *march* takes top 10%.

The decision heuristic RWH can also be used as pre-selection heuristic.

3.3.4. Local Learning

The decision heuristic $lookaheadDecide$ is computationally expensive and to get most out of it, some other techniques are applied during that computation time.

While performing $lookahead(F :: J, \dot{(L)})$, where F is CNF formula and L is a literal in one of clause in $F|_J$, some literals e.g. L_1 may be propagated by the unit rule. This means that the literal L_1 is implied by the literal L and these implications are either called *direct implications* if there exists a clause $\{\overline{L}, L_1\}$ in F or *indirect implication* otherwise. Consider the $level(J) = l$, then the indirect implications can be added as binary clause e.g. $\{\overline{L}, L_1\}^l$ to the CNF formula F, in order to further constraint F and this addition of binary clauses is called *local learning* [HvM09]. As the name suggests, these clauses are not globally valid and they must be removed when backtracking, i.e. when level becomes less than l. The following example will make things clear which requires the definition of iterative unit propagation.

Definition 3.15. Given $F :: J$, the function *iterative unit propagation* $iup(F :: J, \dot{L})$ after deciding L is defined as:

$$iup(F :: J, \dot{L}) = s(P) \qquad \text{if } F :: J, \dot{L} \rightsquigarrow^*_{\text{UNIT}} F :: J, \dot{L}, P$$

Example 3.16. Consider CNF formula
$F = \{\{\overline{1}, 2\}, \{\overline{1}, \overline{2}, 3\}, \{\overline{1}, \overline{3}, 4\}, \{1, 3, 6\}, \{\overline{1}, 4, \overline{5}\}, \{1, \overline{6}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
as current level is 0, so $F^0 := F$. Suppose 1 is chosen as a decision literal, then

	$F^0 :: (\,)$	
$\rightsquigarrow_{\text{DECIDE}}$	$F^0 :: (\dot{1})$	$F^0\vert_{(\dot{1})} = \{\{2\}, \{\overline{2}, 3\}, \{\overline{3}, 4\}, \{4, \overline{5}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1}, 2)$	$F^0\vert_{(\dot{1},2)} = \{\{3\}, \{\overline{3}, 4\}, \{4, \overline{5}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1}, 2, 3)$	$F^0\vert_{(\dot{1},2,3)} = \{\{4\}, \{4, \overline{5}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1}, 2, 3, 4)$	$F^0\vert_{(\dot{1},2,3,4)} = \{\{5, \overline{6}\}\}$

$iup(F^0 :: (\dot{1})) = \{2, 3, 4\}$ and the implication clauses are $\{\overline{1}, 2\}$, $\{\overline{1}, 3\}$, $\{\overline{1}, 4\}$. The first implication clauses is is a direct implication while second and third are indirect implications.

One problem with the local learned clauses is that they are too many (more than the original number of clauses in the CNF-Formula) and they can degrade the performance of unit propagation. A computationally cheap solution to this problem is detection of *necessary assignments*, inspired from Stålmarcks's proof procedure [SS98] for propositional logic.

Definition 3.17. Given $F :: J$ and some decision literal $\dot{L}$, then L_1 is a *necessary assignment* iff:

$$L_1 \in iup(F :: J, \dot{L}) \cap iup(F :: J, \dot{\overline{L}})$$

Example 3.18. Consider the last example where
$F^0 = \{\{\overline{1}, 2\}, \{\overline{1}, \overline{2}, 3\}, \{\overline{1}, \overline{3}, 4\}, \{1, 3, 6\}, \{\overline{1}, 4, \overline{5}\}, \{1, \overline{6}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
and $iup(F^0 :: (\dot{1})) = \{2, 3, 4\}$.
Now choosing $\overline{1}$ as the decision literal.

$$\begin{array}{lll} & F^0 :: (\,) & \\ \rightsquigarrow_{\text{DECIDE}} & F^0 :: (\dot{\overline{1}}) & F^0|_{(\dot{\overline{1}})} = \{\{3, 6\}, \{\overline{6}\}, \{4, 5, 6\}, \{5, \overline{6}\}\} \\ \rightsquigarrow_{\text{UNIT}} & F^0 :: (\dot{\overline{1}}, \overline{6}) & F^0|_{(\dot{\overline{1}}, \overline{6})} = \{\{3\}, \{4, 5\}\} \\ \rightsquigarrow_{\text{UNIT}} & F^0 :: (\dot{\overline{1}}, \overline{6}, 3) & F^0|_{(\dot{\overline{1}}, \overline{6}, 3)} = \{\{4, 5\}\} \end{array}$$

$iup(F^0 :: (\dot{\overline{1}})) = \{3, \overline{6}\}$ and the literal 3 is a necessary assignment.

Another technique is to find equivalent literals and add two binary implication clauses to local learnt clauses – one for each direction of equivalence. This technique is called *equivalence reasoning* by Li [Li03].

Definition 3.19. Given $F :: J$, then L_1 and L_2 are *equivalent literals* if:

$$L_2 \in iup(F :: J, \dot{L_1}) \cap \overline{iup(F :: J, \dot{\overline{L_1}})}$$

3.3.5. Double Lookahead

The idea of *double lookahead* to check if a literal gets a high $diff$ score. This check is done by performing another lookahead, as also indicated by the name. The idea of double lookahead comes from the paper [Li99] by Li.

Definition 3.20. Given $F :: J, \dot{L_1}, P$ then $doubleLookahead$ is:
$doubleLookahead(F :: J, \dot{L_1}, P)$

$$= \begin{cases} \top & \text{,if } lookahead(F :: J, \dot{L_1}, P, \dot{L_2}) = \top \text{ and} \\ & \quad lookahead(F :: J, \dot{L_1}, P, \dot{\overline{L_2}}) = \top \\ & \quad \text{for some } L_2 \in atom(F|_{J, \dot{L_1}, P}) \cup \overline{atom(F|_{J, \dot{L_1}, P})} \\ \bot & \text{,otherwise} \end{cases}$$

doublelookahead $(F :: J, \dot{L}, P)$

1 **if** $diff(F :: J, \dot{L}) > trigger$
2 **repeat**
3 **for** every $A \in VarSubset$ and $A \notin atom(s(J, \dot{L}, P))$
4 **if** $lookahead(F :: J, \dot{L}, P, \dot{A})$ is $\top$ and $lookahead(F :: J, \dot{L}, P, \neg\dot{A})$ is $\top$
5 **return** $\top$
6 **end if**
7 $F := F \cup doublelookaheadresolvent(F :: J, \dot{L}, P)$
8 **end for**
9 **until** nothing (important) has been learnt
10 **end if**
11 $update_{trigger}(F :: J, \dot{L}, P)$
12 **return** $\bot$

Figure 3: Pseudo code of the *doubleLookahead*

Double lookahead is successful (denoted by $\top$ in definition) if it finds conflict on both polarities of a variable, otherwise unsuccessful (denoted by $\bot$). Important point is to know when to perform double lookahead. Li proposes to perform double lookahead on $F :: J, \dot{L}$ if a $diff(F :: J, \dot{L}) > trigger$. He suggest to initialize with $trigger_1 = 0.17 * |atom(F)|$ and update the value of the $trigger_1$ according to following:

$$update_{trigger1}(F :: J, \dot{L}_1, P) = \begin{cases} diff(F :: J, \dot{L}_1) & \text{,if } doubleLookahead(F :: J, \dot{L}_1, P) = \bot \\ 0.17 * |atom(F)| & \text{,if } doubleLookahead(F :: J, \dot{L}_1, P) = \top \end{cases}$$

Heule [HvM07] proposes slight modification to initialization and update of trigger value. He initializes with $trigger_2 = 0$ and update $trigger_2$ according to following:

$$update_{trigger_2}(F :: J, \dot{L}_1, P) = \begin{cases} diff(F :: J, \dot{L}_1) & \text{,if } doubleLookahead(F :: J, \dot{L}_1, P) = \bot \\ trigger_2 & \text{,if } doubleLookahead(F :: J, \dot{L}_1, P) = \top \end{cases}$$

The value of $trigger_2$ is slightly reduced after each $lookaheadDecide$.

While performing double lookahead, some failed literals and necessary assignments can be detected. They can be saved as binary learnt clauses and these clauses are called *double lookahead resolvents*. For example, consider that the double lookahead is performed on $F :: J, \dot{L}_1, P$ and L_2 is detected as necessary assignment, then a binary clause $\{\overline{L_1}, L_2\}^{l-1}$, where $l - 1$ is the level of the learnt clause if $level(J, \dot{L}_1, P) = l$.

Fig. 3 shows the pseudo-code of double lookahead. In start, double lookahead checks according to the $trigger$ value whether to perform the procedure or exit without performing anything. Double lookahead is performed on some of the interesting free variables (pre-selected variables) and checks if it could find a variable whose both polarities lead to a conflict. If double lookahead finds such variable, then it return $\top$ which means that double lookahead is successful and if double lookahead is not able to find such variable then returns $\bot$ which means it is unsuccessful. The procedure also adds double lookahead resolvents and the procedure repeats until no double lookahead resolvents can be added.

4. Search Space Partitioning

The idea of *search space partitioning* approach for parallel SAT solving is to divide the search space of the DPLL into partitions and to solve them in parallel, see Fig. 4. Think of search space partitioning as a function ssp which takes a CNF-Formula and returns a set of CNF forumulas.

Definition 4.1. Given F, mathematically ssp is defined as:

$$ssp(F) = \{F_1, F_2, \ldots, F_n\}$$

such that

- F is satisfiable if there exists $F_i \in ssp(F)$ is satisfiable
- F is unsatisfiable if every $F_i \in ssp(F)$ is unsatisfiable

Hyvärinen [HJN10] divides the search space partitioning approach into two parts: *plain partitioning* and *tree partitioning*. The former does partitioning only once and the later does partitioning more than once if some partition is not solved in a given time limit. A partition is said to be solved if it is shown to be either satisfiable or unsatisfiable. Tree partitioning is also called *iterative partitioning*. Figure 5a shows an example of plain partition, which creates two partitions F_1 and F_2 of the given problem F. Example of iterative partitioning is shown in Figure 5b. It first creates two partitions F_1 and F_2 for the given problem F, then each partition is tried to solve. Suppose the partitions F_1 and F_2 are not solved in given time limit, then the iterative partitioning creates partitions by applying the paritioning function on F_1 and F_2 which results in partitions F_{11}, F_{12}, F_{21}, F_{22}.

As mentioned in the Section 1, the focus of this work is on iterative partitioning. The rest of the document focuses on iterative partitioning and different methods for its creation.

4.1. Partitioning Methods

The methods discussed here to create partitions are different kinds of splitting and the definition of the splitting is as follows:

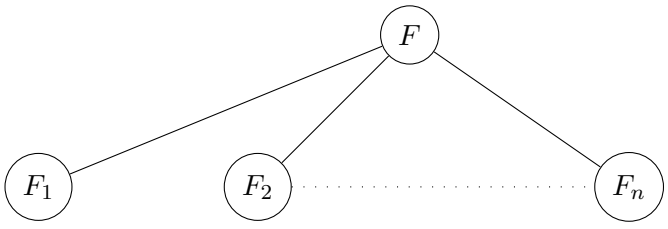

Figure 4: Search Space Partitioning

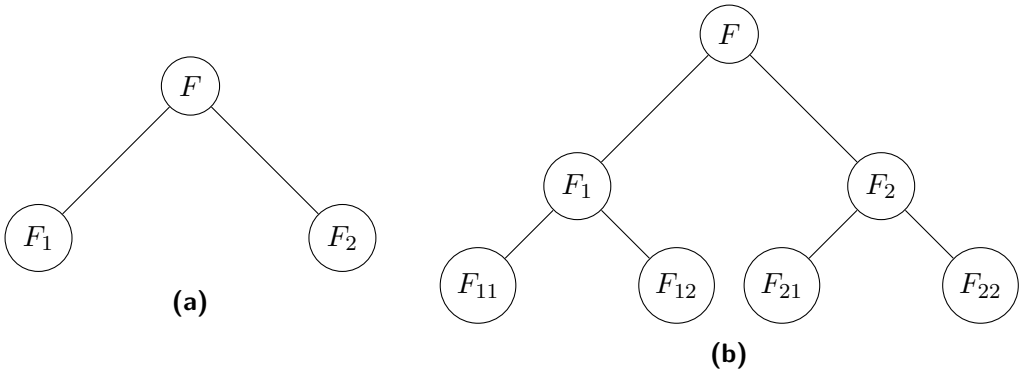

Figure 5: (a) Plain Partition **(b)** Iterative Partition

Definition 4.2. The function $ssp(F)$ is called *splitter* and the elements of the set returned by $ssp(F)$ are called *splittings* if the following condition is observed:

$$F_i \cup F_j \text{ is unsatisfiable, for } 1 \leq i < n \text{ and } i < j \leq n$$

Remark. In this work, splitting and partition are used interchangeably.

4.1.1. Simple Partitioning Method

The *simple partitioning method* creates a search tree by branching on literals and returns the child nodes as splitting. For example, given F, the splittings of F created by simple partitioning method are $simple(F) = \{F_1, F_2, F_3, F_4\}$.

$$F_1 = F \cup \{\{L_1\}, \{L_2\}\}$$
$$F_2 = F \cup \{\{L_1\}, \{\overline{L_2}\}\}$$
$$F_3 = F \cup \{\{\overline{L_1}\}, \{L_3\}\}$$
$$F_4 = F \cup \{\{\overline{L_1}\}, \{\overline{L_3}\}\}$$

Figure 6 shows how a simple partitioning method can be used to create splittings. The beauty of simple partitioning method is its simplicity, i.e. it is easy to create. The

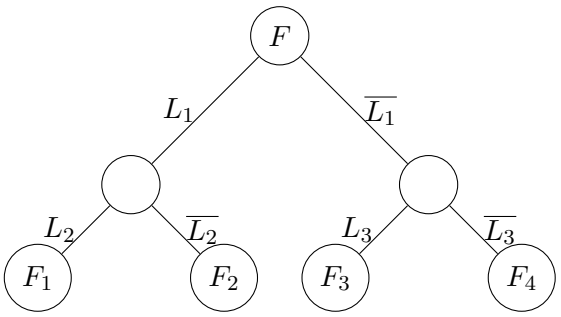

Figure 6: Simple Method

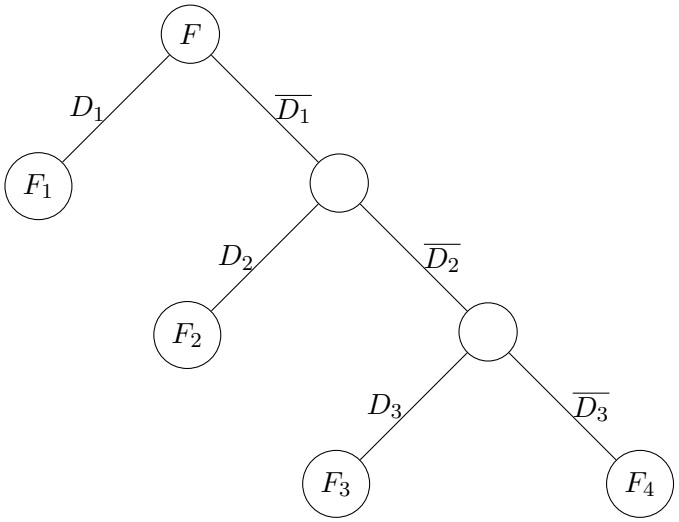

Figure 7: Scattering Partitioning Method

drawback with simple partitioning method is that it can not guarantee the number of splittings it produces and the reason is that some of the splitting are faced with conflict when they are created. For example, in the Figure 6 it can be the case that $\overline{L_2}$ is failed literal, due to which F_2 is unsatisfiable splitting and $simple(F)$ returns 3 splittings instead of 4.

4.1.2. Scattering Partitioning Method

The *scattering partitioning method* [HJN06] creates a search tree same as simple method but by branching on cubes and clauses.

Definition 4.3. Given a cube $D = \{\{L_1\}, \{L_2\}, \ldots, \{L_k\}\}$, then the function $cube2clause(D)$ returns a clause containing all the literals in the cube.

$$cube2clause(D) = \{L_1, L_2, \ldots, L_k\}$$

Then negation of a cube D, in symbols $\overline{D}$, is given by:

$$\overline{D} = \overline{cube2clause(D)}$$

Then splitting of F created by scattering can be explained by the following example. Consider F, then scattering produces 4 splittings, i.e. $scattering(F) = \{F_1, F_2, F_3, F_4\}$, which are

$$\begin{aligned}
F_1 &= F \cup D_1 := \{\{L_1\}, \{L_2\}\} \\
F_2 &= F \cup \{\overline{D_1}\} \cup D_2 := \{\{L_3\}\} \\
F_3 &= F \cup \{\overline{D_1}\} \cup \{\overline{D_2}\} \cup D_3 := \{\{L_4\}\} \\
F_4 &= F \cup \{\overline{D_1}\} \cup \{\overline{D_2}\} \cup \{\overline{D_3}\}
\end{aligned}$$

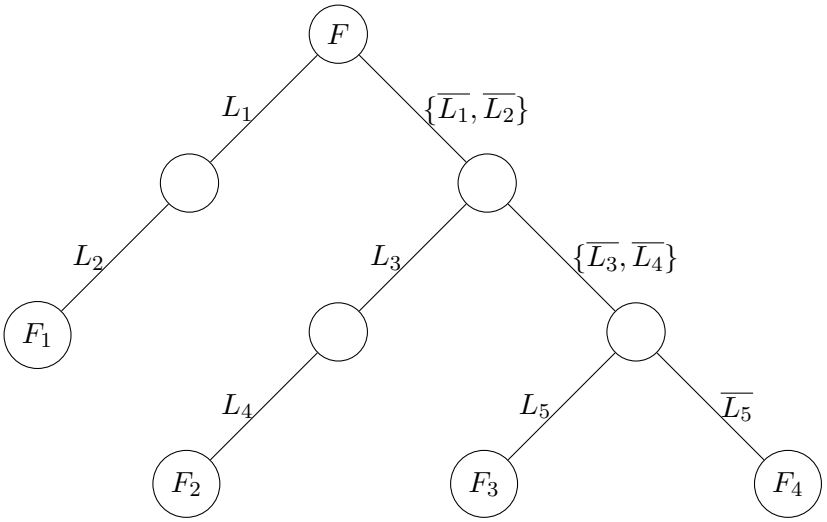

Figure 8: Scattering Method with Sequence Cutoff Heuristic

Figure 7 shows how scattering method creates splittings. Starting from the root of the tree in the Figure 7, the scattering method makes a certain number of decisions to create the left branch with the cube D_1 and right branch is the negation of D_1 (by complementing the set obtained from the function $cube2clause(D_1)$). This process of creating cube and clause is applied iteratively on the right branch until the desired number of leaves (splittings) are produced. An advantage about scattering is that it can say with higher probability about the number of splittings it will produce than the simple partitioning method. In other words, scattering method has better control over the number of splittings than the simple partitioning method.

It is quite clear that the process of creating splitting, either simple or scattering method, is actually selection of variables. Hyvärinen and Manthey [HM12a] use VSIDS heuristic to select the variables for splitting (called *VSIDS splitter* in this work). VSIDS heuristic is discussed in Section 3.2. Hyvärinen [HJN10] also uses lookahead heuristic to choose the variables and their polarity (called *lookahead splitter* in this work). He uses simple decision heuristic $diff1$ and no local reasoning techniques. This work extends his idea of lookahead splitting by additional decision heuristic, polarity heuristic, local reasoning, double lookahead and variable pre-selection heuristic. The lookahead techniques are discussed in Section 3.3.

4.2. Cutoff Heuristic

An important point is the number of selected variables per splitting and this can be modelled by *cutoff heuristic*. A *static cutoff heuristic* chooses fixed number d of variables per splitting. According to static cutoff heuristic, a splitting is created when $level(J) = d$ is reached. This can be applied to both simple and scattering search.

A slightly modified static cutoff heuristic for scattering search is given by Hyvärinen [Hyv11] which uses separate fixed number d_i of variables for each splitting i and I call it *sequence cutoff heuristic*. To compute the values of d_i, following equations are

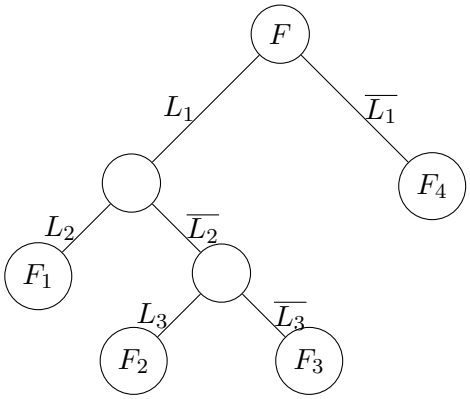

Figure 9: Simple Method with Dynamic Cutoff Heuristic

used:

$$d_i = arg_{x \in \mathbb{N}} min |r_i - 2^{-x}|$$
$$\text{where} \quad r_i = \frac{1}{n - i + 1}$$

Here is an example.

Example 4.4. Given F, consider a scattering produces 4 splitting, i.e. $scattering(F) = \{F_1, F_2, F_3, F_4\}$ (see Figure 8).
As $n = 4$, so
$r_1 = \frac{1}{4} \quad d_1 = 2 \quad F_1 = F \cup \{\{L_1\}, \{L_2\}\}$
$r_2 = \frac{1}{3} \quad d_2 = 2 \quad F_2 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{L_3\}, \{L_4\}\}$
$r_3 = \frac{1}{2} \quad d_3 = 1 \quad F_3 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{\overline{L_3}, \overline{L_4}\}\} \cup \{\{L_5\}\}$
$r_4 = 1 \quad d_4 = 0 \quad F_4 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{\overline{L_3}, \overline{L_4}\}\} \cup \{\{\overline{L_5}\}\}$
Figure 8 shows sequence cutoff heuristic for scattering. Starting from root, the left branch makes 2 decsions L_1 and L_2 (because of $d_1 = 2$). Then on the right branch, the clause $\{\overline{L_1}, \overline{L_2}\}$ is added to the formula F. This right branch is further branched with 2 decisions L_3, L_4 on the left branch and the clause $\{\overline{L_3}, \overline{L_4}\}$ on the right branch. Similarly the last produced right branch is again branched with one decision L_5.

A *dynamic cutoff heuristic* is used in cube&conquer [HKWB12] and it is as follows:
CC-Condition: $level(J)^2 * |s(J)| < \theta_{cutoff} * |atom(F)|$
A splitting is created if the above condition is violated. This dynamic cutoff heuristic is made for plain partitioning (splitting is done once) and this work is focused on iterative partitioning, so the dynamic cutoff heuristic needs to be modified. The heuristics are different for simple method and scattering method. at the dynamic cutoff heuristic for simple method.

Given $F :: J$, the modified dynamic cutoff heuristic for simple method creates a splitting when the lookahead splitter violates the following condition:
Condition-1 : $level(J)^2 * |s(J)| * log(numconflict) < \theta_{cutoff} * |atom(F)|$

The dynamic cutoff heuristic for scattering method creates a splitting if any of the following condition is violated by the lookahead splitter:
Condition-2 : $level(J)^2 * |s(J)| * log(numconflict) < \theta_{cutoff} * |atom(F)| * log(remainchild)$
Condition-3 : $level(J) < maxclausesize - 1$
where $numconflict$ is the number of conflicts that will be used as a limit to solve each splitting, $remainchild$ is the number of remaining number of splittings to be created and θ_{cutoff} is the magic constant which need to be tuned. θ_{cutoff} is decreased by factor 0.7 if the chosen variable becomes a failed literal and is increased by factor 0.05 after creating each splitting (to avoid getting value too low). The difference between condition-1 and condition-2 is the $log(remainchild)$ which is inspired from the sequence cutoff heuristic, so that number of decision literals decreases as the number of created splittings increases. The condition-3 enforces a fixed upper bound of the decision variables per splitting. This limit comes from the simplification hypothesis (see Section 3.3). As the goal is to produce simpler sub-problem and in scattering search the literals selected for one splitting appear as a complemented literals in a clause in the latter splitting. Due to the simplification hypothesis, scattering method should not make decisions more than the size of the maximum clause of the original problem. Although, no clauses are added to original problem as done in scattering method, but the condition-3 can serve as a guide for simple search. Here is an example of comparison between dynamic cutoff heuristic and simple cutoff heuristic for simple method.

Example 4.5. Given F, the simple splitting of F using dynamic cutoff heuristic are $simple(F) = \{F_1, F_2, F_3, F_4\}$ (see Fig. 9).

$$
\begin{aligned}
F_1 &= F \cup \{\{L_1\}, \{L_2\}\} \\
F_2 &= F \cup \{\{L_1\}, \{\overline{L_2}\}, \{L_3\}\} \\
F_3 &= F \cup \{\{L_1\}, \{\overline{L_2}\}, \{\overline{L_3}\}\} \\
F_4 &= F \cup \{\{\overline{L_1}\}\}
\end{aligned}
$$

The difference between static cutoff heuristic and sequence cutoff heuristic for simple partitioning method can seen by comparing Figure 6 and Figure 9.

5. Specification and Implementation

This work is an extension to the SAT solver Splitter [HM12b] from the SAT Challenge 2012. The implementation of Splitter is based on MiniSAT 2.2 [Nik10]. The SAT solver Splitter uses iterative partitioning with VSIDS heuristic. It has been slightly changed and now it uses number of conflicts as a metric to limit the solving phase; earlier it used time as a metric to do that. I have extended Splitter with lookahead techniques and call it as lookahead splitter (LA splitter).

The specification of lookahead splitter can be seen as three different parts (see Figure 10):

1. Structure – partitioning method, cutoff heuristic.

2. Decision – variable decision heuristic, polarity heuristic, variable pre-selection heuristic, double lookahead.

3. Learning – failed literal, necessary assignment, variable equivalence, double lookahead resolvents, clause learning.

The structure is about choosing the partitioning method i.e. simple or scattering and cutoff heuristic which can be static, sequence or dynamic. Figure 10 shows them as *structure options* box. The decision part of the lookahead splitter includes the heuristics about variable decision, polarity and variable pre-selection for performing lookahead. It also includes about choosing to use or not to use double lookahead. The decision part is shown as *decision options* box in Figure 10. The different options of learning part of lookahead splitter are given in the Figure 10 as *learning options* box. It includes the option to use the failed literal, necessary assignment, variable equivalence, double lookahead resolvents, clause learning while creating splitting and this option mentioned as *local*. An important thing to mention here is the use of *partition* option mentioned in the Figure 10, when used pushes the learning to splitting. Here is an example to clarify the learning options:

Example 5.1. Consider CNF formula
$F = \{\{\overline{1}, 2\}, \{\overline{1}, \overline{2}, 3\}, \{\overline{1}, \overline{3}, 4\}, \{1, 3, 6\}, \{\overline{1}, 4, \overline{5}\}, \{1, \overline{6}\}, \{4, 5, 6\}, \{5, \overline{6}\}\}$
To keep things simple, I will create splittings by deciding one variable. Suppose the pre-selection heuristic gives the variable set $\{1, 6\}$. With learn options off for lookahead will choose 1, because $diff1$ score (which is based on the number of assigned variables) for the variable 1 is higher than the variable 6. Here are the runs of lookahead ARS:
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{1}) \rightsquigarrow^*_{UNIT} F :: (\dot{1}, 2, 3, 4)$
$F :: (\dot{1}, 2, 3, 4) \rightsquigarrow_{LA\_BACK} F :: ()$
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{\overline{1}}) \rightsquigarrow^*_{UNIT} F :: (\dot{\overline{1}}, \overline{6}, 3)$
$F :: (\dot{\overline{1}}, \overline{6}, 3) \rightsquigarrow_{LA\_BACK} F :: ()$
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{6}) \rightsquigarrow^*_{UNIT} F :: (\dot{6}, 1, 2, 3, 4, 5)$
$F :: (\dot{6}, 1, 2, 3, 4, 5) \rightsquigarrow_{LA\_BACK} F :: ()$
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{\overline{6}}) \rightsquigarrow^*_{UNIT} F :: (\dot{\overline{6}})$
$F :: (\dot{\overline{6}}) \rightsquigarrow_{LA\_BACK} F :: ()$

By deciding on literal 1, the number of assigned variables is 4 and by deciding on literal $\overline{1}$, the number of assigned variables is 3. Then $diff1$ is calculated as:
$diff1(1) = 1024 * 4 * 3 + 4 + 3 = 12295$
Similarly the $diff1$ score for variable 6 can be calculated as:
$diff1(6) = 1024 * 6 * 1 + 6 + 1 = 6151$

The splittings of F produced by lookahead with local learn and partition learn option can be seen in Table 1. Here is the run of lookahead ARS with local learn option and partition learn option (the runs are same for local learn and partition learn, the difference is only in the splittings produced):
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{1}) \rightsquigarrow^*_{UNIT} F :: (\dot{1}, 2, 3, 4)$
$F :: (\dot{1}, 2, 3, 4) \rightsquigarrow_{LA\_BACK} F :: ()$
$F :: () \rightsquigarrow_{DECIDE} F :: (\dot{\overline{1}}) \rightsquigarrow^*_{UNIT} F :: (\dot{\overline{1}}, \overline{6}, 3)$
$F :: (\dot{\overline{1}}, \overline{6}, 3) \rightsquigarrow_{LR} F, \{\{3\}^0\} :: (\dot{\overline{1}}, \overline{6}, 3)$ 3 is a necessary assignment
$F, \{\{3\}^0\} :: (\dot{\overline{1}}, \overline{6}, 3) \rightsquigarrow_{LA\_BACK} F, \{\{3\}^0\} :: ()$
$F, \{\{3\}^0\} :: () \rightsquigarrow_{UNIT} F, \{\{3\}^0\} :: (3)$
$F, \{\{3\}^0\} :: () \rightsquigarrow_{DECIDE} F :: (\dot{6}) \rightsquigarrow^*_{UNIT} F :: (3, \dot{6}, 1, 2, 4, 5)$
$F, \{\{3\}^0\} :: (3, \dot{6}, 1, 2, 4, 5) \rightsquigarrow_{LA\_BACK} F :: (3)$
$F, \{\{3\}^0\} :: () \rightsquigarrow_{DECIDE} F :: (3, \dot{\overline{6}}) \rightsquigarrow^*_{UNIT} F :: (3, \dot{\overline{6}})$
$F, \{\{3\}^0\} :: (\dot{\overline{6}}) \rightsquigarrow_{LA\_BACK} F :: (3)$
The $diff1$ is calculated as above.

The difference between the learn options of LA splitter is clear from Table 1 i.e., with the learn option off and the learn option local, the local learnt clause (in this example it is necessary assignment 3) is not included in the splitting while the learn option partition adds the variable 3 in the splittings.

In the rest of this section, I am going to discuss some important implementation details, parameter tuning and the evaluations of the final experiments.

5.1. Implementation

Some important implementation details which differ from specification are discussed here. As mentioned earlier, Splitter is based on MiniSAT and MiniSAT uses lazy data structure [MMZ+01] for the representation of formulas. The lazy data structure can only detect clauses which become unit or unsatisfied and due to this reason it makes very difficult to calculate the $diff2$ score (the number of newly created binary clauses; discussed in Section 3.3). As a remedy to this problem, I have implemented the idea given by [MvVW06] which approximates the $diff2$ score by counting the ternary clauses (clauses of size 3) which become binary.

Learn Option	Splitting F_1	Splitting F_2
Off	$F \cup \{\{1\}\}$	$F \cup \{\{\overline{1}\}\}$
Local	$F \cup \{\{1\}\}$	$F \cup \{\{\overline{1}\}\}$
Partition	$F \cup \{\{1\}, \{3\}\}$	$F \cup \{\{\overline{1}\}, \{3\}\}$

Table 1: Example of Learn Options in LA splitter

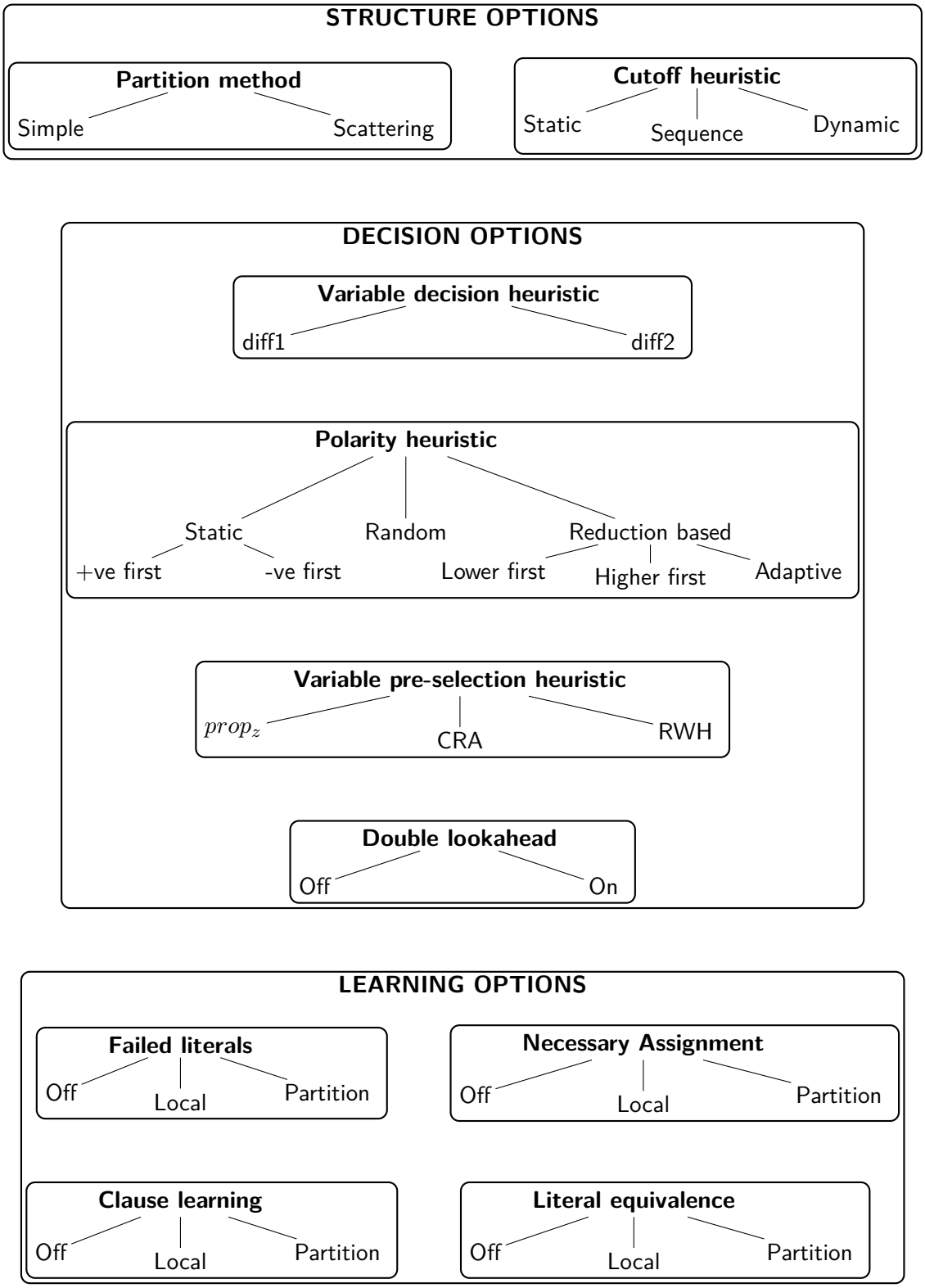

Figure 10: Lookahead Splitter Options

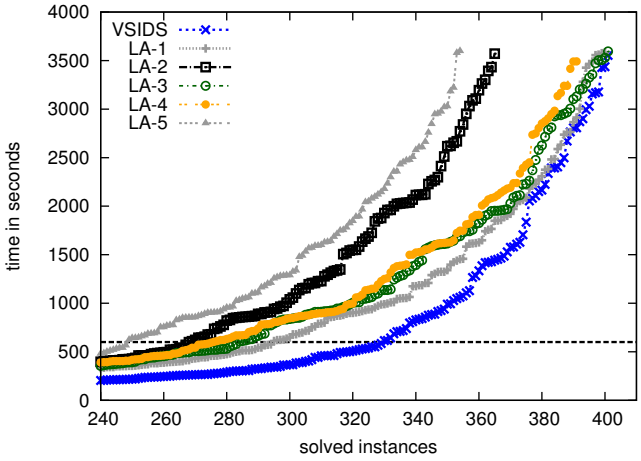

Figure 11: Training Set Creation

The decision heuristic RWH has been discussed in detail in Section 3.3. The calculation of a literal score requires quite a number of multiplications and divisions. It is possible that the calculation could overflow or underflow due to limited expressiveness of the values that a computer could handle. For this particular case, the score of a literal is bounded by upper and lower limit. The values for upper and lower bound are taken from the technical report [AF10], see Appendix A for exact values.

Unit propagation is the heart of a DPLL based SAT solver and the speed of unit propagation is inversely proportional to the number of clauses. The lookahead splitter with learning option learns quite a number of clauses, making the overall performance of the lookahead splitter slower. To cater this problem, I remove all of the local learnt clauses if their count exceeds certain limit. The reason for removing all local learnt clauses is that their number grows very fast (many clauses can be learnt by performing a lookahead on single variable) and selective deletion of clauses would have taken a lot of time. This limit in the implementation is set large enough such that it provides the opportunity to utilize the local learnt clauses before deletion. This limit is equal to the number of the variables in the formula.

5.2. Benchmarks

The benchmark for performing the tests consists of the whole SAT Challenge 2012 [JLBRS12] (601 instances) and this set of instances is called *test set*. The tests run on 16 core AMD Opteron 6274 CPUs with 2.2 GHz. To reduce the time to find the reasonably good configuration for lookahead splitter, a smaller subset called *training set* is chosen from the whole test set. Following is the procedure used to create the training set:

- Perform tests with 3600 seconds time-out and 4 threads on the test set with 6 different configurations (1 VSIDS splitter and 5 LA splitter configurations).
- Divide the test set in to three disjoint subsets:

1. Hard set – the instances which are timed out by all configurations
2. Medium set – the instances whose solution time is between closed interval $(600, 3600)$ seconds by any of the configurations.
3. Easy set – the instances whose solution time is between closed interval $(0, 600)$ seconds by all configurations.

- Take 10%, 50% and 10% of the hard, medium and easy set respectively.

Using this procedure, 109 instances were selected. Figure 11 shows a cactus plot of the 6 configurations used for selecting the instances for training set. The x-axis shows the number of solved instances and the y-axis shows the instance runtime. The area below the doted line at time 600 contains the easy set and the area above is that doted line contains the medium set.

5.3. Parameter Tuning

Parameter tuning is about finding a good configuration of parameters (for lookahead splitter, parameters can be seen as options shown in Figure 10). Parameter tuning is performed on the training set. It is very difficult to play with every parameter (lookahead splitter options, see Figure 10) and tune them in this work. I take the parameter values from other references which have been studied before. Some of the parameters which have not been studied in the context of iterative partitioning with scattering, are tried to be tuned. For a complete list of parameters of the lookahead splitter, see Appendix A. Before I start with the tuning of parameters, the metrics used for evaluation are given below:

- Solved instance: the number of solved instances, higher is better.
- Solved SAT: the number of solved instances which are satisfiable, higher is better.
- Solved UNSAT: the number of solved instances which are unsatisfiable, higher is better.
- Median runtime: the median of runtime over all instances, lower is better.
- Mean split time: the mean of the time used by each call of splitter, lower is better.
- Score: the score of a configuration using the ranking by [VG11], higher is better.

Configuration	Solved instances	Median Runtime	Score
diff1	85	956.65s	-12
diff2	83	808.83s	3
diff4	**88**	**707.17s**	**9**

Table 2: Statistics of decision heuristic

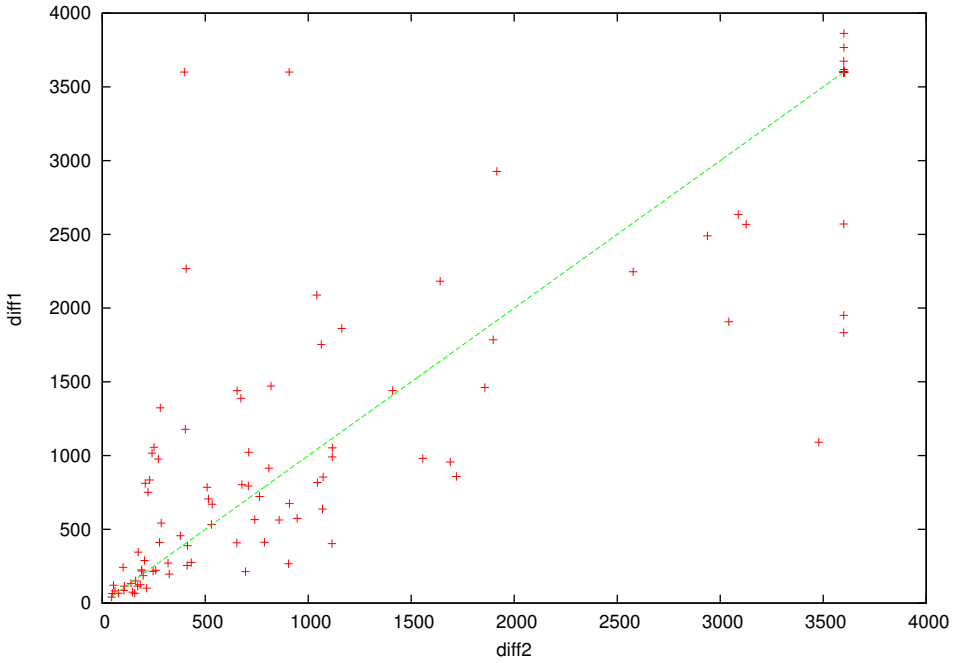

Figure 12: Runtime comparision of decision heuristic

The metric score needs a bit of explanation. The score takes not only the number of solved instances into account, but also the solving time of each instance with some restriction. The restriction is that between two configurations, the score does not consider solving time of an instance if the difference between the solving time by the two configurations is less than 60s. It is clear from the Figure 12 that neither the decision heuristic $diff1$ nor the $diff2$ is better overall. The x-axis in the plot shows the instance runtime solved by the configuration diff2 with decision heuristic $diff2$, while y-axis in the same show the instance runtime solved by of the configuration diff1 with decision heuristic $diff2$. The parameters of the two configurations are the same except for the decision heuristic. On some instances, the configuration diff1 performs better and on some the configuration diff2 performs better. Table 2 show that the configuration diff1 has an edge over the configuration diff2 due to higher number of solved instances. The former solves 85 instances while the latter solves 82 instances. On the other hand, the configuration diff2 seems faster than the configuration diff1 due to better median runtime i.e., 808.83s compared to 956.65s. This gives the intuition to combine the decision heuristics $diff1$ and $diff2$. A simple way is to combine them linearly. Following is the new combined decision heuristic called $diff4$:

$$diff4 = 0.3 * diff1 + 0.7 * diff2 \tag{1}$$

The higher weight to $diff2$ in Equation (1) is due to better median and score, see Table 2. Another reason to give more weight to $diff2$ is due to the importance of $diff2$ over $diff1$ found in literature [BHvMW09]. In Table 2, the configuration diff4 is a lookahead splitter configuration same as diff1 and diff2 with only difference of decision heuristic $diff4$. The

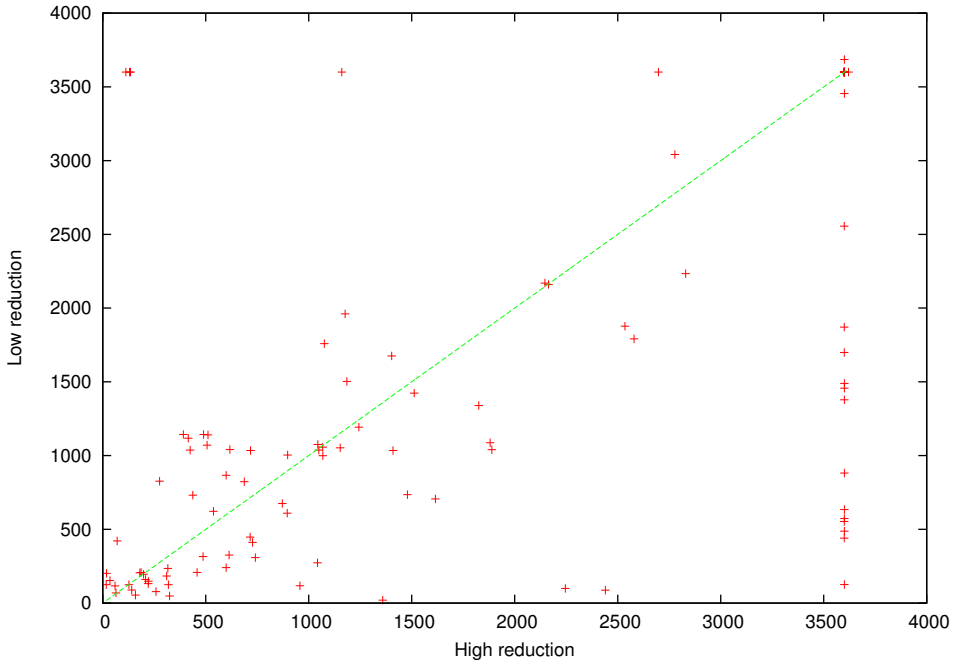

Figure 13: Runtime comparision of polarity – High reduction vs Low reduction

configuration diff4 outperforms both of the configurations diff1 and diff2 by solving 88 instances compared to 85 and 83. The median runtime of the configuration diff4 is 707.17s which is better than 956.65s and 808.83s of the configurations diff1 and diff2 respectively. The score of the configuration diff4 is 9, better than the score of configurations diff1 and diff2 which are -12 and 3 respectively. All the measure in Table 2 support the decision heuristic $diff4$ as the tuned value for the variable decision heuristic.

The next parameter to tune is polarity heuristic. I have experimented with different polarity heuristic and the most interesting case is discussed here which is the comparison between high reduction and low reduction. Figure 13 shows a cross plot of two configurations namely *high reduction* and *low reduction*. The only difference between these two configurations is the polarity heuristic. The former uses high reduction polarity heuristic and later uses low reduction polarity heuristic. The x-axis shows the runtime of instance solved by the configuration high reduction and y-axis shows the runtime of instance solved by low reduction. It is quite clear that the configuration low reduction is outperforming the configuration high reduction (see the number of timeout instances of high reduction).

Configuration	Solved Instances	Median Runtime	Mean Split Time	Score
No Learn	**86**	1024.26s	42.24s	-8
Local Learn	83	863.42s	32.63s	-6
Partition Learn	85	**629.17s**	**22.93s**	**14**

Table 3: Statistics of learning options

One possible reason for better performance of low reduction over high reduction polarity heuristic is that by choosing low reduction polarity the probability of making a mistake is less than choosing higher reduction. This less mistake probability for lower reduction polarity is due to the less number of assigned variables. The tuned value for polarity heuristic is set to be low reduction.

The behaviour of the learning options in lookahead splitter is analysed now. The configurations used for this purpose have the same parameter options except for the learning options. An important detail about the configuration partition learn is that the literal equivalence is set to be local and the reason for that is the lookahead splitter could find huge number of literal equivalences which are only stored locally as binary clauses but not included in splittings. Although the configuration no learn has solved 1 more instance but it is a lot slower than than the configuration partition learn. The configuration partition learn has median runtime of 629.17s which is much faster than median runtime 1024.26s of the configuration no learn and 863.42s of the configuration local learn. The score also tell the same story and the scores are 14, -8, -6 for the configurations partition learn, no learn and local learn respectively. The mean split time is an important measure and it should not be too high because this might cause starvation (there can be a situation when there are no splittings to solve and the splitter is taking too much time to produce splittings). The lowest mean split time is given by the configuration partition learn which is 22.93s while the configurations no learn and local learn give higher mean split time i.e., 42.24s and 32.63s. The configuration partition learn gives the tuned values for the learning options of the lookahead splitter.

The next thing is to see whether it is worth to use double lookahead in the lookahead splitter. For this purpose, two similar lookahead splitter configurations are used which only differ on the use of double lookhead. The configuration woDLA is a lookahead splitter configuration without the double lookahead option, while the configuration wDLA is a lookahead splitter configuration with double lookahead option turned on. Table 4 shows the statistics of the configurations woDLA and wDLA. It shows that the use of double lookahead option in the lookahead splitter is not giving significant benefit, rather it is slowing down the performance. The number of solved instances is 84, median runtime is 776.87s and score is -4 by the configuration wDLA which is slightly bad than the configuration woDLA having 85 number of solved instances, 629.17s median runtime and 4 score. The big difference is seen in the mean split time which is 50.52s for the configuration wDLA compared to 22.93s for the configuration woDLA. As discussed earlier, the high mean split time could cause starvation. The high mean split time for the configuration wDLA could be due to the trigger value (condition when double lookahead is performed) as the success rate of double lookahead is only 29.33%. The double lookahead is successful

Configuration	Solved Instances	Median Runtime	Mean Split Time	Double LA Success Rate	Score
woDLA	**85**	**629.17s**	**22.93s**	-	**4**
wDLA	84	776.87s	50.52s	29.33%	-4

Table 4: Statistics of double lookahead

if it finds conflict on both branches of a variable. Currently the trigger value is set to $0.17 * |atom(F)|$ with adaptive update function. I believe that with a trigger value which increases the success rate of double lookahead to more than 50%, then the overall performance of the configuration wDLA can be improved. I choose not to use double lookahead for now and leave the task of finding better trigger value for future work.

5.4. Evaluation

The final experiments are run with 16 threads in parallel, 16 GB memory limit and 3600s walltime limit over the whole test set of 601 instances. Following are the configuration used for the final experiments:

1. RAND: uses scattering method with random decision literal for splitter.
2. VSIDS: uses scattering method with VSIDS decision heuristic for splitter.
3. LA-simp: uses simple method with LA decision heuristic for splitter.
4. LA-scat: uses scattering method with LA decision heuristic for splitter.

All the above mentioned configurations use number of conflict to limit the solve time which is equal to 512000. The number of splittings to be produced per splitter call is set to 8 and sequence cutoff heuristic is used for the configurations RAND, VSIDS and LA-scat. The configuration LA-simp used a static cutoff heuristic which is set to the value 4, i.e. LA-simp will go up to level 4 and will produce at most 16 splittings per splitter call (most of the time the number of splittings produced are less than 16). The configurations LA-simp and LA-scat both use $diff4$ decision heuristic, lower reduction polarity heuristic, RWH pre-selection heuristic, no double lookahead and all learning options set to partition except for literal equivalence which is set to local (for complete options of LA splitter, see Figure 10).

Table 6 shows the number of solved instances, with median runtime and score of the 4 configurations. Number of solved instances is further divided into satisfiable (SAT) and unsatisfiable (UNSAT) ones. The configuration LA-scat solves the most number of instances and solves 20 more instances compared to the configuration VSIDS. The median runtime of the configuration LA-scat is better than the configuration VSIDS, i.e. 255.93s compared to 278.12s. The score of LA-scat is highest among these configurations and it indicates that LA-scat is a lot better than the configuration VSIDS due to the big difference of scores 155 to 30. Similarly the configuration LA-simp with 428 solved instances and 140 score looks better than the configuration VSIDS with 414 solved instances and

Configuration	Solved	SAT	UNSAT	Median Runtime	Score
RAND	352	236	116	588.01s	-325
VSIDS	414	246	168	278.12s	30
LA-simp	428	**250**	178	286.04s	140
LA-scat	**434**	249	**185**	**255.93s**	**155**

Table 5: Number of solved instances

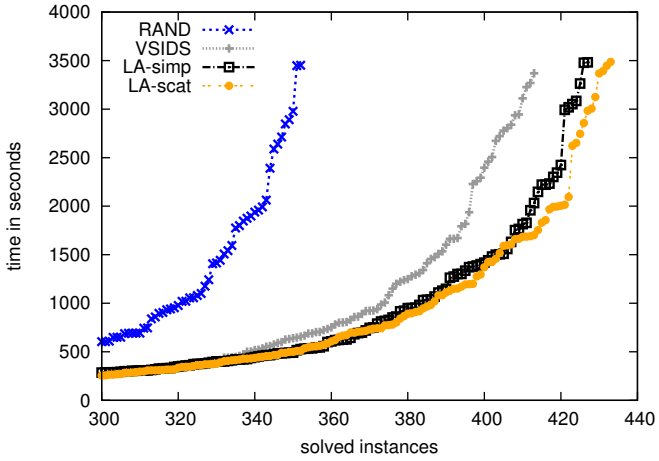

Figure 14: Solved instances and runtime

30 score, but the configuration VSIDS has better median runtime 278.12s than the configuration LA-simp with median runtime 286.04s. Although, the configuration LA-simp and the configuration LA-scat differ in terms of partitioning method, the former uses simple method while the latter uses scattering method, but both of them beat the configuration VSIDS and both use lookahead decision heuristic. With these results, it can be concluded that lookahead is better decision heuristic than VSIDS for creating splittings and lookahead splitter with scattering is faster than VSIDS splitter with scattering. Figure 14 gives a bird's eye view of the 4 configurations. The x-axis shows the number of solved instances and the y-axis shows the runtime of an instance. The configuration LA-scat is a clear winner in the figure. The results negate the conclusion of [HJN10] that VSIDS is better than lookahead for splitting.

Although the configuration LA-simp beats the configuration LA-scat in number of solved SAT instances by 1 (Table 6), but seeing the configuration RAND which solved 236 SAT instances suggests that one could get lucky in solving SAT instances and we can ignore the tiny lead of the configuration LA-simp over LA-scat. The main competition lies in solving UNSAT instances, which is clearly won by LA-scat. The lead in solving UNSAT instances by the configuration LA-scat over the configuration LA-simp, i.e. 185 to 178, supports the claim by [HJN10] that scattering method is better approach for solving UNSAT instances than the simple method.

Configuration	Mean Splittings Count	Mean Split time	CPU Ratio
RAND	65.01	**0.60s**	**14.71**
VSIDS	**42.21**	22.20s	14.00
LA-simp	123.83	15.50s	14.05
LA-scat	80.46	12.30s	13.69

Table 6: Mean time to split and CPU ratio

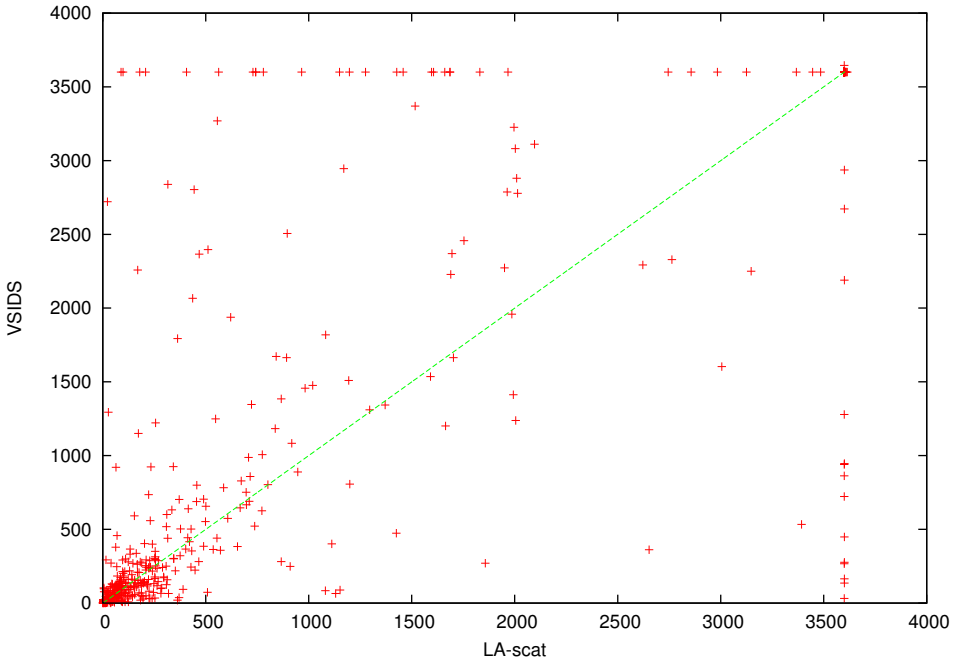

Figure 15: Lookahead splitter vs VSIDS splitter

Table 6 shows the mean of splittings produced, mean split time and CPU ratio of configurations. The CPU ratio tells how many cores are used in average for solving an instance. The configuration LA-scat has mean split time of 12.30s which is much lower than the configuration VSIDS with mean split time of 22.20s and considering the number of solved instances, it can be said that LA-scat is computationally efficient partitioning technique. The configuration RAND has the highest CPU ratio 14.71 and one possible reason is that it spends less time on creating splittings (mean split time is the lowest 0.60s), but overall performance of the configuration RAND is worse than the other configurations. This bad performance may be interpreted that the time to split should be large enough to produce good splittings. Seeing from the mean splitting count of the configurations suggest that this measure could be misleading in judging the best configuration. In current scenario, the configuration VSIDS has the least value for mean splittings count, i.e. 42.21 which is almost half of the configuration LA-scat same value 80.46, but as it has been discussed earlier that the overall performance of LA-scat is a lot better than VSIDS.

The configurations LA-scat and VSIDS are compared in Figure 15. The x-axis shows the runtime of the configuration LA-scat while the y-axis show the runtime of the configuration VSIDS. This figure tells the same story discussed earlier that the configuration LA-scat is overall better and faster than the configuration VSIDS, but it also shows that there are few instances where the configuration VSIDS is faster or sometimes better (solved by the configuration VSIDS and timed out by the configuration LA-scat). These few instances which are solved only or solved faster by the configuration VSIDS compared to the configuration LA-scat gives the motivation to have a configuration selector. The job of a configuration selector is to select a configuration based on some feature and

researching a good configuration selector could be a possible future direction of this work. The configuration selector can select the configuration once at the start or at each iterative call to splitter. The latter approach can lead to some interesting results.

6. Conclusion

In this work, I have discussed lookahead techniques for iterative search space partitioning and shown some nice results. I have shown statistically and also with a novel ranking method [VG11] that the conclusion of Hyvärinen [HJN10] seems to be wrong and in fact lookahead is a better method than VSIDS for iterative search space partitioning. In terms of solved instances, the iterative search space partitioning with lookahead solves 20 more instances than the iterative search space partitioning with VSIDS. Another claim by Hyvärinen is that the scattering method is better than the simple partitioning method for solving unsatisfiable instances and this work supports his claim. Scattering method solved 185 unsatisfiable instances while simple partitioning method solved 178 unsatisfiable instances. Another important result is that the VSIDS splitter performs very well on few instances where the lookahead splitter performs poorly, which gives the motivation to look into some hybrid approach.

Some worth mentioning minor results are that the lower reduction polarity heuristic for scattering seems good strategy and the decision heuristic which counts the number of assigned variables is not the optimal one. A combined decision heuristic is given in this work which is based on counting the number of assigned variables and approximated number of newly created binary clauses. This combined decision heuristic seems to give good results. Another result is that the local reasoning of lookahead seems to improve overall performance of lookahead splitter.

Future directions of this work can be the following:

1. Designing a good configuration selector that can choose between VSIDS and lookahead splitter configurations based on some instance features.

2. Finding a better trigger method for double lookahead that could improve the success rate of double lookahead and the overall performance of lookahead splitter.

3. It would be interesting to see a Tabu scattering method (idea from stochastic local search), meaning that the scattering does not decide on a variable which has already been decided in some splitting earlier.

References

[ABL+10] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In Inês Lynce and Ralf Treinen, editors, *LoCoCo*, volume 29 of *EPTCS*, pages 11–22, 2010.

[AF10] Dimitrios Athanasiou and Marco Alvarez Fernandez. Recursive weight heuristic for random k-sat. Technical report, Delft University of Technology, 2010.

[BBD+12] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors. *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2012. ISBN ISBN 978-952-10-8106-4.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Manage. Sci.*, 35(2):164–176, February 1989.

[DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01, pages 248–253, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[DD04] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-sat formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 305–306. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24605-3_36.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(7):1165–1178, July 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[GHM+12] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving periodic event scheduling problems with sat. In He Jiang, Wei Ding, Moonis Ali, and Xindong Wu, editors, *IEA/AIE*, volume 7345 of *Lecture Notes in Computer Science*, pages 166–175. Springer, 2012.

[Goe10] Asvin Goel. A column generation heuristic for the general vehicle routing problem. In *Proceedings of the 4th international conference on Learning and intelligent optimization*, LION'10, pages 1–9, Berlin, Heidelberg, 2010. Springer-Verlag.

[Heu08] Marijn J.H. Heule. *SmArT solving: Tools and techniques for satisfiability solvers*. PhD thesis, TU Delft, 2008.

[HJN06] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving sat in grids. In *Proceedings of the 9th international conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 430–435, Berlin, Heidelberg, 2006. Springer-Verlag.

[HJN10] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer-Verlag.

[HKWB12] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Accepted for HVC 2011*, 2012. Accepted for HVC 2011.

[HM12a] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 214–227. Springer, 2012.

[HM12b] Antti Eero Johannes Hyvärinen and Norbert Manthey. Splitter – a scalable parallel sat solver based on iterative partitioning. In Balint et al. [BBD+12], page 62. ISBN ISBN 978-952-10-8106-4.

[HMN+11] Steffen Hölldobler, Norbert Manthey, Hau Van Nguyen, Julian Stecklina, and Peter Steinke. A short overview on modern parallel SAT-solvers. In

Proceedings of the International Conference on Advanced Computer Science and Information Systems, pages 201–206, 2011.

[HV95] John N. Hooker and V. Vinay. Branching rules for satisfiability. *J. Autom. Reasoning*, 15(3):359–383, 1995.

[HvM06] Marijn J.H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, mar 2006.

[HvM07] Marijn J.H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.

[HvM09] Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. Volume 185 of Biere et al. [BHvMW09], February 2009.

[HW12] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel sat solving. In Jörg Hoffmann and Bart Selman, editors, *AAAI*. AAAI Press, 2012.

[Hyv11] Antti Eero Johannes Hyvärinen. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Helsinki, Finland, 2011. Doctoral Dissertations 118/2011.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI '92, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371. Morgan Kaufmann, 1997.

[Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Inf. Process. Lett.*, 71(2):75–80, July 1999.

[Li03] Chu-Min Li. Equivalent literal propagation in the dll procedure. *Discrete Appl. Math.*, 130(2):251–276, August 2003.

[LMS06] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 136–141. Springer, 2006.

[MdWH10] Sid Mijnders, Boris de Wilde, and Marijn J. H. Heule. Symbiosis of search and heuristics for random 3-sat. In David Mitchell and Eugenia Ternovska, editors, *Proceedings of the Third International Workshop on Logic and Search (LaSh 2010)*, 2010.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.

[MvVW06] Dimosthenis Mpekas, Michiel van Vlaardingen, and Siert Wieringa. The first steps to a hybrid SAT solver. Technical report, Delft University of Technology, 2006.

[Nik10] Niklas Sörensson. Minisat 2.2 and minisat++ 1.1. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf`, 2010.

[Rou12] Olivier Roussel. Description of ppfolio 2012. In Balint et al. [BBD+12], page 46. ISBN ISBN 978-952-10-8106-4.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarcks's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 1998.

[vdTHB12] Peter van der Tak, Marijn J. H. Heule, and Armin Biere. Concurrent cube-and-conquer. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 475–476, Berlin, Heidelberg, 2012. Springer-Verlag.

[VG11] Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 317–328, Berlin, Heidelberg, 2011. Springer-Verlag.

A. Lookahead Splitter Parameters

Parameter	Description
`-split-method`	Which structure to use. `-split-method=0` is simple and `-split-method=1` is scattering
`-split-depth`	Number of decision to be made for each splitting. If `-split-depth=0` and `-no-dseq` then dynamic cutoff heuristic is used, if `-split-depth=0` and `-dseq` then sequence based cutoff heuristic is used.
`-dseq`	Use sequence based cutoff heuristic for scattering; requires `-split-depth=0`
`-no-dseq`	Do not use sequence based cutoff heuristic for scattering
`-split-thres`	Threshold for dynamic cutoff heuristic; default is 1
`-split-penal`	Penalty factor for the dynamic cutoff heuristic threshold, if a decision literal becomes failed literal; default is 0.7
`-thres-inc`	Increment factor for the dynamic cutoff heuristic threshold, applied in start of the lookahead splitter; default is 0.05
`-child-count`	Number of children (splittings) to produce by scattering method; default is 8
`-la-heur`	Decision heuristic to be used. `-la-heur=0` for $diff1$, `-la-heur=1` for $diff2$, `-la-heur=4` for $diff4$ and `-la-heur=5` for RWH. Default is 4.
`-num-iterat`	Number of iteration for lookaheadDecide; default is 2.
`-dir-prior`	Direction priority. `-dir-prior=0` is for selecting always positive polarity while `-dir-prior=1` is for selecting always negative polarity. `-dir-prior=2` is for selecting the polarity with higher reduction while `-dir-prior=3` is for selecting the polarity with lower reduction. `-dir-prior=4` is for selecting random polarity and `-dir-prior=6` for adaptive direction heuristic. Default is 3
`-dir-adp-fac`	Factor for adaptive direction heuristic; default is 0.1
`-double-la`	Use double lookahead for splitting phase; off by default
`-no-double-la`	Do not use double lookahead for splitting phase
`-double-decay`	Decay rate for the double lookahead threshold (trigger); default is 0.95
`-shrk-clause`	Shrink clauses in start of the lookahead splitter; default is on
`-no-shrk-clause`	Do not shrink clauses in start of the lookahead splitter
`-presel-heur`	Pre-selection to be used. `-presel-heur=0` for $prop_z$, `-presel-heur=1` for CRA and `-presel-heur=2` RWH. Default is 2.
`-presel-fac`	Factor of the free variable to be used in lookahead; default is 0.1
`-presel-min`	Minimum number of pre-selection variables. Default is 128
`-presel-max`	Maximum number of pre-selection variables. Default is 2048
`-h-upper`	upper bound for the score of RWH; default is 10900
`-h-lower`	lower bound for the score of RWH; default is 0.1
`-h-cl-wg`	Importance weight of a shorter clause; default is 5
`-h-maxcl`	The maximum size of the clause to be used in RWH; default is 7
`-h-acc`	Accuracy of the RWH, number of iterations to perform; default is 9

Parameter	Description
`-fail-lit`	Failed literal learning options. `-nec-assign=0` means it is turned off. `-nec-assign=1` means to use it for local learning. `-nec-assign=2` means use it for local learning as well as push it to splitting. Default is 2.
`-nec-assign`	Necessary assignment learning options. `-nec-assign=0` means it is turned off. `-nec-assign=1` means to use it for local learning. `-nec-assign=2` means use it for local learning as well as push it to splitting. Default is 2.
`-clause-learn`	Clause learning options. `-clause-learn=0` means turned off. `-clause-learn=1` means to use it for local learning. `-clause-learn=2` means to use is for local learning and also push it to splitting. Default is 2.
`-var-eq`	Literal equivalence options. `-var-eq=0` means do not use literal equivalence option, `-var-eq=1` means use literal equivalence for checking if lookahead on an equivalent literal has been performed and if yes then do not perform lookahead, `-var-eq=2` means to use `-var-eq=1` and add the equivalent literals relation as binary clauses in the local learnt clauses. Default is 2.

Declaration

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

Dresden, 12 November 2012

Ahmed Irfan