A Theory-Based Decision Heuristic for Disjunctive Linear Arithmetic

Dan Goldwasser

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE MASTER DEGREE

University of Haifa
Faculty of Social Science
Department of Computer Science

June, 2008

A Theory-Based Decision Heuristic for Disjunctive Linear Arithmetic

By:
Dan Goldwasser

Supervised by:
Dr.Ronen Shaltiel (University of Haifa)
Dr. Ofer Strichman (Technion)
Dr. Shai Fine (IBM, Haifa Research Lab)

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE MASTER DEGREE

University of Haifa
Faculty of Social Science
Department of Computer Science

June, 2008

Approved by: _____ Date: _____
(Supervisor)

Approved by: _____ Date: _____
(Supervisor)

Approved by: _____ Date: _____
(Supervisor)

Approved by: _____ Date: _____
(Chairperson of M.A Committee)

Acknowledgments

This work summarizes an intensive research process, spanning over several years and covering several research fields. The successful result of this effort was possible due to the help and efforts of several people, I would like to take the opportunity to show my appreciation for their efforts.

First and foremost, I would like to thank my thesis advisors Ofer Strichman, Shai Fine and Ronen Shaltiel for the help and guidance they provided throughout this time.

I would especially like to express my deep appreciation and gratitude to Ofer Strichman, whose endless patience and commitment made the completion of this research work possible.

I would like to thank the Department of Computer Science at the University of Haifa, the Caesarea Edmond Benjamin de Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science (C.R.I.) and Akavia family for the help and financial support they provided.

Abstract

This work studies the decision problem of Disjunctive Linear Arithmetic over the Reals from the perspective of computational geometry.
Given a formula, the geometric search space can be defined as the set of d -cells in a *linear hyperplane arrangement* induced by the formula's predicates. We show that traversing this space, rather than the Boolean space as done by current approaches, may have an advantage when the number of variables is smaller than the number of predicates (as it is indeed the case in the standard SMT-Lib benchmarks used for evaluation by the research community). We then continue by showing a branching heuristic that is based on approximating T-implications, based on a geometric analysis. We achieve modest improvement in run time comparing to the commonly used heuristic used by competitive solvers.

Contents

List of Tables

List of Figures

1 Introduction

The Satisfiability problem has long been recognized as one of the fundamental problems of computer science. Phrased simply as deciding whether a given formula has an assignment that makes the formula evaluate to true, this problem poses some of the greatest challenges to both theoretical and applied computer science research communities.

The practical difficulty of solving the satisfiability problem is determined by the logical theory in which the formula is defined; for example the famous propositional satisfiability problem, although perhaps the most famous NP-complete problem, can in practice be efficiently solved in many cases. Other theories cannot be decided efficiently and some are even undecidable. The research field investigating the different satisfiability problems is usually referred to as Satisfiability Modulo Theory (SMT). In recent years considerable progress has been made in this field, much of this progress can be attributed to the SMT-LIB initiative (Satisfiability-Modulo- Theory Library), which coordinates the research effort in this field.

This dissertation discusses Disjunctive Linear Arithmetic, also known as Quantifier-free Linear arithmetic over the reals in the terminology of the SMT community (abbreviated as QF_LRA or LRA for short), which is arguably the most important decidable first-order theory in verification, other than propositional logic, and a subject for research [11] and annual competitions in the SMT community [3].

The problem of deciding LRA can be formally defined as follows – given a Boolean combination of predicates of the form $\sum_{i=1}^{n} a_i \cdot x_i \bowtie a_0$, where $\bowtie \in \{\leq, <, \geq, >, =, \neq\}$ and $a_i, x_i \in \mathcal{R}$, does a satisfying assignment exist?

Most approaches to this problem combine a decision procedure for a conjunction of linear arithmetic predicates (referenced as DP_T) and a SAT solver implementing the DPLL algorithm [7, 8]. The DPLL algorithm can be thought of as a search procedure over a binary tree representing the space of possible assignments; in this tree an internal node represents a partial assignment and a leaf-node represents a full assignment. Given an LRA formula, it is first converted into a propositional formula by replacing every linear predicate with a propositional variable, the new formula is the $Boolean - skeleton$ of the original formula. Although the Boolean skeleton can be solved by a SAT solver, the result can not be directly used as the Boolean skeleton does not capture the dependencies between predicates induced by the underlying theory; these dependencies are incrementally added as the search over the propositional space progresses. The added dependencies are encoded as propositional clauses. Consider first the following basic procedure, implemented in the previous generation of systems, such as CVC [24] and an

early version of MathSat [1]:

1. Encode each predicate with a new propositional variable.
2. Solve the resulting abstract formula with a SAT solver. If it is unsatisfiable – abort and declare the formula unsatisfiable.
3. Otherwise check with DP_T if the assignment, denoted α, is consistent in T. That is, whether the conjunction of the formula's predicates, each in the polarity assigned to it by α, is T-satisfiable. If yes – abort and declare the formula satisfiable.
4. Otherwise, add to the propositional abstraction a *lemma* in the form of a propositional clause, which rules out α (this will force the SAT solver to backtrack and find another assignment).
5. Return to step 2.

This procedure describes a simple protocol between the DPLL search framework and DP_T; the Boolean search continues until a complete assignment satisfying Boolean skeleton is found, only at that point is DP_T applied to validate the assignment.

In recent years a new line of solvers emerged, implementing the DPLL(T) framework. DPLL(T) is a generalization of DPLL for solving a decidable first-order theory T, assuming the existence of a decision procedure DP_T for a conjunction of T predicates. It first appeared in abstract form in a paper by Tinelli [25] and later materialized into the award-winning SMT solver Barcelogic [15]. All competitive SMT solvers, including Yices [11], Z3 [9], ArgoLib [19], MathSAT [1] and CVC-3 [4], to name a few, decide LRA by instantiating the DPLL(T) framework [25, 15] with general simplex, as introduced by Dutertre and de Moura in [11, 12].

DPLL(T) improves the basic procedure presented above in several dimensions. First, it calls DP_T after every partial assignment. This means that it cannot just abort with a 'Satisfiable' answer when DP_T returns TRUE. One possibility is that it would simply return the control to the SAT solver, but instead it applies *theory propagation*, which means that it finds predicates that are implied by the theory. Such predicates are said to be T-implied. For example, if $x = y$ and $y = z$ are two predicates assigned TRUE by the SAT solver, the theory solver can deduce that the predicate $x = z$ must be TRUE as well, and report this information to the SAT solver (assuming such a predicate exists. Typically solvers in this framework refrain from adding new predicates). A more formal description of DPLL(T) is given in Sect. 3.

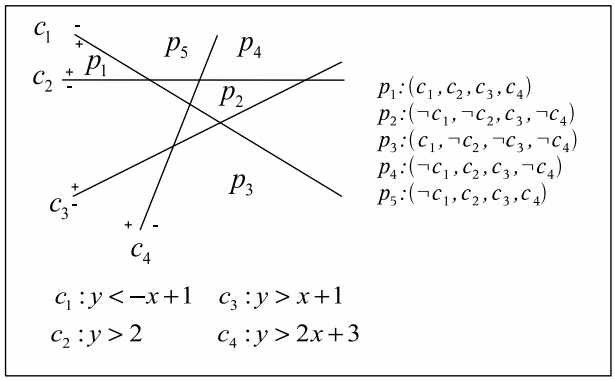


Figure 1: A two dimensional linear arrangement of hyperplanes induced by a set of constraints. Each cell P_i can be mapped to a full assignment of these constraints – a mapping of some of the cells appears on the right side of the figure.

The potential of theory propagation in the case of LRA, that is, the average number of implications given a partial assignment, can be estimated using a theoretical result in computational geometry by Haussler and Welzl [16]. Geometrically, each linear constraint is a hyperplane, and the geometrical representation of these hyperplanes in d dimensions, where d is the number of variables in the input linear system, is called a *hyperplane linear arrangement* [13]. Fig. 1 demonstrates a linear arrangement in two dimensions. Each *cell* (i.e., a convex polytope) in a linear arrangement contains exactly the infinite set of points that evaluate all the linear predicates in the same way. Hence, there is a 1-1 (but not onto) mapping from cells to 2^n, where n is the number of constraints. The number of cells is bounded by $O(n^d)$, which, note, can be much smaller than 2^n. Hence, many combinations of predicates do not correspond to a cell. This is exactly the case that the T-solver declares an assignment as being inconsistent.

In a series of papers, Haussler and Welzl [16] and Clarkson [5], suggested that for a $n \times d$ linear system, if r constraints are randomly selected, such that $d \leq r < n$, and build a linear arrangement, then with high probability – a probability of $(1 - 1/r^c)$, where c is a constant – the interior of any cell in the new arrangement is intersected by at most $O((n \log r)/r)$ of the remaining $n - r$ constraints. (This result refers to the number of variables d as a constant. See [14] for a more detailed explanation, and for an explicit proof of this property).

The relevance of this result to theory propagation is clear: if the current partial assignment is T-consistent, it corresponds to a cell, and the value of any unassigned linear constraint that does not intersect this cell is implied. The value r in our case is simply the number of assigned predicates in the current partial assignment.

Since theory propagation is a measure of efficiency, not of correctness, the question of how much such propagation should be done depends on the efficiency of the algorithm that deduces this information and perhaps also on the investigated formula. In the case of LRA *exhaustive theory propagation*, i.e., learning all possible T-implications, and sometimes even learning one such implication, is not cost-effective.

This work initially suggests an algorithm searching the assignments space defined by an arrangement of the linear constraints appearing in the input formula. This algorithm relies on exhaustive theory propagation for the search to converge quickly. As it can be expected the computational cost of applying exhaustive theory propagation makes this algorithm extremely inefficient compared to the standard DPLL(T) algorithm when applied to SMT benchmarks. In a second approach, we do not attempt to solve this problem, but rather to show a method in which some information from the theory can still be obtained in a cost-effective manner. Specifically, this work presents a method to get *approximated* T-implications and how to integrate them in the solving process without jeopardizing soundness. The approximated information is affecting the decision heuristic: the *decision variable* is still chosen using the SAT solver's normal considerations, while the variable's value is decided using theory related considerations. This is in contrast to the current practice in which decisions are made solely by the SAT solver, and are affected by the theory only indirectly, via the lemmas added by the solver.

The rest of this work is structured as follows – First some preliminary definitions are introduced, followed by a description of the DPLL(T) framework and general simplex in Sect. 3. Section 4 presents a geometric interpretation of the search space and how it can be used for identifying implications that can help the search process. Section 5 describes a search algorithm based on a hierarchical decomposition of the search space. Section 6 describes our method for approximating geometric implication and presents some experimental results for this method. Sect. 7 concludes this work, with some thoughts on possible future uses of the observations presented in the work.

2 Preliminaries

2.1 Geometric Definitions

In this section the geometric terms used throughout this thesis work are defined and explained. The main geometric concepts explained in this section are *Hyperplane Arrangements* and their properties; which are essentially geometric data structures, wildly discussed in the computational geometry field. Informally put, given a finite collection H of hyperplanes in $\mathcal{R}^d$, a hyperplane arrangement is the decomposition induced by H of $\mathcal{R}^d$ into connected cells. A natural measure for the complexity of the arrangement is the number of cells induced by the arrangement. The rest of this paragraph provides a formal definition of hyperplane arrangements.

Definition 1 (Hyperplane) *A hyperplane h is defined by a linear equation of the form $f(x) = b$, where f is a linear mapping $\mathcal{R}^d \to \mathcal{R}$, $x \in \mathcal{R}^d$ and b is a constant.*

Definition 2 (Hyperplane Arrangement) *A hyperplane arrangement A is a collection of hyperplanes in $\mathcal{R}^d$. A hyperplane arrangement induces a decomposition of $\mathcal{R}^d$ into connected cells.*

Definition 3 (Position Vector) *Given an arrangement A composed of a collection of hyperplanes $h_1 ... h_n$, a function u_i mapping a point in $\mathcal{R}^d$ into a relative position in the arrangement is defined as follows:*

$$u_i(p) = \begin{cases} u_i(p) = 1 & p \in h^+ \\ u_i(p) = -1 & p \in h^- \\ u_i(p) = 0 & p \in h \end{cases}$$

Where h^+, h^- denotes the positive and negative half spaces respectively. The vector $u(p) = (u_1(p), ..., u_n(p))$ is called the position vector of a point p.

Definition 4 (Arrangement's Cells) *Two points p and q are considered equivalent if $u(p) = u(q)$. The equivalent class thus defined is called faces or cell of the arrangement A. Each cell is associated with a dimension, which is the difference between the formula's dimension and the number of zero values in its position vector. The term i-cell is used to denote a cell of dimension i.*

Example 1 *Figure 2 depicts the decomposition of $\mathcal{R}^2$ into connected cells using the position vector notation defined above. The vertex v_1 is a cell in the arrangement with a dimension of zero as its position vector contains a number of zero values equal to the arrangement dimension.*

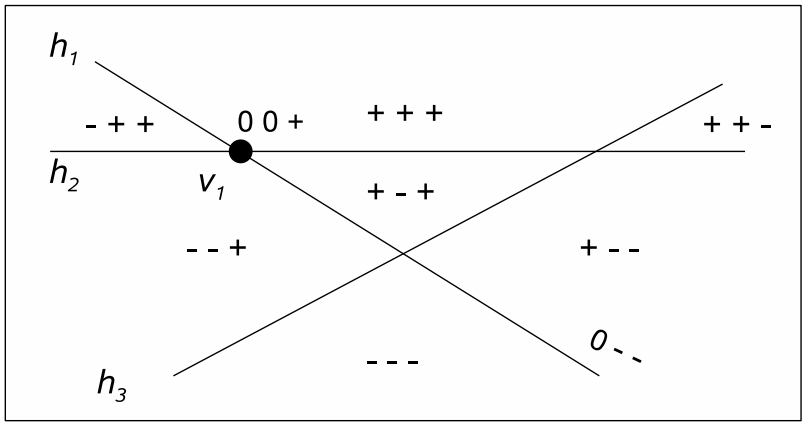


Figure 2: Cells of different dimensions in an arrangement

2.2 Propositional Logic

In this section the basic definitions and terms in propositional logic are defined and explained.

Definition 5 (Literal) *Let P be a fixed finite set of propositional variables. If $p \in P$, then p is an atom and p and $\neg p$ are literals of P.*

Definition 6 (Clause) *A clause C is a disjunction of literals $l_1 \vee l_2 \vee ... l_n$. A unit clause is a clause consisting of a single literal.*

Definition 7 (CNF Formula) *A (CNF) formula is a conjunction of one or more clauses $C_1 \wedge C_2, ..., \wedge C_k$*

Definition 8 (Truth assignment) *A (partial truth) assignment M is a set of literals such that $\forall p \in P$, $\neg((p \in M) \wedge (\neg p \in M))$.*

A literal l is true in M if $l \in M$, is false in M if $\neg l \in M$, and is undefined in M otherwise. The assignment M is total over P if no literal of P is undefined in M, and partial otherwise.

Definition 9 (State of a clause under an assignment) *A clause C is true in M if at least one of its literals is true in M. It is false (also referenced as conflicting) in M if all its literals are false in M, and it is a unit clause if only one of its literals is unassigned in M and it is undefined in M otherwise.*

Definition 10 (Satisfiability under an assignment) *A formula F is true in M, or satisfied by M, if all its clauses are true in M. In that case, M is a model of F. If F has no models then it is unsatisfiable.*

2.3 Propositional Satisfiability

The basic DPLL procedure can be considered as a search through a binary tree spanning all possible assignments to the input formula, in which each internal node is mapped to a partial assignment and the leaves represent a full assignment. The DPLL procedure, for example as presented in Algorithm 1, progresses by choosing an unassigned variable and deciding its value (line 4) , identifying and propagating implications of this decision (line 6) and backtracking in case of a conflict (line 10). Each decision is associated with a decision level, the depth of decision in the branch in the binary tree leading to the decision. The values implied by a decision are also associated with that decision level.

The key procedures used in Algorithm 1 are briefly reviewed in the following paragraph, a detailed description and implementation details of these procedure can be found in [18].

- DECIDE
  This procedure chooses an unassigned variable and a truth value for it. It returns a $false$ value if and only if there are no more variables to assign. There are several heuristics for making these decisions, given the relevance to this work, some heuristics are reviewed in more detail in section 2.3.1.

- BCP
  Values implied by the current assignment can be recognized using the **unit clause rule**- stating that unit clauses must be extended to satisfy the (last) unassigned literal in that clause. This procedure identifies and propagates implied values by a repeated application of the unit clause rule until either a conflict is encountered or there are no more implications. It returns conflict if and only if encountered a conflict. This repeated process is called Boolean Constraint Propagation (BCP). BCP is applied in Line 2 because unary clauses at this stage are unit clauses.

- ANALYZE-CONFLICT
  Modern solvers are *conflict driven* - when a conflict is encountered it is analyzed to prune large search spaces and the search continues by *Backjumping*- which means that several decisions can be undone in a single step. This is done by analyzing the conflict to identify it's cause - the *conflict clause*. This clause is added to the original formula, and the search backjumps to the highest decision level in the conflict clause (other than the current). ANALYZE-CONFLICT is responsible for computing the backtracking level, detecting global unsatisfiability, and adding new constraints on the search in the form of new clauses. It returns the decision level

to which the solver should backtrack to.

Algorithm 1 The DPLL framework.

```
1: function DPLL()
       Input a CNF formula φ;
2:   if BCP () = "conflict" then return "Unsatisfiable";
3:   while (TRUE) do
4:     if ¬DECIDE () then                                  ▷ Full assignment
5:       return "Satisfiable";
6:     while (BCP () = "conflict") do
7:       btrack-level := ANALYZE-CONFLICT ();
8:       if btrack-level < 0 then
9:         return "Unsatisfiable";
10:      else BackTrack(btrack-level);
2
```

2.3.1 Decision Heuristics

The importance of choosing a variable correctly is dramatic as different decision strategies will significantly affect the efficiency of the solver. Many different decision heuristics have been proposed over the years. The early decision heuristics, for example Jeroslow-Wang are essentially greedy algorithms which select variables that will maximize the number of implications or satisfied clauses by using simple computations, such as focusing on literals which appear in short clauses. Other heuristics are state-dependent and count literal occurrences in unresolved clauses, this adds some cost to the decision process. Recent heuristics are state-independent, making the heuristic computationally lighter, and focus the solver on variables appearing in recent conflicts. In the rest of this section the $VSIDS$ heuristic is reviewed.

- Variable State Independent Decaying Sum (VSIDS)
  This heuristic chooses variables which appeared in recent conflicts; each literal is assigned a counter which is increased when that literal appears in a newly added clause. Periodically all the counters are divided by a constant, focusing the heuristic on literal which appeared in recently learnt conflict clauses. Another variation of this heuristic, implemented in solvers such as MiniSAT, assign counters to variables instead of literals, allowing it to select a decision variable while the truth value for the chosen variable is decided on using a different method (usually using a default truth value).

2.4 Propositional Encoding

As previously mentioned, current approaches for deciding LRA construct a propositional abstraction of the input formula, and refine it by incrementally encoding the dependencies imposed by the underlying theory. If these dependencies are encoded exhaustively before attempting to decide the satisfiability of the formula, the encoding scheme is said to be *Eager*, if the dependencies are encoded throughout the search then the encoding scheme is said to be *Lazy*. Current decision procedures, including the state-of-the-art DPLL(T) method presented in the following section, use the lazy encoding scheme. The rest of this paragraph defines formally this encoding scheme.

Definition 11 *Given a theory predicate p appearing in an LRA formula t, p is associated with a unique Boolean variable $e(p)$ called the* Boolean encoder *of p. $e(t)$ denotes the Boolean formula resulting from substituting all the theory predicates appearing in the formula with their Boolean encoders. $e(t)$ is known as the propositional (or Boolean) skeleton of t.*

3 The DPLL(T) Framework

State-of-the-art SMT solvers follow the DPLL(T) framework [25]. The components of the algorithm are those of DPLL and a decision procedure DP_T for a conjunctive fragment of a theory T, such as the generalized Simplex algorithm presented in the following section. The name DPLL(T) emphasizes that this is a framework that can be instantiated with a different theory T and a corresponding decision procedure. In the version of DPLL(T) presented in Algorithm 2 (see also Fig. 3, which is copied from [18]), a procedure called DEDUCTION is invoked in line 13 after no more implications can be made by BCP. DEDUCTION performs theory propagation: it finds T-implied literals and communicates them to the DPLL part of the solver in the form of a constraint t, also called a *lemma*. Hence, in addition to implications in the Boolean domain, there are also implications due to the theory T.

What are the restrictions on these new clauses? They have to be implied by the input formula φ and restricted to the atoms in φ (or some finite superset thereof). Let α denote the current assignment and $\hat{T}h(\alpha)$ the conjunction of T-literals corresponding to this assignment. If $\hat{T}h(\alpha)$ is unsatisfiable, the lemma $e(t)$ (where $e(t)$ denotes t after each predicate is replaced with its propositional encoder) has to block α. If $\hat{T}h(\alpha)$ is satisfiable, t is required to fulfill one of the following two conditions in order to guarantee termination:

1. The clause $e(t)$ is an asserting clause under α. This implies that the addition of $e(t)$ to the current propositional formula and a call to BCP leads to an assignment to the encoder of some literal.

2. When DEDUCTION cannot find an asserting clause t as defined above, t and $e(t)$ are equivalent to TRUE.

The second case occurs, for example, when all the Boolean variables are already assigned, and thus the formula is found to be satisfiable. In this case, the condition in line 15 is met and the procedure continues from line 5, where DECIDE is called again. Since all variables are already assigned, the procedure returns "Satisfiable".

As previously mentioned in the introduction, theory propagation has no influence on correctness, rather only on efficiency, and therefore the question of how much to infer on the theory side and propagate depends on the theory and the benchmark set. It turns out, empirically, that exhaustive theory propagation in the case of LRA is not cost-effective (see, e.g., [12]). Moreover, even checking for consistency of the current partial assignment is too costly in practice. Instead, competitive solvers only do light-

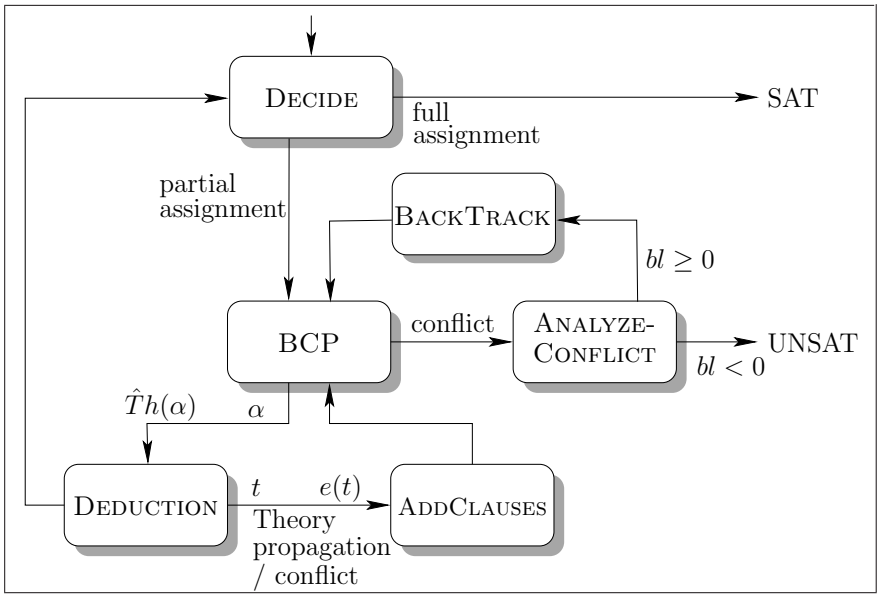


Figure 3: The main components of DPLL(T). Theory propagation is implemented in DEDUCTION.

weight theory propagation and defer the consistency check to when a full assignment is achieved, as will be described later in Sect. 3.1.

There are other variations to DPLL(T) that are used in competitive solvers, including procedures for strengthening the lemmas and more aggressive invocations of DP_T (after every partial assignment rather than only after BCP). These optimizations are not relevant to the current work, however.

3.1 General Simplex

The standard de-facto decision procedure DP_T for the conjunctive fragment of linear arithmetic over the reals is *general simplex*, as was introduced in [11]. This procedure determines the satisfiability of a conjunction of linear constraints (hence, unlike the original simplex, it does not aim to optimize the value of a linear objective function). General simplex is now implemented in most competitive SMT solvers due to its superior performance in the context of SMT.

Let $A\vec{x} \leq \vec{b}$ be the input linear system, where A is a $n \times d$ coefficient matrix, $\vec{x}$ is a vector of d variables, and $\vec{b}$ is a vector of constants. General simplex begins by transforming this system into *general form*, which consists of two types of linear constraints: equalities of the form $\sum_i a_i x_i = 0$ and constrains of the form $x_i \geq l_i$ or $x_i \leq u_i$, where l_i and u_i are constants. The transformation is done as follows: given a constraint of the

Algorithm 2 The DPLL(T) framework.

```
1: function DPLL(T)
2:     AddClauses (cnf(e(φ)));
3:     if BCP () = "conflict" then return "Unsatisfiable";
4:     while (true) do
5:         if ¬Decide () then                          ▷ Full assignment
6:             return "Satisfiable";
7:         repeat
8:             while (BCP () = "conflict") do
9:                 btrack-level := Analyze-Conflict ();
10:                if btrack-level < 0 then
11:                    return "Unsatisfiable";
12:                else BackTrack(btrack-level);
13:            t :=Deduction (Th(α));
14:            AddClauses (e(t));
15:        until t ≡ true
```

form $\Sigma a_{ij} x_j \leq b_i$, it replaces it with the two constraints

$$\begin{aligned} &\textstyle\sum a_{ij} x_j - s_i = 0 \\ &s_i \leq b_i \ , \end{aligned}$$

where s_i is a new variable, which is called a *bound variable*. The new bound variables constitute the initial set of what is called the *basic* variables, whereas the other variables constitute the initial set of *nonbasic* variables. The basic variables are also called the *dependent* variables, reflecting the fact that their value is determined by the values of the nonbasic variables. The partitioning of the variables to these two sets change throughout the algorithm.

In addition to these two sets, the algorithm also maintains an assignment β to all variables. Two invariants are maintained during the run of the algorithm:

1. The assignment β satisfies all equalities (i.e., it satisfies $A\vec{x} = 0$), and
2. β satisfies those *bound variables* (the new s_i variables) that are currently in the nonbasic set.

Initially the assignment is 0 to all variables. This satisfies the first invariant trivially, and the second one because all the bound variables are basic. Then, the algorithm searches for a basic variable that violates one of its bounds. If there is no such variable the instance is declared satisfiable, since the current assignment satisfies both the equations and all the bound variables. Otherwise, suppose that the assignment to the basic variable

x_i violates its upper bound, and hence has to be reduced. Simplex searches for a nonbasic variable with a positive coefficient that its current value is higher than its lower bound (or such a variable with a negative coefficient that its current value is lower that its upper bound). If there is such a variable x_j, it means that the value of x_i can be reduced by changing the value of x_j. If not – the instance is declared unsatisfiable. Suppose that there exists such a variable x_j (which is then said to be *suitable*). The next step is to change the current assignment and perform *pivoting*, which is essentially the same operation that is done in Gaussian elimination. Pivoting between these two variables means that they exchange places (x_i becomes nonbasic whereas x_j becomes basic), the coefficient matrix is updated accordingly, the assignment to x_i is reduced to meet its upper bound, and the assignment to the other variables are updated so the first invariant is maintained. More details about the pivot operation can be found in [11]. Through a series of such pivoting operations simplex updates its assignment β until it satisfies the input linear system, or declares the system unsatisfiable. Our method uses the assignment β and the pivot operations to approximate T-implications, as will be described later on.

Pseudocode for general simplex appears in Alg. 3. In Fig. 4, assuming the system comprises a conjunction of the predicates $c_1, \ldots, c_5$, general simplex's initial assignment corresponds to the origin (0 to all variables), which is marked as v_1 in the figure. As more pivoting operations take place the assignment is updated, and the points move closer to the target cell P_1.

Algorithm 3 General Simplex

1: **function** General Simplex
2: Transform the system into the general form
Ax=0 and $\bigwedge_{i=0}^{m} l_i \leq s_i \leq u_i$
3: Set B to the set of additional variables $s_1, ..., s_m$
4: Construct a tableau for A
5: Determine a fixed order on the variables
6: If there is no basic variable that violates its bounds, returns "Satisfiable"
Otherwise, let x_i be the first basic variable in the order that violates its bounds
7: Search for the first suitable nonbasic variable x_j in the order for pivoting with x_i.
If there is no such variable, return "Unsatisfiable".
8: Perform the pivot operation on x_i and x_j.
9: Go to step 5.

Recall that in the context of DPLL(T) the linear solver is used incrementally: linear predicates are added or erased as the search progresses. While for most theories, competitive implementations of DPLL(T) check for T-consistency of every partial assignment (and perform theory propagation as described in Sect. 1), this is not cost-effective in

the case of LRA, at least as long as no better alternative to general-simplex is found. Instead, competitive solvers use a lightweight checking procedure called ASSERT – see Alg. 4. This procedure can only detect inconsistencies of bounds, for example if both $x_i \leq 5$ and $x_i \geq 6$ are asserted. In addition it updates the assignment of nonbasic variables so the second invariant is maintained.

Algorithm 4 Procedure Assert-Upper detects simple T-inconsistencies in the current assignment to the predicates, and maintains an assignment which satisfies the bounds of the nonbasic variable.

```
1: function ASSERT UPPER (x_i < c_i)
2:     if c_i ≥ u_i then return "satisfiable";
3:     if c_i < l_i  then return "unsatisfiable";
4:     u_i := c_i;
5:     if x_i is a nonbasic variable and β(x_i) > c_i then
6:         update-assignment(x_i, c_i);
```

4 Geometric Representation

4.1 Representing the Geometric Search Space

Given a linear arithmetic formula φ, denoted by $C(\varphi)$ the set of linear predicates appearing in φ. Each linear constraint $c \in C(\varphi)$ is represented as a hyperplane in R^d, partitioning R^d into two halfspaces: in c^+ all points satisfy c, and in c^- all points do not satisfy c. An intersection of $C(\varphi)$ halfspaces form cells, which are convex regions in R^d. As was seen in Sect. 1 these cells can be mapped into an assignment to the predicates in $C(\varphi)$.

For example, consider the cell marked P_1 in Fig. 4. This region is the intersection of the positive halfspaces of φ's constraints and hence corresponds to the assignment (c_1, c_2, c_3, c_4).

The space of feasible assignments to the predicates can be described with a hyperplane linear arrangement, which is a well known data structure that is used in computational geometry. An arrangement captures the decomposition of a d-dimensional space into connected cells, induced by a set of hyperplanes in R^d. Each cell in the hyperplane arrangement is associated with a dimension: vertices (i.e., hyperplane intersection points) have a dimension of zero, while the convex regions formed by the intersection of halfspaces have a dimension equal to the total number of variables appearing in $C(\varphi)$.

The number of cells is bounded by $O(n^d)$, where $n = |C(\varphi)|$ and d is the total number of distinct variables appearing in $C(\varphi)$.[1] This implies that the complexity of enumerating theory-consistent assignments is exponential in the number of dimensions whereas the complexity of enumerating values in the Boolean space is exponential in the number of constraints.[2] The difference in the two spaces plays a crucial role during the DPLL(T) search: the greater the ratio is, the greater the chance that a propositional assignment is inconsistent in T (in other words, it does not correspond to any cell in the arrangement).[3] Since this difference depends on the values of d and n, These values were checked in various SMT-LIB benchmarks – see Table 1. The results show that the number of predicates is greater than the dimension, hence the linear search space is smaller than the propositional one in these benchmarks.

[1] The 'O' notation is not precise here, because the constant actually depends on d. The convention used by Halperin in [17] is followed here, which used this convention based on an assumption that d is small relatively to n. The bound is in fact $\sum_{i<d} \binom{n}{d-i}$.

[2] possible implications of this gap in complexity are discussed in Sect. 7.

[3] As a result one may even tune the search procedure according to this ratio.

Table 1: Proportion of predicates Vs. variables (dimensions) in the SMT-LIB benchmarks

Benchmark	Predicates:Dimension
QF RDL SCHEDULING	10.9:1
QF RDL SAL	6.7:1
QF LRA SC	3.9:1
QF LRA START UP	6.9:1
QF LRA UART	6.1:1
QF LRA CLOCK SYNCH	3.3:1
QF LRA SPIDER BENCHMARKS	3.2 :1
QF LRA SAL	6.1:1
MathSAT benchmarks (difference logic)	44.5:1
SEP benchmarks (difference logic)	17:1

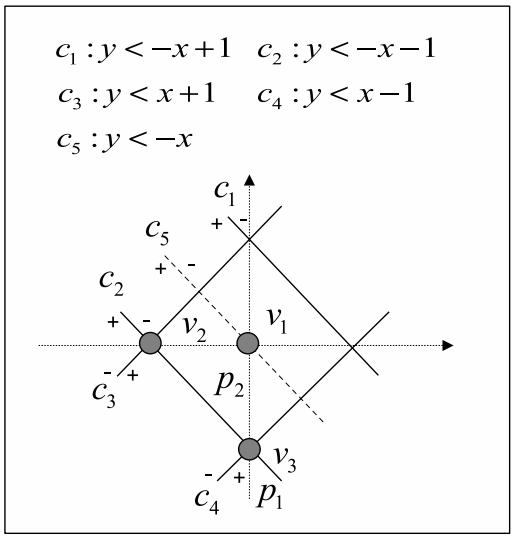


Figure 4: A linear arrangement corresponding to five linear constraints. The axes and the points v_1, v_2, v_3 are not part of the arrangement, but useful for understanding the progress of simplex given the same constraints. The points v_1, v_2, v_3 represent a possible progress of the assignment maintained by simplex. v_1 is the initial assignment, which is always the origin.

4.2 Geometric Representation of T-Implications

A partial propositional assignment is *T-consistent* if it can be mapped into a cell and *T-inconsistent* otherwise. Viewed geometrically, the value of an unassigned predicate p is *T-implied* by a partial assignment, if the cell induced by the partial assignment is contained within any of the halfspaces defined by p.

Example 2 *In Fig. 4 the subset (c_2, c_4) is T-consistent and forms the cell P_1. If c_1 is the current decision variable, deciding on $\neg c_1$ would lead to a conflict, expressed geometrically as an empty intersection of c_1^- and P_1. Hence, c_1 is T-implied by the partial assignment. Indeed, P_1 is completely contained within one of c_1's halfspaces.*

Now consider P_2 as the current partial assignment. The value of c_5 is not implied as both its halfspaces have a non-empty intersection with P_2.

As the DPLL search advances and the partial assignment grows (i.e., more linear constraints are asserted), more values are likely to be T-implied. The reason is that more predicates imply a smaller cell (or even an empty cell if the partial assignment is T-inconsistent), and hence the chances of an unassigned predicate to intersect this cell is smaller. This observation was tested empirically: Fig. 5 describes the ratio between the partial assignment size and the number of predicates implied by it for two benchmarks. The number of T-implications was measured by randomly selecting 100 different partial propositional assignments of equal size and averaging the number of T-implied values by each such partial assignment. Indeed, it is clear that the probability of an unassigned predicate to be T-implied grows with the partial propositional assignment.

4.3 Identifying T-Implications

There are two natural ways to identify T-implications that can be thought of. Given a system of constraints S corresponding to the current partial assignment and an unassigned predicate p, the first method (called *plunging* in [10]) is to solve S together with the negation of p. If the system is unsatisfiable, it means that S implies p. This is a generic method that is relevant for all decidable theories. The second method is to consider the vertices of the cell corresponding to S: if they fall on both halfspaces of p, then the value of p is not T-implied.[4] For example in Fig. 4 the vertices of the cell P_2 fall on both halfspaces of c_5.

[4]This can be applied directly only to closed cells. For open cells a different test has to be made, such as plunging.

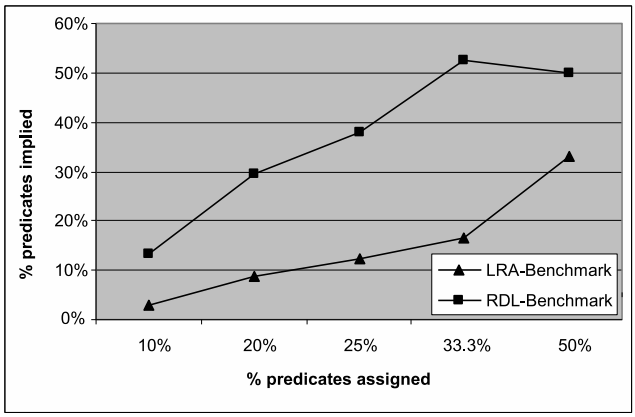


Figure 5: The proportion of assigned vs. implied values at different points during the search. Two benchmarks were checked: a Linear Arithmetic (LRA) benchmark – invar.induct – containing 633 constraints and 163 variables, and a Difference Logic (RDL) benchmark – abz5_1000.smt – containing 1011 constraints and 102 variables.

These methods are too expensive in practice, the first because it corresponds to solving a full linear system for each predicate, and the second because there can be an exponential number of vertices to consider for each cell. In the following two sections we present two approaches exploiting the geometrical interpretation of the search space: the first, described in section 5 presents an algorithm based on a hierarchical decomposition of the search space and employs exhaustive theory propagation. This algorithm uses plunging to exhaustively identify implications, and as can be expected, it does not perform well in practice. The second approach recognizes the computational cost of performing T-propagation and performs a cheap approximation the geometric implications instead.

5 Deciding Disjunctive Linear Arithmetic by Hyperplane Arrangement Traversal

5.1 Introduction

Our initial attempt to exploit the combinatorial properties of the geometric search space is to traverse this space instead of the Boolean search space when looking for a satisfying assignment. As previously mentioned, this space corresponds to the set of d-cells in a hyperplane arrangement induced by the formulas constraints. A clear advantage of this approach is the theoretical bound on the number of such cells, which is known to be $O(n^d)$, which is considerably smaller than the Boolean search space if $d < n$ as is indeed the case in SMT benchmarks. However, for many of these benchmarks a straight-forward construction is unrealistic as the number of variables appearing in the formula (or the *dimension* of the formula) is large. A potential approach that can help alleviate this problem is to use an incremental construction of the hyperplane arrangement, in which only a subset of the linear predicates are chosen at each step, and a partial arrangement is constructed; each cell in this arrangement corresponds to a partial assignment to the formulas Boolean skeleton. Using the partial assignment new values are implied by both the geometric and the propositional structure of the formula. Recalling the theoretical results by Welzl [16] and Clarkson [5] mentioned in Section 1, it can be expected that the number of implications would be considerable.

5.2 Cell Traversal Algorithm

This method is described in detail in Algorithm 5 (and in Fig. 6). The algorithm is applied recursively, where at each recursion level an arrangement is constructed, in which all the cells should have a non empty intersection with a cell chosen in the former level of the recursion. Phrased differently, the algorithm advances by selecting a cell, identifying hyperplanes intersecting that cell (i.e., the set of hyperplanes whose values are not implied by the current assignment) and decomposing the cell using a subset of these hyperplanes. In case the chosen cell conflicts with the Boolean skeleton of the formula, a new cell is chosen from the arrangement.

The AddImplications procedure described in Algorithm 6 describes the interplay between Boolean and geometric constraint propagation procedure, which are consecutively executed until the process converges and no new values are added, or until a conflict is encountered. Viewed geometrically, the geometric constraint propagation procedure identify implied values of constraints that lie outside of the cell defined by the current

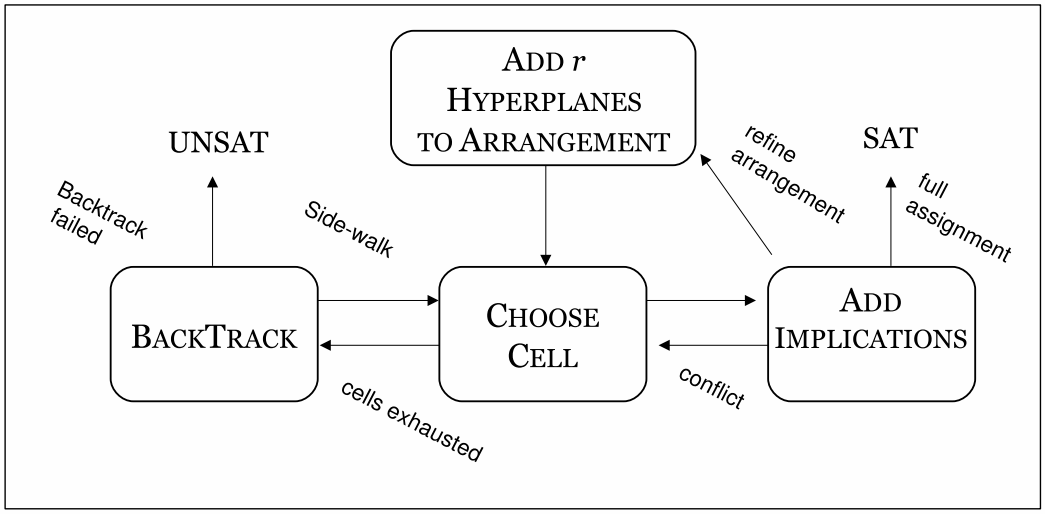


Figure 6: The main components of the Cell-traversal algorithm

assignment, while the Boolean constraint propagation procedure identifies implied values of constraints that lie inside the cell (it may also identify values of constraints outside the cell if applied before the geometric procedure). Given the considerable computational cost of the geometric procedure, in our implementation we always apply BCP exhaustively before invoking the geometric procedure.

Algorithm 5 Cell Traversal Algorithm

1: **function** SOLVE($\psi, assigned$)
2: **if** FULLASSIGNMENT () **then return** "SAT"
3: choose r unassigned linear predicates randomly and construct $A(H_r)$
4: **while** (remains an unvisited cell in $A(H_r)$) **do**
5: Select an unvisited *cell* in $A(H_r)$
6: **if** ADDIMPLICATIONS (*cell*) $\neq$ "Conflict" **then return** SOLVE ($\psi, assigned \cup cell$)
7: **return** "UNSAT"

Example 3 *Fig. 7 describes several stages in the search process defined by Algorithm 5, the stages (and figures) are ordered from left to right, and referenced throughout this example in this order.*
Consider a set $l_1, ..., l_8$ of linear constraints, and Boolean skeleton defined over these constraints -

Algorithm 6 Add Implications Algorithm

```
1: function AddImplications(cell)
2:     cell_reference = φ
3:     while (cell_reference ≠ cell) do
4:         cell_reference = cell
5:         if AddGeometricImplications () = "Conflict" then return "Conflict"
6:         if BCP () = "Conflict" then return "Conflict"
```

$$(\neg e(l_1) \vee e(l_2)) \wedge (e(l_1) \vee e(l_2)) \wedge$$
$$(e(l_3) \vee e(l_4)) \wedge (e(l_5) \vee e(l_2)) \wedge (e(l_4) \vee e(l_7)) \wedge$$
$$(e(l_4) \vee e(l_6) \vee \neg e(l_8)) \wedge (e(l_8) \vee e(l_1))$$

In the first stage, appearing in Fig. 7.1, a subset of the constraints are chosen $\{l_5, l_6, l_7\}$ *and partial arrangement is constructed. Then in Fig. 7.2 , a cell is chosen, corresponding to the partial assignment -*$\{l_5 = true, l_6 = false, l_7 = true\}$*. This assignment also* geometrically *implies an assignment to the following constraints–* $\{ l_2 = false, l_3 = true, l_4 = false, l_8 = false \}$ *as these constraints do not intersect that cell, their value can be determined. This assignment, while consistent with the theory, conflicts with the Boolean structure of the formula as it forces* l_1 *to accept contradicting values to satisfy the first two clauses in the Boolean skeleton. The algorithm therefore moves to a new cell in the arrangement (Fig. 7.3), corresponding to the assignment -* $\{l_5 = false, l_6 = true, l_7 = true\}$*. Using the* AddImplications *procedure in the next step (Fig. 7.4) identifies the values implied by the current assignment. These values can be implied by the theory, i.e., the values of hyperplanes not intersecting this cell, or by applying BCP. In this case the value implied by appplying BCP,* $l_2 = true$*,is of a hyperplane intersecting the cell.*

At this point a new recursive step is taken, and the algorithm constructs a new arrangement from the remaining predicates, which in this case consists only of l_8 *(Fig. 7.5). Finally, the algorithm selects a cell in that arrangement, validates that it does not conflict with the Boolean skeleton, and terminates the search with a SAT value (Fig. 7.6).*

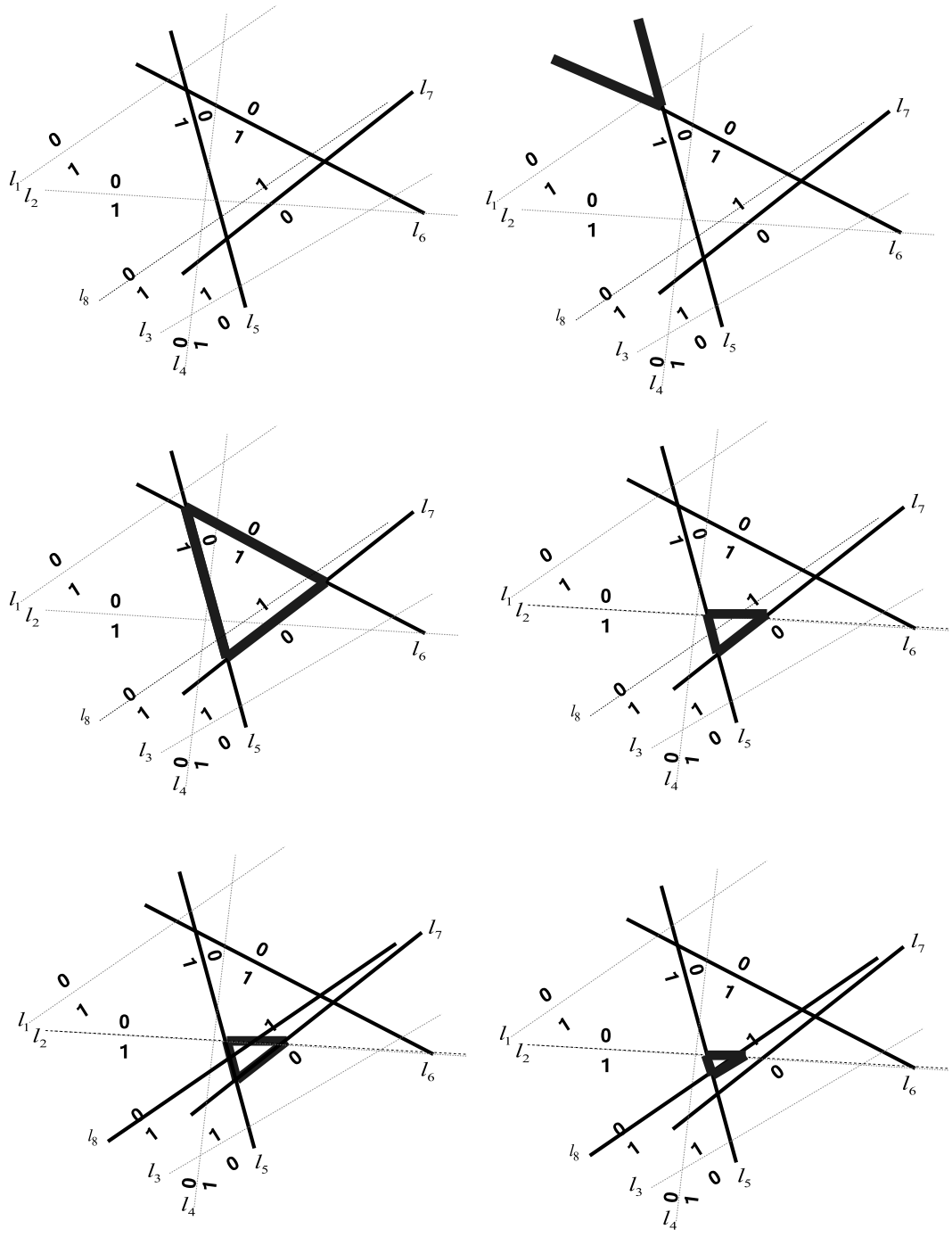


Figure 7: Different stages in the search algorithm, referenced as 7.1- 7.6 ordered top to bottom, left to right. Bold lines represent the current assignment, thick bold lines are the boundaries of the actual cell induced by the assignment.

5.3 Experimental Results and Analysis

Benchmarks originating from real-world applications typically have a high dimension, making an explicit construction of the hyperplane arrangement unrealistic in practice. In our implementation we used an incremental construction in which only a single hyperplane was added to the arrangement at each stage; since we applied exhaustive theory propagation, any hyperplane not intersecting the cell induced by the current assignment was assigned by the AddImplications procedure. Any remaining unassigned linear predicates at this point is known to intersect the current cell and therefore both values can be assigned to this predicate, thus enabling a symbolic representation of the search space simply by storing the current assignment. This approach closely resembles the DPLL(T) approach with exhaustive theory propagation, in which all the DEDUCTION procedure described in Algorithm 2, is continuously applied until all possible values implied by the theory are identified. This approach has been shown to be ineffective in other studies. The results of our testing also confirm this result. We evaluated this method over a benchmark set taken from the LRA spider benchmarks suite and compared the results to the results obtained by using ArgoLib, a DPLL(T) solver over the same set of benchmarks. ArgoLib outperformed our system by two orders of magnitude - solving the benchmark suite in 14.83 seconds while it took our system 4430.75 seconds to solve it.

6 Encoding Geometric Clues in the DPLL(T) Framework

In this section we present a new approach for using geometric information. Unlike Algorithm 5 which uses exact information and suffers from the added computational cost of obtaining it, this approach tries to approximate this information at a much lower cost.

6.1 Approximating T-Implications

A possible way to alleviate the computational problem of obtaining geometric information is to approximate it; using exact geometric information essentially requires enumerating the vertices of the cell induced by the partial assignment (plunging in the case of LRA uses a simplex based solver which can be considered in a worst-case analysis as equivalent to enumerating the cell's vertices). In this section we describe an approach that approximates implications using a single vertex, identified using the geometric representation of the T-solver's state, and hence does not incur the computational cost of solving a linear system.

Using approximations, however, prevents us from using this information to identify implications as described in the DPLL(T) framework, since an approximated implication may affect the soundness of the algorithm; The use of such information needs to be restricted, allowing the DPLL search to recover in case the value was not in fact implied. The approach presented in this work solves this problem by using this information as 'hints' to the decision heuristic as to the value of the current decision variable. In other words, it only uses this information to affect the decision, not to create new implications. The choice of decision variable is still made by the SAT solver, but when the T-solver has an approximated estimation of the value of this variable, it passes this information to the DPLL solver which then assigns it to the decision variable. Hence, wrong information results in slower solving, not incorrect result. How can T-implications be approximated? It is possible, for example, to generate a small number of points inside the cell corresponding to the current partial assignment (or better of, a small number of vertices of this cell), and then guess the value of an unassigned predicate according to the halfspace of the predicate in which these points fall. If they fall on both sides, the decision on the value can be made by the SAT solver. For example, consider once again the constraint c_5 and the region P_1 appearing in Fig. 4. The partial assignment $(c_1, ..., c_4)$, which corresponds to P_1 implies $c_5 = true$. But identifying this value for c_5 can be done by generating only one of P_1's vertices and checking whether it falls in the

positive or negative halfspace of c_5.

An obvious requirement is the ability to generate such a point with little computational cost. Unfortunately, generating points which are known to fall within a cell defined by the intersection of constraints proved to be a non-trivial issue.[5]

The method our system uses is simple but not very accurate. It relies on the assignment β that is maintained by simplex. Recall that due to efficiency considerations, in competitive DPLL(T) solvers simplex is not fully invoked after each partial (propositional) assignment, and rather only the ASSERT procedure (Alg. 4) is invoked. This means that β does not necessarily correspond to a point in the cell associated with the current partial assignment. It is also possible that there is no such cell at all, if the current assignment is T-inconsistent. Thus, although using the assignment adds no additional complexity, it can only be used, as before, to approximate implications rather than infer them.

An attempt to improve this approximation can be made by trying to make β more accurate. This can be done by invoking the pivot operation for some bounded number of times k, or until a definite conclusion is reached (i.e., β is in the cell or the current system is T-inconsistent). Thus, k is an *accuracy parameter*. However, additional pivot operations can also make the number of satisfied constraints go down, since the pivot operation is not monotonic in this sense. Hence our implementation takes the assignment β that satisfies the largest number of bound predicates along the way.

Algorithm 7 describes the heuristic – given a partial assignment α and an unassigned predicate c, the heuristic searches for an assignment to the linear variables that maximizes the number of satisfied bound constraints, by performing pivoting k times and checking if the number of satisfied predicates increased after each time. The point defined by the assignment is then used to set the value of c: if the point falls in the positive halfspace, p is assigned TRUE and FALSE otherwise.

[5] One of the methods we tried for generating such points was to randomly select r constraints out of the partial assignment and identify the point of intersection of these constrains using Gaussian Elimination. The problem is that unless r is large, most of the points generated in this manner are bound to fall outside the target cell, which makes this method inefficient in practice. Another approach, also computationally expansive, can be to solve a linear programming problem and use the solution as our point.

Algorithm 7 Theory based decision heuristic.

Input: A partial (propositional) assignment α, unassigned linear predicate p, accuracy parameter k

```
1: procedure DetermineValue
2:     for i:=0 to k do
3:         Pivot();
4:         if #SatBounds(β_i) > #SatBounds(β_{i-1}) then
5:             point= β_i ;
6:     if point ∈ p^+ then return True
7:     else return False
```

6.2 Taking Decisions at T-Inconsistent Points

Recall that the current partial assignment is not necessarily T-consistent because in practice most solvers perform a complete T-consistency checks at selected intervals or even only when the assignment is full. This is the strategy of the SMT solver Argolib [19], on top of which the heuristic was implemented.

The ratio of times that decisions are taken when the partial assignment is T-consistent was approximated empirically. The larger this number is, the less our heuristic will be affected by this problem. This number was evaluated by running an external T-solver for validating the partial assignment at each decision point. It should be noted that the outcome of this validation was ignored, and in no way affected the operation of the solver. This experiment was run for each of the benchmarks that we used for evaluation, as described in Sect. 6.3. The average proportion of decisions taken when the partial assignment was T-consistent across all benchmarks was 0.78.

6.3 Evaluation

We tested the performance of our approach on the LRA benchmarks that were used in SMT-COMP'07. We set the timeout to 30 minutes for each benchmark. Our implementation is based on the open-source solver ArgoLib. This tool is based on DPLL(T) and the general simplex algorithm for solving linear arithmetic.

ArgoLib's original decision heuristic is the same as in MiniSAT: it selects a decision *variable* using a VSIDS-like [21] heuristic and sets its value to FALSE. Also implemented and evaluated was a system which uses a decision heuristic in which a decision variable is assigned its last value if it was assigned before, and FALSE otherwise. Such a heuristic was used in CSP solvers [2], introduced to SAT in [23] and lately adopted by RSAT under

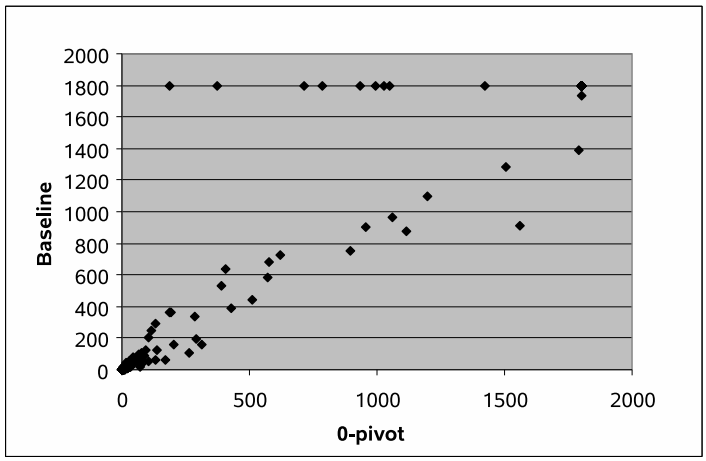


Figure 8: Baseline vs. 0-pivot (run-time).

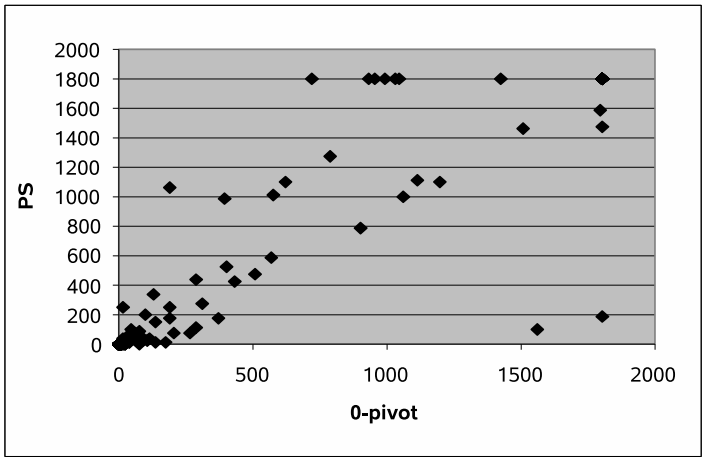


Figure 9: PS vs. 0-pivot (run-time).

Table 2: Overall results. The results show the number of instances solved correctly in less than 30 minutes and the overall time spent by the solver.

	Baseline	**PS**	**0-pivot**
Score	172	175	180
Total time	72229.8	68656.04	64887.43
	2-pivot	**4-pivot**	**14 -pivot**
Score	177	174	173
Total time	70491.21	75261.75	84125.97

the name 'progress saving' (PS) [22]. This heuristic is referenced as the PS strategy throughout this work.

Several variants of our approach were evaluated by choosing different values of the accuracy parameter k. Performance was measured in terms of the total number of instances solved correctly before time-out and the overall running time.

Table 2 summarizes the results for six different strategies: *Baseline* and *PS* correspond to the original ArgoLib's heuristic and its enhancement with the PS heuristic as define above. Both of these heuristics do not use the theory solver directly to make a decision. The strategy *k-pivot* refers to our heuristic where k pivoting steps are made before deciding on a value.

The table shows that our heuristic somewhat outperforms the baseline, regardless of the number of pivoting operations performed. Increasing k turns out to be not cost-effective (there were several benchmarks, however, that it improved run-time). Overall the system achieved a modest improvement of 11% in run time and 5% in the number of solved instances over all benchmarks.[6] There was no significant difference between the satisfiable and unsatisfiable instances.

Figures 8–9 show detailed results when comparing 0-pivot with the two SAT-based heuristics. From Fig. 8 it is evident that with minor exceptions, the 0-pivot heuristic has a comparable performance with the baseline heuristic, but it is able to solve most of the instances which cannot be solved by the baseline system, which means that it is more robust. Figure 9 shows that PS, although fails less than the baseline heuristic, still fails in many cases that can be solved with the 0-pivot strategy. We also checked the accuracy of our approximation. For the first 100 benchmarks in the SMT-LIB benchmark set[7],

[6]Since the improvement is small, one may wonder if a random choice of the value cannot compete with such results. When running such a test, the result was considerably worse than the baseline: 78070.6 sec and 172 solved instances.

[7]`http://www.smtcomp.org/2007/benchmarks/QF_LRA.tar.gz`

the following data was measured, with a time out of 30 min:

1. Total number of decisions: 710782.

2. Number of decisions resulting in T-inconsistency: 299130 (42% of partial assignments). This is much higher than the average of 22% reported in the previous section, which can be attributed to the different selection of benchmarks.

3. Number of implied decisions (checked with 'plunging'): 19799 (4.8% of T-consistent partial assignments). There are two possible reasons for this particularly small number in comparison to Fig. 5: First, it may indicate that most decisions are made in a very small decision level and that relatively few predicates are implied with BCP. Such a scenario leads to small partial assignments, and hence a small number of T-implications, when most of the decisions are made. Second, here the statistics refer to one variable per decision – the variable that was chosen by the SAT solver due to propositional considerations, whereas the statistics in Fig. 5 refer to all unassigned variables. It is possible that lemmas bias the SAT solvers's decision variable towards those that are not implied.

4. Number of times k-pivot approximation with $k = 0$ lead to a correct implication, when the partial assignment is consistent: 14447 (72% of T-implications).

5. Number of times k-pivot approximation with $k = 22$ lead to a correct implication, when the partial assignment is consistent: 14456 (73% of T-implications). This result is very surprising: even after that many pivot operations, the improvement in accuracy is very marginal. This explains why $k = 0$ is the best, empirically.

6. Number of times the value chosen by the baseline algorithm (simply FALSE) lead to a correct implication, when the partial assignment is consistent: 12557 (63.4% of T-implications).

There are two main points to observe: first, that the 0-pivot strategy increases the accuracy of the decision from 63.4% to 72% (and, recall, it does so with almost 0 cost). Second, that the fact that in less than 5% of the cases there was an implied value, may possibly indicate that the 0-pivot strategy is also helpful when the decided value is not T-implied. We can speculate why this happens when the instance is satisfiable: the predicate partitions the cell into two parts which are most likely not even. The chosen point has a higher probability to be in the bigger of the two parts, and there is a higher probability that the solution resides in that part.

7 Discussion and Future work

The improvement in run time that was showen is very modest. Yet there are several aspects in this work that are novel and may lead to future research:

1. As far as we know this is the first work that studies the problem of deciding disjunctive linear arithmetic from the perspective of computational geometry;

2. It is the first work in the context of DPLL(T) that lets the theory guide the Boolean search directly, i.e., not by adding new clauses;

3. It is the first work in this context that considers the problem of using conjectured information without losing soundness.

As discussed in Sect. 4.1, the number of cells is exponential in the number of variables, whereas the number of Boolean assignments is exponential in the number of predicates. Since the former is typically much smaller than the latter (see Table 1), it raises the question whether there is a way to build an efficient SMT solver that exploits this fact. An explicit traversal of the linear arrangement does not seem a reasonable direction, but perhaps there is a way to represent the cells in an arrangement symbolically with a function – this would enable us to build a solver in which the theory leads the search rather than the SAT solver. In other words, in the current DPLL(T) framework the SAT solver leads the search: SAT suggests an assignment, and the theory solver checks it. Also, only the SAT solver can declare the formula unsatisfiable. This will change if we can find a method in which the linear space is traversed rather than the Boolean one. This will open a new research direction of finding decision heuristics in the linear domain, i.e., choosing which cell should be traversed next.[8]

[8] A possible direction is in line with [14], which presents a decision procedure for LRA that is led by the theory side. Given a CNF-style formula, they check if its negation – in DNF – is valid. Each term in this DNF represents a polygon, and checking whether the whole formula is valid corresponds to checking whether the union of these polygons covers R^d. Doing so efficiently is the subject of the above reference: they consider the polygons one at a time in a random order, and maintain their union incrementally. Perhaps such an approach can also be used in a DPLL(T) style algorithm led by the theory.

References

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. 18th International Conference on Automated Deduction (CADE'02)*, 2002.

[2] A. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, Univ. of Oregon, 1995.

[3] C. Barrett, L. de Moura, and A. Stump. Design and results of the 1^{st} satisfiability modulo theories competition (SMT-COMP 2005). *The Journal of Automated Reasoning*, 35:373–390, 2005.

[4] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification (CAV)*, pages 298–302, 2007.

[5] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2(195–22), 1987.

[6] S. A. Cook. The complexity of theorem-proving procedures. *Third Annual ACM Symposium on Theory of Computing*, 1971.

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

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(201-215), 1960.

[9] L. de Moura and N. Bjorner. Z3: An efficient smt solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.

[11] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. 18^{th} Intl. Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 81–94. Springer, 2006.

[12] B. Dutertre and L. de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, Stanford Research Institute (SRI), 2006.

[13] H. Edelsbrunner. *Algorithms in combinatorial geometry.* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[14] E. Ezra and S. Fine. On the cover of convex polyhedra in d-space. Technical Report H-0259, IBM Haifa Research Lab, Israel, 2008.

[15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc.* 16^{th} *Intl. Conference on Computer Aided Verification (CAV'04)*, number 3114 in Lect. Notes in Comp. Sci., pages 175–188. Springer, 2004.

[16] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Computational Geometry*, 2:127 – 151, 1987.

[17] D. S. Hochbaum, editor. *approximation-algorithms for NP-hard problems.* PWS Publishing Company, 1997.

[18] D. Kroening and O. Strichman. *Decision procedures – an algorithmic point of view.* Theoretical computer science. Springer-Verlag, May 2008.

[19] F. Maric and P. Janicic. argo-lib: A generic platform for decision procedures. In *IJCAR*, pages 213–217, 2004.

[20] J. P. Marques-Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. *IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.

[22] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. SAT competition'07, 2007.

[23] O. Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'00)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.

[24] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.

[25] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. 8-th European Conference on Logics in Artificial Intelligence*, volume 2424, pages 308–319. Springer, 2002.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.

Appendices

A Propositional Satisfiability

The Boolean satisfiability problem (SAT) has long been recognized as one of the fundamental problems of computer science, and has drawn both theoretic and practical research interest. Being at the heart of the basic NP-complete problem [6] all suggested algorithms for this problem are likely to be computationally intractable using a worst-case complexity analysis. However, many instances of this problem emerging from real world applications can be solved efficiently by recent algorithms. Modern SAT solvers can be roughly divided into two groups- DPLL based algorithms [7, 8] and stochastic search based algorithms; in this work DPLL based algorithms are reviewed and analyzed as algorithms for the SMT problem depend or at least make use of DPLL based algorithms. In general, the basic DPLL procedure can be considered as search through a binary tree spanning all possible assignments to the input formula, in which each internal node is mapped to a partial assignment and the leaves represent a full assignment. The DPLL procedure progresses by choosing an unassigned variable and deciding its value, identifying and propagating implications of this decision and backtracking in case of a conflict. Current solvers, for example [21], identify the problem's underlying structure which enables pruning large search spaces quickly by learning from conflicting assignments and identifying the important variables in the formula. In this chapter current algorithms are reviewed and analyzed using abstract-DPLL [25], an abstract declarative framework used to explain DPLL based solvers for both the propositional satisfiability and the satisfiability modulo theories. In this framework the DPLL procedure is modeled as a transition system. An algorithmic description of the DPLL procedure, conforming to the specifications of the formal framework is discussed in the following subparagraphs.

A.1 Abstract DPLL

Definition 12 (State) *A* state *in the transition system is either $FailState$ or the pair $M||F$, in which F is a CNF Boolean formula and M is a set of annotated literals representing the current partial assignment. A literal annotated as l^d represents a decision literal.*

The DPLL procedure is modeled as a set of transition rules, called the transition relation, which is a binary relation $\Rightarrow$, over the states. Given a state S the binary relation defines if there is a transition rule applicable to that state and if so, a definition

of the following state S' is also provided. In the following subparagraphs several transition systems are presented using different subsets of the DPLL transition rules presented in the next definition. A transition system can be used to decide the satisfiability of an input formula F by applying the transition rules to generate a derivation $\phi||F \Rightarrow ... \Rightarrow S_k$, where S_k is a final state, i.e no transition rules can be applied on S_k. it should either be a $FailState$ in which case F is unsatisfiable or $S_k = M||F$, and M is a model of F.

Definition 13 (DPLL transition rules) *The abstract-DPLL framework contains the following transition rules:*

1. *Unit Propagate*

$$M||F,\ C \vee l \Rightarrow Ml||F,\ C \vee l\ if \begin{cases} M \models \neg C \\ l\ is\ undefined\ in\ M \end{cases}$$

2. *Pure Literal*

$$M||F \Rightarrow Ml||F\ if \begin{cases} l\ occurs\ in\ F \\ \neg(\neg l\ occurs\ in\ F) \\ l\ is\ undefined\ in\ M \end{cases}$$

3. *Decide*

$$M||F \Rightarrow Ml^d||F\ if \begin{cases} l \vee \neg l\ occur\ in\ F \\ l\ is\ undefined\ in\ M \end{cases}$$

4. *Fail*

$$M||F,\ C \Rightarrow FailState\ if \begin{cases} M \models \neg C \\ M\ does\ not\ contain\ decision\ literals \end{cases}$$

5. *Backtrack*

$$Ml^dN||F,\ C \Rightarrow M\neg l||F,\ C\ if \begin{cases} Ml^dN \models \neg C \\ N\ does\ not\ contain\ decision\ literals \end{cases}$$

6. *Backjump*

$$Ml^dN||F,\ C \Rightarrow Ml'||F,\ C\ if \begin{cases} Ml^dN \models \neg C \\ exists\ a\ clause\ C' \vee l'\ such\ that: \\ \quad F,\ C \models C' \vee l'\ and\ M \models \neg C', \\ \quad l'\ is\ undefined\ in\ M \\ \quad l'\ or\ \neg l'\ occur\ in\ F\ or\ in\ Ml^dN. \end{cases}$$

7. *Learn*

$$M||F \Rightarrow M||F, C\ if \begin{cases} each\ atom\ of\ C\ occurs\ in\ F\ or\ in\ M \\ F \models C \end{cases}$$

8. *Forget*
   $M||F,\ C \Rightarrow M||F\ if \left\{\ F \models C\right.$

9. *Restart*
   $M \models F \Rightarrow \phi||F$

The basic DPLL procedure system employs the first five transition rules presented in Definition 13, The procedure, originally presented by [8] and improved by [7], is essentially a depth-first search procedure in which the *Decide* and $UnitPropagate$ transition rules are applied interchangeably, and $Backtrack$ applied in case a conflict is encountered.

Modern state of the art solvers do not implement the basic DPLL system, these solver are conflict driven; when a conflict is encountered it is analyzed and used to prune large search spaces. In a transition system modeling these algorithms, the $Backtrack$ transition rule is replaced by the $Backjump$ transition rule; using this rule several decisions can be undone in a single step. Furthermore, using learn transition rule new clauses can be added to the input formula, allowing further applications of the $UnitPropagate$ rule on the learnt clauses. Since potentially an exponential number of clauses can be added to the formula, The $forget$ rule is applied to remove learnt clause which are no longer relevant. The *Restart* rule is yet another improvement used to restart the search in cases the search procedure is not making enough progress.

A.2 Implementation Details

The performance improvement achieved by current SAT solver can be attributed to efficient implementations of the transition rules described in the previous paragraph, and solvers differ in the way these transition rules are implemented. While the initial, basic DPLL algorithm presented over 45 years ago was implemented in a straight forward recursive manner and suffered from the memory explosion problem, current solvers are implemented efficiently in an iterative manner and include smart decision heuristics, conflict driven learning and non-chronological backtracking, efficient deduction mechanism and use highly efficient data structures for storing clauses. Current implementations will typically apply the *Decide* transition rule only when $UnitPropagate$ rule can no longer be applied. If a conflict has been detected as a result of a decision or deduction the algorithm will analyze the conflict cause, *Learn* the conflict clause and $Backjump$ to a prior state. The $PureLiteral$ rule is considered only as a preprocessing step. In the rest of this section some of the algorithms implementing the transition rules are reviewed.

A.2.1 Unit Propagation

The unit propagation rule assigns value to literals needed to satisfy unit clauses, i.e., unresolved clauses in which all but one literal are assigned. The process of identifying and satisfying these clauses is also known Boolean Constraint Propagation (BCP), and in current SAT solvers this process usually consumes most of the solvers run time, hence the importance of efficient implementation. Current solvers usually implement the 2-literal watch algorithm suggested by [21]. In this approach two special literals, not assigned a false value at any given time are monitored for each clause; as long as this condition hold, no literal value is implied for that clause. When one of these literals is assigned a false value, the clause can be in either of the following states -
a. The clause is not implied, i.e. exist at least two literals in which are not assigned a false value (including the other watched literal). A new literal is then chosen to replace the literal assigned a false value.
b. The clause is implied; the unassigned watched literal will have its value implied to a true value.
Using this approach, clauses are visited only when one of the watched literals is assigned a false value instead at each assignment and when backtracking there is no need to change the watched literals.

A.2.2 Conflict Analysis, Learning and Backjumping

A key performance improvement is due to non-chronological backjumping, referenced previously as *Backjump* and *Learn* transition rules. Non-Chronological Backjumping is especially efficient in cases of structured problems, such as in problems emerging from real world applications. Given a conflicting clause, most current solvers will analyze the conflicting clause, learn the conflict cause and possibly flip the value of a decision variable earlier than the recent decision. During the conflict analysis, new clauses are added to the original formula; these clauses do not change the satisfiability of the formula, but can help the solver prune large spaces by avoiding assignments which would reproduce the learnt conflict cause. The concept of non-chronological backjumping was first introduced into the DPLL framework by [20] and became a common practice for all DPLL based SAT solvers. In this section analyzing and learning conflict clause is explained using the concept of an implication graph. This work follows the explanations presented also at [26].

Definition 14 (Implication Graph) *An implication graph is a directed acyclic graph in which each vertex is associated with an assignment of a value to a variable at a given*

decision level. A directed edge in the implication graph corresponds to an assignment implied by another assignment through BCP. Given an implication graph, a conflict occurs if there exists a variable p such that p has two vertices associated with it in the implication graph. A conflict clause is a disjunction of literals excluding from the search the set of assignments to a formula's variables that causes a conflict in the implication graph.

Using the implication graph, whenever a conflict occurs it is analyzed to detect a conflict clause. The solver then Backjumps to the maximum decision level of variables in the conflict clause. After backtracking the conflict clause will become a unit clause and since the current decision variable is a unit literal it is force to flip; such clauses are known as asserting clauses.

The conflict clause is generated by a bipartition of the implication graph. The partition divides the graph into the reason side, containing all the decision variables and the conflict side containing the conflicting variable. Vertices on the reason side that have at least one edge into the conflict side are considered the reason of the conflict - i.e. the conflict clause.

It is therefore easy to intuitively see that different cuts correspond to different learning strategies, as detailed in the following subparagraphs.

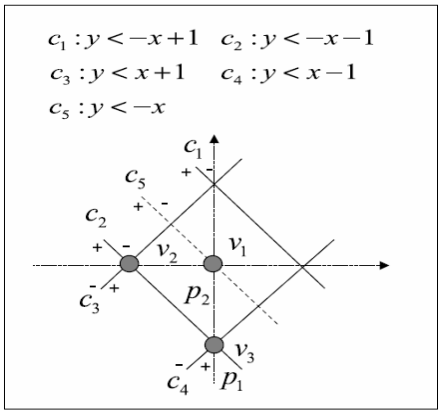
Definition 15 (Vertex Domination) *A vertex V_i is said to dominate a vertex V_j iff any path from the vertex representing the decision variable of the decision level associated with V_i to V_j needs to go through V_i .*

Definition 16 (Unique Implication Point) *A vertex V_i at decision level dl is a Unique Implication Point (UIP) iff any path form the decision variable of dl to the conflicting variables needs to go either through V_i or go through a vertex of decision level higher than dl that is on the reason side of the implication graph partition.*

Since it is possible to have several UIP in a single implication graph, the UIPs are numbered starting from the conflicting variable. To ensure that a conflict clause is an asserting clause the partition needs to have one UIP of the current decision level on the reason side, and all vertices assigned after this UIP on the conflict side. Thus after backtracking the UIP vertex will become a unit literal and make the clause an asserting clause. A simple learning strategy implemented in RelSAT solver is to generate conflict clause be continuously resolving the conflicting clause with its antecedent until the resolved clauses contains only decision variables of the current decision level or smaller. This learning scheme will put all variables assigned at the current decision level, except for the decision variable, on the conflict side and other variables at the reason side.

A different approach known as the *First UIP* learning scheme is focused on learning conflict clause relevant to the current conflict, by making the partition close to the conflicting variable; this is done by identifying the First UIP i.e., the UIP closest to the conflict. The graph partition places the first UIP of the current decision level on the reason side, and places the rest of the variables with edges leading to the conflicting variable and assigned after the first UIP on the conflict side.

$p_1 = c_2^+ \cap c_4^+$, בהנתן משתנה החלטה המקביל לאילוץ c_5- , ההיוריסטיקה תייצר שלוש נקודות שונות - v_1, v_2, v_3 בכל אחת מנקודות אלה כמות שונה של אילוצים מקבוצת האילוצים המגדירה את דפנות התא P1 מקבלת ערך אמת TRUE. עובדה מעניינת היא שהנקודה הראשונה מספיקה על מנת לזהות את הערך המתחייב עבור משתנה זה. עובדה זאת מעלה את חשיבות קביעת כמות הצעדים אותם מפעילה ההיוריסטיקה לשיפור רמת הדיוק באופן אמפירי.



תמונה 3- נקודות שונות לפיהן ניתן לקבוע ערך ע"י היוריסטיקת ההחלטה

בדקנו את השיטה באופן אמפירי על בעיות שנלקחו מתחרות smt-comp ’07. השיוונו בין ביצועי המערכת עבור ערכים שונים של פרמטר הדיוק וכמו כן מול שתי היוריסטיקות החלטה שונות הנפוצות במערכות אחרות. הראינו שיפור של כ-11% בזמן הריצה של האלגוריתם ביחס להיוריסטיקת ההחלטה המקורית הממומשת ע"י רוב הכלים הנוכחיים.

כפי שכבר צוין, זיהוי ערכים אלה הינו יקר חישובית, וניסיון לקבוע את ערכי האילוצים באופן ישיר יגרום בלירידה חדה בביצועי האלגוריתם.

בעבודה זאת אנו מציגים שיטה לשיערוך אילוצים אלה, המבוססת על זיהוי קודקוד יחיד של התא. במהלך החיפוש, כאשר נדרשת החלטה, אנו מזהים נקודה כזאת, וקובעים את ערכו של המשתנה (המקביל לאילוץ גיאומטרי) ע"י קביעת מיקום הנקודה ביחס לאילוץ – אם הנקודה נופלת מצידו החיובי אזי ערכו יקבע לערך אמת TRUE, או FALSE אחרת.

ניתן לראות שערכם של אילוצים אשר ערכם מתחייב (כלומר שאינם חותכים את התא) יקבע בצורה נכונה ע"י היוריסטיקה זאת וע"י כך תמנע החלטה הגורמת לסתירה. גם כאשר הערך אינו מתחייב (האילוץ חותך את התא), בוודאי לא תיווצר סתירה. בנוסף לכך, במקרה ושני הערכים אפשריים אין הערך השני נגרע מתוך החיפוש כך שנאותות האלגוריתם אינה נפגעת.

לדוגמא, בתמונה 2 התא P1 מיוצג ע"י הנקודה v_1. בהנתן האילוץ l_1, נבדוק באיזה חצי-מישור שלו נופלת הנקודה v_1. במקרה זה, הנקודה נמצאת בחצי המישור החיובי ואכן ניסיון לתת ערך אחר היה גורם לסתירה.

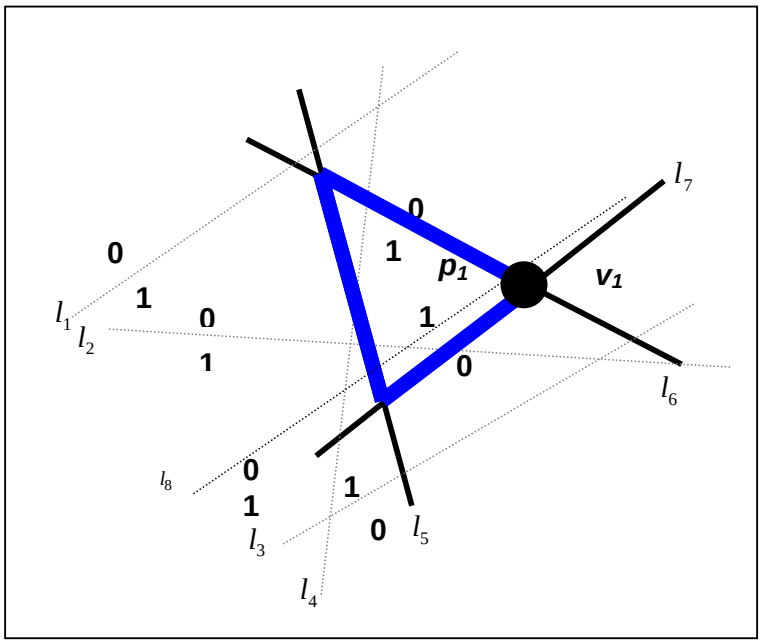
שימוש בהיוריסטיקת ההחלטה זאת דורש זיהוי קודקוד כלשהו של התא עבור כל השמת ערך למשתנה החלטה, דבר ההופך את הגישה הזאת ליקרה מידי באופן כללי. כפיתרון לבעיה זאת אנו מנסים לצמצם את עלות זיהוי הנקודה הנמצאת בתוך התא, ע"י שיערוך נקודה זאת. ניתן לשערך את ערכי הנקודה הזאת ברמות דיוק שונות אולם רמת דיוק גבוהה יותר גוררת באופן טבעי עלות חישובית גבוהה יותר , רמת הדיוק הדרושה לשיפור מיטבי של התוצאות נבדקה אמפירית.

נציג איפה את התהליך המתבצע ע"י ההיוריסטיקה -

בנקודת הפתיחה ממנה מתחיל החיפוש, מוצב הערך-0 לכל משתני הנוסחה. בהנתן ההשמה החלקית המגדירה את ערכיהם של קבוצת אילוצים, הנקודה המתקבלת מקיימת חלק מאותם האילוצים. בכל צעד נוסף נבחר אילוץ מקבוצת האילוצים המגדירים את התא, אשר הנקודה הנוכחית מפירה ונתקן את ההשמה כך שאילוץ זה יקבל ערך אמת זהה תחת ההשמה המוגדרת ע"י הנקודה. נבצע צעד זה כמות מוגדרת (לפי פרמטר) של פעמים, ונבחר את הנקודה בה כמות מקסימלית של אילוצים מתקיימת. נתייחס לכמות הצעדים המבוצעת ע"י ההיוריסטיקה *כפרמטר דיוק*, אשר דורש יותר זמן חישוב עבור ערכים מדויקים יותר.

לדוגמא בתמונה 3, נתונה ההשמה החלקית $c_2 = TRUE, c_4 = TRUE$ המגדירה את התא-

DPLL(T). כל החלטה הקובעת את ערכו של אחד המשתנים למעשה מחלקת את המרחב לחצי - החצי בו האילוץ הלינארי מתקיים. אוסף ההחלטות, או ההשמה החלקית הנשמרת במהלך החיפוש מגדירה את אוסף חיתוכי המרחב, אוסף זה מגדיר גוף קמור במרחב (*פולי'טופ*). לדוגמה, תמונה 2, מציגה קבוצה שלאילוצים $\{l_5, l_6, l_7\}$ אשר ערכם נקבע, כל אחד מאילוצים אלה מחלק את המרחב לשני חצאים - חיובי (בו כל נקודה מספקת את הפרדיקט הלינארי) שלילי, המסומנים ע"י 1 ו 0 בהתאמה. חיתוך השטחים המוגדרים ע"י השמה לאילוצים אלה יוצר את הגוף הקמור P1 .



תמונה 2 תיאור גיאומטרי של החיפוש

בהנתן משתנה אשר ערכו אינו נקבע עדיין, ניתן לזהות האם ערכו מתחייב ע"י התיאוריה בהנתן ההשמה החלקית – אם לשני חצאי המישור המוגדרים ע"י האילוץ יש חיתוך לא-ריק עם התא המוגדר ע"י ההשמה החלקית אזי השמת כל אחד משני הערכים לא יגרום לסתירה. כאשר רק לאחד מחצאי המישור חיתוך לא ריק עם התא, אזי השמת ערך זה מתחייבת. הגדרה שקולה היא – הערך מתחייב אם העל-מישור המוגדר ע"י האילוץ אינו חותך את התא המוגדר ע"י ההשמה החלקית, ולהיפך עבור אילוצים אשר ערכם אינם מתחייב.

ניתן לראות מתוך תמונה 2, את האילוצים אשר ערכם נקבע ביחס לתת השמה, אלו הם קבוצת האילוצים אשר אינם חותכים את התא P1 .

היוריסטיקת החלטה עבור DPLL(T) המבוססת על ניתוח גיאומטרי של מרחב החיפוש

להחלטות (בתמונה 1 – DECIDE) לפיהן נקבע הערך של משתנים יש חשיבות רבה ליעילות האלגוריתם, מערכות קיימות משתמשות בהיוריסטיקות החלטה על מנת לבחור ערכים אלה. היוריסטיקות אלה, שהושאלו מאלגוריתמים עבור לוגיקה פסוקית, מבוססות על ניתוח המבנה הבוליאני של הנוסחה, ואינן מתחשבות במבנה המוכתב ע"י התיאוריה. במקרים מסוימים השמות המתקבלות כתוצאה משמוש בהיוריסטיקות אלה גורמות לסתירה כאשר ההשמה נבדקת ביחס לתיאוריה. לדוגמא בהנתן הנוסחה - $(x = y) \wedge (x > z) \vee (y < z)$ השמה המעניקה ערך אמת TRUE לכל הפרדיקטים בנוסחא תוביל לסתירה.

ניתן להבין שקיים קשר הדוק בין כמות הערכים המתחיבים ע"י התיאוריה (כלומר שהשמה אחרת לערכים אלה תגרום לסתירה) לבין החשיבות של שיקולים הנגזרים מהתיאוריה להיוריסטיקת ההחלטה.

במקרה של תיאורית LRA, בהסתמך על עבודות תיאורטיות קודמות, קיים פוטנציאל לכך שבמקרה הממוצע, מספר רב של ערכים יקבעו בהנתן ההשמה החלקית. כלומר, בהנתן משתנה שערכו אינו קבוע, בהסתברות גבוהה ערכו של המשתנה נגרר ע"י התיאוריה בהנתן ההשמה החלקית. במידה ויושם למשתנה זה ערך אחר, האלגוריתם יתקל בסתירה.

דרך פשוטה להתחשב בתוצאות אלה היא כדילקמן - בהנתן השמה חלקית ומשתנה שערכו אינו נקבע, יש לבדוק האם ערכו של המשתנה נגרר ע"י התיאוריה בהנתן ההשמה החלקית. אולם קיים קושי בשימוש ישיר בדרך זאת - זיהוי הערכים המתחיבים ע"י הנוסחה מחייב פתרון של מערכת לניארית, כלומר בדיקה ע"י הפעלת פרוצדורת הכרעה (המבוססת על Simplex –) האם קוניונקציה המחברת את ההשמה החלקית עם אחד מהליטרלים של משתנה ההחלטה הינה ספיקה. בפועל פתרון זה יקר מידי.

בעבודה זאת אנו מציגים שיטה לשיערוך משתנים שערכם נגרר ע"י התיאוריה, שעלותה קטנה בהרבה מקביעת ערכים אלה באופן ישיר. למרות שמדובר בשיערוך בלבד, נאותות האלגוריתם נשמרת כיוון שאנו מקודדים מידע זה כהיוריסטיקת החלטה, כך שבמידה והשיערוך אינו נכון האלגוריתם יכול להמשיך בחיפוש.

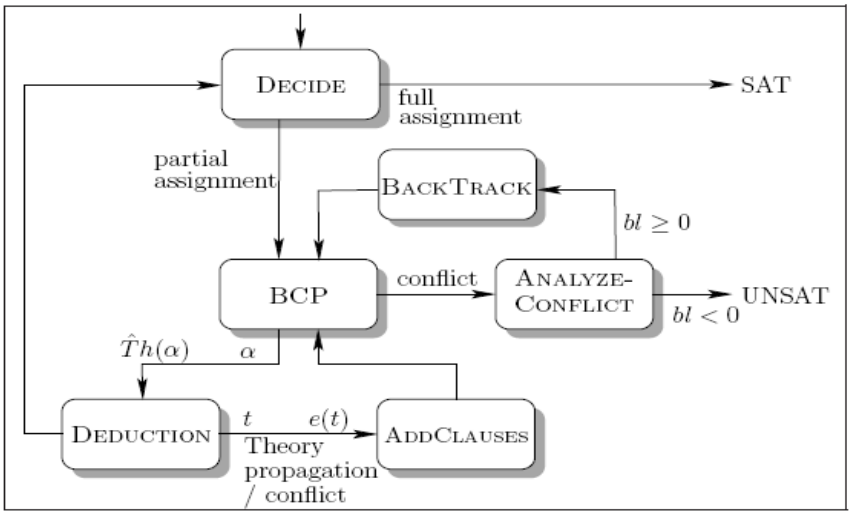
היוריסטיקה זאת מבוססת על ניתוח גיאומטרי של מרחב החיפוש הנחקר ע"י אלגוריתם ה-

שיטות עדכניות לפתרון בעייה זאת, משלבות באופן הדוק יותר את שתי פרוצדורות ההכרעה, ומאפשרות להן לפעול באופן חלקי, בניגוד לפרוטוקול הפשוט המחייב מציאת פיתרון מלא לשלד הבוליאני.

התקדמות נוספת הוצגה ע"י סכימת ה- DPLL(T) הממומשת בכל המערכות העדכניות - בנוסף לכך שהקריאה לפרוצדורת ההחלטה נעשת עבור השמות חלקיות, הפרוצדורה יכולה לזהות משתנים אשר ערכם מתחייב על ידי התיאוריה (אחרת יגרמו לסתירה) בהנתן ההשמה החלקית. לדוגמא בהנתן שהפרדיקטים הבאים מתקימים - (x=y) ו (y=z) אזי ברור שערכו של הפרדיקט (z=x) נקבע אף הוא.

תיאור כללי של סכימת ה- DPLL(T) מוצג בתמונה 1.

ניתן לחשוב על החיפוש שמבצע האלגוריתם כחיפוש על עץ החלטה בינארי, הפורש את כל האפשריות להשמת ערכים למשתני הנוסחה. האלגוריתם מתקדם על עץ זה ע"י בחירת משתנה אשר ערכו אינו נקבע (*משתנה ההחלטה*) , וקובע את ערכו (בתמונה 1 – DECIDE). לאחר מכן מתבצעת בדיקה ערכו (בתמונה 1 – BCP)- האם ההשמה יוצרת סתירה? במידה וכן, האלגוריתם משחזר את מצבו הקודם וממשיך ממנו את החיפוש. במידה ולא, מתחיל תהליך איטרטיבי בו מושם ערכם של משתנים אשר ערכם נגרר ע"י המבנה הלוגי או ע"י התיאוריה. תהליך זה עוצר בשני מצבים- כאשר ישנה סתירה או כאשר הוא מתכנס , כלומר כאשר אין עוד ערכים חדשים המתחייבים מההשמה הנוכחית. האלגוריתם עוצר בשני מצבים – כאשר הוא מצליח להשלים השמה מלאה ללא סתירות או כאשר אין עוד השמות חדשות לנסות.



תמונה 1 - פרוצדורת DPLL(T)

הקדמה – הגדרת הבעיה

בעיית ההכרעת הספיקות של נוסחא היא אחת מהבעיות הנחקרות ביותר במדעי המחשב. בעיה זאת ניתנת להגדרה באופן פשוט – *בהנתן נוסחא φ המוגדרת מעל אוסף משתנים* A , *האם קיימת השמה ל-* A , *כך ש-φ תקבל ערך אמת –TRUE*.

הגדרה זאת למעשה מגדירה משפחה של בעיות, השונות בתיאוריה הלוגית מעליה מוגדרת הבעיה.

בעוד פתרון בעיה זאת עבור לוגיקה פסוקית הינו אפשרי גם עבור בעיות גדולות משמעותית, (בעיה זאת היא אומנם NP complete אולם קיימים עבורה אלגוריתמים יעילים) עבור לוגיקות מורכבות יותר האתגר למציאת פיתרון בזמן סביר עדיין במידה רבה פתוח.

בעבודה זאת אנו דנים באחת מלוגיקות אלה, המוגדרת ע"י אילוצים לניאריים מעל קבוצת המספרים הממשים (LRA) ובעלי מבנה בוליאני כללי. לדוגמא האם לנוסחא הבאה - $(x = y) \wedge (x > z) \vee (y < z)$ קיימת השמה למשתנים x,y,z כך שהנוסחה תקבל ערך אמת TRUE.

עבודה עדכנית בתחום

אלגוריתמים קיימים עבור בעיה זאת משלבים אלגוריתם חיפוש של בעיית הספיקות עבור לוגיקה פסוקית בשילוב עם פרוצדורת הכרעה ספציפית עבור LRA, לדוגמא ע"י שימוש באלגוריתם סימפלקס. פרוצדורה זאת מקבלת כקלט רק אילוצים בעלי מבנה קוניוקטיבי ("וגם").

למעשה בעיית ההכרעה נפתרת ע"י שתי פרוצדורות שונות, עם פרוטוקול מוגדר להחלפת מידע. לדוגמא, הנה פרוטוקול פשוט שהיה בשימוש מערכות מדור קודם –

1. קידוד כל פרדיקט לינארי המופיע בנוסחא כמשתנה מלוגיקה פסוקית, כלומר נחליף כל פרדיקט לינארי - $\sum \alpha_i x_i \infty c, \infty \in \{>,<,\leq,\geq,=\}, c \in R$ במשתנה בולאני. הנוסחה החדשה מכונה – *השלד הבוליאני* של הנוסחה המקורית.
2. חיפוש השמה מספקת לשלד הבוליאני בעזרת אלגוריתם ללוגיקה פסוקית, אם השמה כזאת אינה קיימת – השב כי אין השמה מספקת לבעייה המקורית.
3. אחרת – יש לבדוק את הפתרון שהתקבל בעזרת אלגוריתם המיועד לתיאוריה זאת. אם הפתרון אינו סותר את האילוצים הלינארים המוגדרים בנוסחא – החזר ערך "ספיק"
4. אחרת- הוסף פסוקית הסותרת פיתרון זה בלוגיקה פסוקית, וחזור ל -2 .

הליטרלים בה הם בעלי ערך FALSE. היא פסוקית –unit אם רק אחד מהליטרלים בה עדיין לא נמצא תחת השמה.

6. הגדרה (*ספיקות תחת השמה*) נוסחה היא ספיקה תחת השמה אם כל פסוקיותיה מסופקות תחת ההשמה.

הגדרות

בחלק זה נציג מספר הגדרת הדרושות להבנת עבודה זאת, ההגדרות הינן משני סוגים – הגדרות גיאומטריות והגדרות הקשורות לבעיית הספיקות הבוליאנית.

- **הגדרות גיאומטריות**

1. הגדרה (*על-מישור*) על-מישור מוגדר ע"י משוואה לינארית מהצורה $f(\boldsymbol{x}) = b$, כך ש-f הינה פונקציה לינארית מהצורה $\boldsymbol{x} \in \boldsymbol{R}^d, \boldsymbol{R}^d \rightarrow \boldsymbol{R}$, ו- b הינו ערך קבוע
2. הגדרה (*סידור של על-מישורים*) סידור של על מישורים הוא אוסף על מישורים ב- $\boldsymbol{R}^d$ סידור על-מישורים מגדיר חלוקה של המרחב לאוסף תאים קמורים.
3. הגדרה (*וקטור מיקום*) בהנתן אוסף על מישורים בסידור, נגדיר את הפונקציה הבאה –

$$u_i(p) = \begin{cases} +1, \ if \ \ p \in h_i^+ \\ 0, \ if \ \ p \in h_i \\ -1, \ if \ \ p \in h_i^- \end{cases}$$

כך ש- h_i^-, h_i^+ מציינים את שני חצאי המרחב המוגדרים ע"י h_i. הוקטור $u(p) = (u_1(p), ..., u_n(p))$ נקרא וקטור המיקום של נקודה P.

4. הגדרה (*תאים*) כל הנקודות אשר חולקות באותו וקטור מיקום מגדירות תא.

- **הגדרות הקשורות לבעיית הספיקות הבוליאנית**

1. הגדרה (*ליטרל*) תהי P קבוצה של משתנים, אם משתנה p שייך לקבוצה זאת, אזי p ושלילתו הינם ליטרלים.
2. הגדרה (*פסוקית*) פסוקית היא דיסיונקציה של ליטרלים
3. הגדרה (*נוסחה*) נוסחה (בפרט, נוסחת CNF) היא קוניונקציה של פסוקיות
4. הגדרה (*השמה*) השמה, או *השמת אמת* היא קבוצת ליטרלים כך ש $\forall p \in P \mid \neg((p \in M) \wedge (\neg p \in M))$, ליטרל הוא בעל ערך TRUE בהשמה אם הוא שייך להשמה ולהפך אם שלילתו שייכת להשמה.
5. הגדרה (*מצב של פסוקית תחת השמה*) פסוקית היא מסופקת תחת השמה אם לפחות אחד מהליטרלים שלה הוא בעל ערך – TRUE בהשמה. הפסוקית מכילה סתירה, אם כל

היוריסטיקת החלטה עבור פרוצדורת הכרעה של אילוצים לינאריים המבוססת על ניתוח גיאומטרי של מרחב החיפוש

דן גולדוואסר

<u>תקציר</u>

עבודה זאת חוקרת את בעיית ההכרעה עבור אילוצים לינאריים המוגדרים מעל קבוצת המספרים הממשים ובעלי מבנה בוליאני כללי תוך שימוש בשיטות ומושגים הלקוחים מעולם הגיאומטריה החישובית.

בהנתן נוסחה המכילה אילוצים, אנו מגדירים את מרחב החיפוש הגיאומטרי כקבוצת התאים המוגדרת ע"י חלוקת המרחב לגופים קמורים המוגדרת ע"י האילוצים הלינאריים המופיעים בנוסחא.

עבודה זאת מראה כי לשימוש במרחב חיפוש זה, בניגוד למרחב החיפוש הבוליאני, יש יתרון גודל כאשר כמות האילוצים המופיעים בנוסחא גדולה מכמות המשתנים מעליהם מוגדרים אילוצים אלו, כפי שאכן המצב בבעיות הנפוצות בתחום זה.

על מנת לנצל תכונות אלה, העבודה גם מציגה היוריסטיקת החלטה המבוססת על שיערוך של ערכים הנגררים ע"י המבנה הגיאמטרי של הנוסחה במהלך החיפוש. שיערוך זה נעשה ע"י ניתוח מרחב החיפוש הגיאומטרי. אנו מראים ששימוש בהיוריסטיקה זאת בפיתרון הבעיות הסטנדרטיות בתחום מביא לשיפור בזמן הריצה.

היוריסטיקת החלטה עבור פרוצדורת הכרעה של אילוצים לינאריים המבוססת על ניתוח גיאומטרי של מרחב החיפוש

מאת:

דן גולדוואסר

בהנחיית:

ד"ר רונן שלתיאל (אוניברסיטת חיפה)

ד"ר עופר שטריכמן (הטכניון)

ד"ר שי פיין (מעבדת המחקר של IBM בחיפה)

עבודת גמר מחקרית (תיזה) המוגשת כמילוי חלק מהדרישות

לקבלת התואר "מוסמך האוניברסיטה"

אוניברסיטת חיפה

הפקולטה למדעי החברה

החוג למדעי המחשב

מאושר על ידי _____ תאריך _____

(מנחה העבודה)

מאושר על ידי _____ תאריך _____

(מנחה העבודה)

מאושר על ידי _____ תאריך _____

(מנחה העבודה)

מאושר על ידי _____ תאריך _____

(יו"ר הוועדה החוגית למ"א)

היוריסטיקת החלטה עבור פרוצדורת הכרעה של אילוצים לינאריים המבוססת על ניתוח גיאומטרי של מרחב החיפוש

דן גולדוואסר

עבודת גמר מחקרית (תיזה) המוגשת כמילוי חלק מהדרישות
לקבלת התואר "מוסמך האוניברסיטה"

אוניברסיטת חיפה
הפקולטה למדעי החברה
החוג למדעי המחשב

יוני, 2008