A Theory-Based Decision Heuristic for DPLL(T)

Dan Goldwasser
CS, Haifa University
Haifa, Israel
dgoldwas@cs.haifa.ac.il

Ofer Strichman
Information System Engineering,
IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Shai Fine
IBM Haifa Research Lab,
Haifa, Israel
shai@il.ibm.com

***Abstract*—We study the decision problem of disjunctive linear arithmetic over the reals from the perspective of computational geometry. We show that traversing the *linear arrangement* induced by the formula's predicates, rather than the DPLL(T) method of traversing the Boolean space, 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). 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.**

I. Introduction

Quantifier-free Linear arithmetic over the reals (QF_LRA in the terminology of the SMT community, or LRA for short) is arguably the most important decidable first-order theory in verification, other than propositional logic, and a subject for research [8] and annual competitions in the SMT community [3]. All competitive SMT solvers, including Yices [8], Z3 [6], ArgoLib [16], MathSAT [1] and CVC-3 [4], to name a few, decide LRA by instantiating the DPLL(T) framework [21], [12] with general simplex, as introduced by Dutertre and de Moura in [8], [9].

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 [21] and later materialized into the award-winning SMT solver Barcelogic [12]. It is based on an interplay between a SAT solver and DP_T. Consider first the following basic procedure, which existed prior to DPLL(T) in systems such as CVC [20] 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.

DPLL(T) improves this procedure 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). We will give a more formal description of DPLL(T) in Sect. II.

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. We do not attempt to solve this problem in this article, but rather to show a method in which some information from the theory can still be obtained in a cost-effective manner. Specifically, we show 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.

We can learn about the potential of theory propagation in the case of LRA, that is, the average number of implications given a partial assignment, from a theoretical result in computational geometry by Haussler and Welzl [13]. 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* [10]. 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

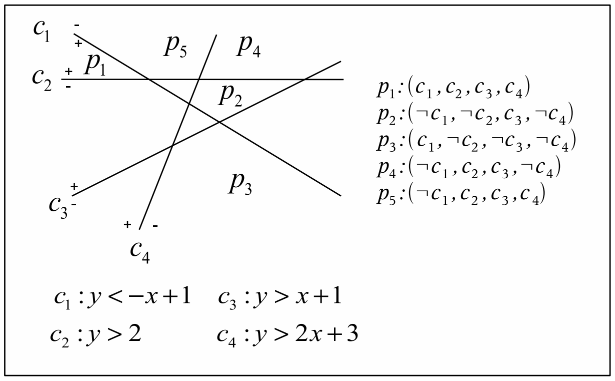
c_1
c_2
c_3
c_4
p_1
p_2
p_3
p_4
p_5
$p_1 : (c_1, c_2, c_3, c_4)$
$p_2 : (\neg c_1, \neg c_2, c_3, \neg c_4)$
$p_3 : (c_1, \neg c_2, \neg c_3, \neg c_4)$
$p_4 : (\neg c_1, c_2, c_3, \neg c_4)$
$p_5 : (\neg c_1, c_2, c_3, c_4)$
$c_1 : y < -x + 1$
$c_3 : y > x + 1$
$c_2 : y > 2$
$c_4 : y > 2x + 3$

Fig. 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.

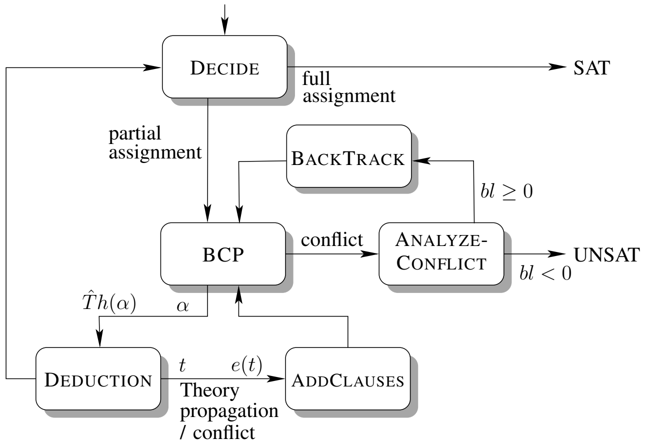

Fig. 2. The main components of DPLL(T). Theory propagation is implemented in DEDUCTION.

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 [13] and Clarkson [5], suggested that for a $n \times d$ linear system, if we randomly select r constraints 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 [11] 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.

The rest of this paper is structured as follows. We begin by describing DPLL(T) and general simplex in Sect. II-A. Section III presents a geometric interpretation of the search space and how it can be used for approximating implications that can help the search process. Section IV describes our experimental results. We conclude in Sect. V with some thoughts on possible future uses of the observations we make.

II. PRELIMINARIES

A. The DPLL(T) framework

State-of-the-art SMT solvers follow the DPLL(T) framework [21]. The components of the algorithm are those of DPLL and a decision procedure DP_T for a conjunctive fragment of a theory T. The name DPLL(T) emphasizes that this is a framework that can be instantiated with a different theory T and a corresponding decision procedure. We assume that the reader is familiar with the basics of DPLL in the description that follows.

In the version of DPLL(T) presented in Algorithm 1 (see also Fig. 2, which is copied from [15]), 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, we require t 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 we wrote 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., [9]). Moreover, even checking for consistency of the current partial assignment is too costly in practice. Instead, competitive solvers only do light-weight theory propagation and defer the consistency check to when a full assignment is achieved, as we will describe later in Sect. II-B.

Algorithm 1 The DPLL(T) framework.

```
1: function DPLL($T$)
2:     AddClauses ($cnf(e(\varphi))$);
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:                 $btrack\text{-}level$ := Analyze-Conflict ();
10:                if $btrack\text{-}level < 0$ then
11:                    return "Unsatisfiable";
12:                else BackTrack($btrack\text{-}level$);
13:            $t$ :=Deduction ($\hat{Th}(\alpha)$);
14:            AddClauses ($e(t)$);
15:        until $t \equiv true$
```

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.

B. 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 [8]. 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 form $\Sigma a_{ij} x_j \leq b_i$, it replaces it with the two constraints

$$\begin{array}{l} \sum a_{ij} x_j - s_i = 0 \\ s_i \leq b_i \ , \end{array}$$

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 we can reduce the value of x_i by changing the value of x_j. If not – the instance is declared unsatisfiable. Suppose that there exists such a variable x_j (we say then that x_j is *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 [8]. This is not essential for understanding the rest of the paper, however, and was only brought here for completeness. The important point is that 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. 2. In Fig. 3, 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.

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. I), 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. 3. This procedure can only detect inconsistencies of bounds, for

Algorithm 2 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.
```

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 3 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 $c_i \geq u_i$ 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 $\beta(x_i) > c_i$ then
6:         update-assignment($x_i$, $c_i$);
```

III. Geometric representation

A. Background on linear arrangements

Given a linear arithmetic formula φ, we denote 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 we saw in Sect. I these cells can be mapped into an assignment to the predicates in $C(\varphi)$.

For example, consider the cell marked P_1 in Fig. 3. 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

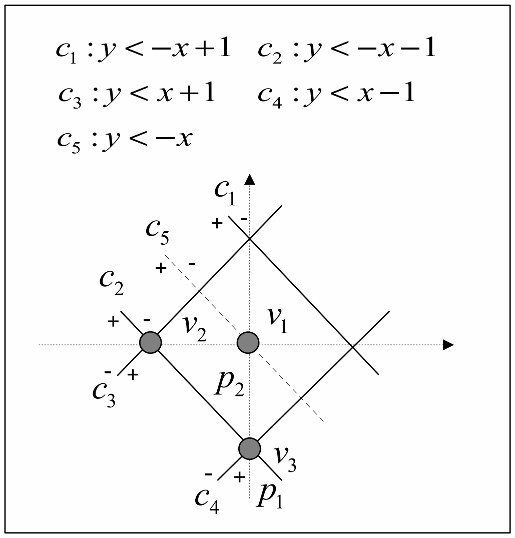

Fig. 3. A linear arrangement corresponding to five linear constraints. The axes and the points v_1, v_2, v_3 are not part of the arrangement, but useful for understanding the progress of simplex given the same constraints. The points $v_1, \ldots, v_3$ represent a possible progress of the assignment maintained by simplex. v_1 is the initial assignment, which is always the origin.

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, we checked these values in various SMT-LIB benchmarks – see Table I. 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.

B. 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 1: In Fig. 3 the subset (c_2, c_4) is T-consistent and forms the cell P_1. If c_1 is the current decision variable, decid-

[1]The 'O' notation is not precise here, because the constant actually depends on d. We follow here the convention used by Halperin in [14], 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]We discuss possible implications of this gap in complexity in Sect. V.

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

TABLE I
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

ing 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. We tested this observation empirically: Fig. 4 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.

C. Identifying T-implications

There are two natural ways to identify T-implications that we are aware of. Given a system of constraints S corresponding to the current partial assignment and an unassigned predicate p, the first method (called *plunging* in [7]) 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. 3 the vertices of the cell P_2 fall on both halfspaces of c_5. Both of these methods turn out to be 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.

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

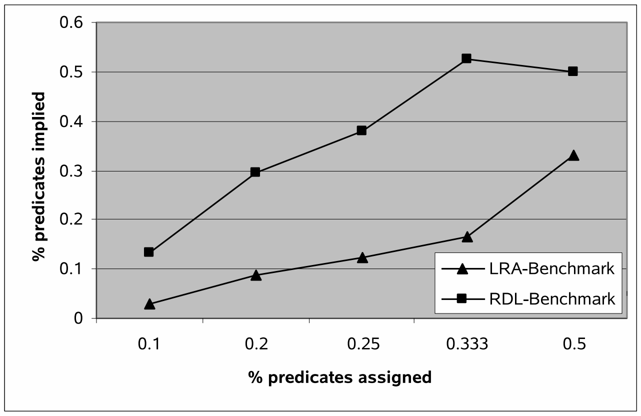

Fig. 4. The proportion of assigned vs. implied values at different points during the search. Two benchmarks were checked: a Linear Arithmetic (LRA) benchmark – invar.induct – containing 633 constraints and 163 variables, and a Difference Logic (RDL) benchmark – abz5_1000.smt – containing 1011 constraints and 102 variables.

A possible way to alleviate the computational problem of checking all vertices is to approximate the geometric representation of the T-solver's state. This, however, prevents us from using this information to identify implications since an approximated implication may affect the soundness of the algorithm; we need therefor to restrict the use of such information, allowing the DPLL search to recover in case the value was not in fact implied. Our system 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, we only use it 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? we can, for example, 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. 3. 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. 3) 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, we can only use it, as before, to approximate implications rather than infer them.

We can attempt to improve this approximation 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.

D. Taking decisions at T-inconsistent points

Recall that the current partial assignment is not necessarily T-consistent because in practice most solvers perform complete T-consistency checks at selected intervals or even only when the assignment is full. This is the strategy of the SMT solver Argolib [16], on top of which we implemented our heuristic.

We checked empirically the ratio of times that decisions are taken when the partial assignment is T-consistent. The larger this number is, the less our heuristic will be affected by this problem. We evaluated this number 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. We ran this experiment for each of the benchmarks that we used for evaluation, as described in Sect. IV. The average proportion of decisions taken when the partial assignment was T-consistent across all benchmarks was 0.78.

IV. EVALUATION

We tested the performance of our approach on the LRA benchmarks [3] 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 [17]

[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.

TABLE II
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

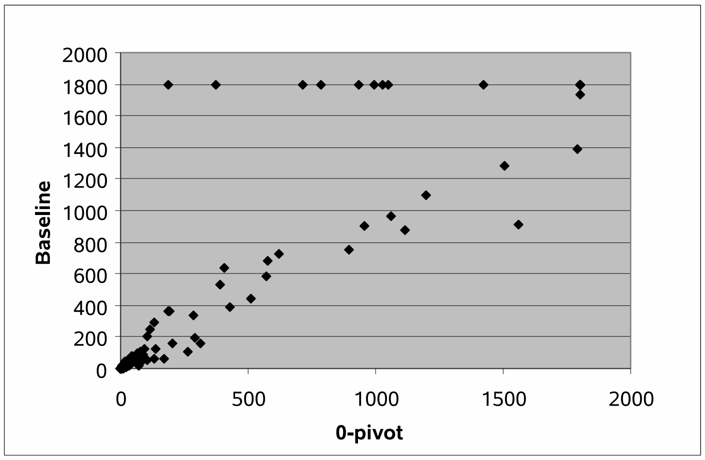

Fig. 5. Baseline vs. 0-pivot (run-time).

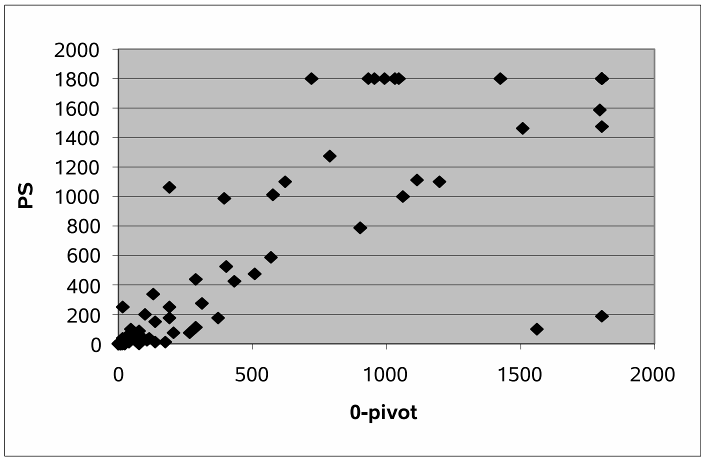

Fig. 6. PS vs. 0-pivot (run-time).

heuristic and sets its value to FALSE. We also implemented and evaluated the system using 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 [19] and lately adopted by RSAT under the name 'progress saving' (PS) [18]. We'll call it the PS strategy.

We evaluated several variants of our approach 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 II 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 we 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 5–6 show detailed results when comparing 0-pivot with the two SAT-based heuristics. From Fig. 5 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 6 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-COMP'07 benchmark set, we measured the following data, 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% that we 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. 4: 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. 4 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.

V. Discussion and Future work

The improvement in run time that we showed 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;[7]
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. III-A, 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

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

[7]An exception is the recently published technical report [11] that we mentioned earlier. Given a CNF-style formula, the authors suggest to 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.

Table I), 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 build efficiently a symbolic representation of the cells in an arrangement – 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.

Acknowledgement

We thank Esther Ezra for her advice on various computational geometry issues.

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] Clark Barrett, Leonardo de Moura, and Aaron 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] Clark Barrett and Cesare 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] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.

[8] Bruno Dutertre and Leonardo 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.

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

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

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

[12] 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.

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

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

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

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

[17] 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.

[18] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. SAT competition'07, 2007.

[19] Ofer 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.

[20] 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.

[21] 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.