Volume Computation for Boolean Combination of Linear Arithmetic Constraints*

Feifei Ma[1,2], Sheng Liu[1,2], and Jian Zhang[1]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2] Graduate University, Chinese Academy of Sciences
{maff,lius,zj}@ios.ac.cn

Abstract. There are many works on the satisfiability problem for various logics and constraint languages, such as SAT and Satisfiability Modulo Theories (SMT). On the other hand, the counting version of decision problems is also quite important in automated reasoning. In this paper, we study a counting version of SMT, i.e., how to compute the volume of the solution space, given a set of Boolean combinations of linear constraints. The problem generalizes the model counting problem and the volume computation problem for convex polytopes. It has potential applications to program analysis and verification, as well as approximate reasoning, yet it has received little attention. We first give a straightforward method, and then propose an improved algorithm. We also describe two ways of incorporating theory-level lemma learning technique into the algorithm. They have been implemented, and some experimental results are given. Through an example program, we show that our tool can be used to compute how often a given program path is executed.

1 Introduction

The past decade has seen much interest in the satisfiability (SAT) problem, i.e., determining whether a Boolean formula (in conjunctive normal form) is satisfiable. It is a classical decision problem in propositional logic reasoning. More recently, Satisfiability Modulo Theories (SMT), as an extension to SAT, has received more and more attention [3,9,18,16,19]. Instead of Boolean formulas, SMT checks the satisfiability of logical formulas with respect to combinations of background theories (often expressed in classical first-order logic with equality).

On the other hand, the counting version of the decision problems is also quite important in automated reasoning. For instance, the model counting problem, i.e., counting the number of models of a propositional formula, is closely related to approximate reasoning [20,2]. It has been studied by various researchers, especially in recent years. See for example, [14,2,23,22,1,8,13,21].

* This work is supported by the National Science Foundation of China (NSFC) under grant No. 60673044. Correspondence to: Jian Zhang, P.O.Box 8718, Beijing 100190, CHINA.

The counting version of SMT has received less attention so far, although it is also very important. It has potential applications in various areas, such as program analysis and verification, as well as approximate reasoning. Suppose we have a knowledge base or a formal description of some application, specified by an SMT formula Φ, and we are given a formula φ such that neither φ nor $\neg\varphi$ is a logical consequence of Φ. Then it is reasonable to assume that, the more models of Φ support φ, the more likely φ is true for the application.

There are various ways of analyzing programs statically. One class of analysis techniques checks the program's properties by processing individual paths in the program's flow graph. Not all paths in the graph correspond to program executions. A path is called *feasible* if there are some initial data values for the variables that can drive the program to be executed along that path. Otherwise, the path is called *infeasible*. There are quite some works on path feasibility analysis. A basic approach is to compute the path condition (which is a set of constraints in the form of SMT instances), and decide whether it is satisfiable or not. The program path is feasible if and only if the path condition is satisfiable. So the path feasibility analysis problem is reduced to a constraint solving problem [24,26].

Sometimes we can go one step further and ask *how many* data values satisfy the path condition, which means *how often* the program path can be executed. A path is a *hot path* if it is frequently executed. Identification of such paths is necessary in some applications such as embedded systems.

In this paper, we study how to compute the volume of the solution space (or, how to count the number of solutions), given a set of SMT instances (or more precisely, Boolean combination of linear arithmetic constraints).

The paper is organized as follows. We first recall some basic concepts and give some notations in the next section. In Section 3, we outline a straightforward method, and then in Section 4 we propose an improved algorithm. We have implemented the methods, and some experimental results are given in Section 5. In Section 6 we describe the application of the techniques to program analysis. Then we discuss some issues and mention some related works. Finally we conclude the paper.

2 Background

This section describes some basic concepts and notations. We also mention some existing techniques and tools that will be used later.

The main object of study in this paper can be regarded as an SMT instance where the theory is restricted to the linear arithmetic theory. More specifically, we study a set of constraints involving variables of various types (including integers, reals and Booleans). There can be logical operators (like AND, OR) and arithmetic operators (like addition, subtraction).

We use b_i $(i > 0)$ to denote Boolean variables, x_j $(j > 0)$, y, ... to denote numeric variables.

A simple example of constraints is $x_1 + x_2 < x_3$. We call it a linear arithmetic constraint (LAC), which is a comparison between two linear arithmetic expressions. Such a constraint can be denoted by a Boolean variable. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals.

In this paper, a constraint ϕ is represented as a Boolean formula $PS_\phi(b_1, \ldots, b_n)$ together with definitions in the form: $b_i \equiv expr_{i1}\ op\ expr_{i2}$. Here $expr_{i1}$ and $expr_{i2}$ are numeric expressions, *op* is a relational operator like '$<$', '$=$', etc. The Boolean formula PS_ϕ is the *propositional skeleton* of the constraint.

For a Boolean formula, a *model* is an assignment[1] of truth values to all the Boolean variables such that the formula is evaluated to TRUE. The *satisfiability* problem is concerned with the existence of models, while the *model counting* problem is to compute the number of models.

The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. But the subject of this paper is how to compute the *volume* of the solution space (or, how to count the number of solutions) for the formulas. Obviously this problem generalizes both the model counting problem in the propositional logic and the classical volume computation problem for convex polytopes.

Generally speaking, a polytope is defined as the bounded intersection of finitely many halfspaces/inequalities. Formally, it is usually described using the H-representation $\{\mathbf{x}|\mathbf{Ax} \le \mathbf{b}\}$ (where $\mathbf{A}$ is a matrix of dimension $m \times d$ and $\mathbf{b}$ a vector of dimension m). If $\mathbf{x}$ is a real vector, there are already some tools available to compute the continuous volume of a polytope. For example, `vinci` [6] is such a tool, whose input is a set of linear inequalities over reals.

Sometimes we are interested in the number of *integer* points in the solution space. Tools are also available for counting such points inside a bounded polytope, e.g., `azove` [4] and `LattE` [17]. `azove` is a tool designed for counting and enumeration of 0/1 vertices. That is to say, the domain of the variables is $\{0, 1\}$. Given a polytope $\{\mathbf{x}|\mathbf{Ax} \le \mathbf{b}\}$, all 0/1 points lying in it can be counted or enumerated. We extended the domain of each variable to $\{0, 1, ..., 2^l - 1\}$ so that it can count or enumerate all integer points with word length l in the polytope. `LattE` is a similar tool dedicated to the counting of lattice points inside convex polytopes and the solution of integer programs. But all the parameters in the matrix $\mathbf{A}$ and vector $\mathbf{b}$ should be integers.

3 A Straightforward Method

We know that an SMT(LAC) instance ϕ is satisfiable if there is an assignment α to the Boolean variables in PS_ϕ such that:

1. α propositionally satisfies ϕ, or formally $\alpha \models PS_\phi$;
2. The conjunction of theory predicates under the assignment α, which is denoted by $\hat{Th}(\alpha)$, is consistent with respect to the addressed theory.

We call an assignment satisfying the above conditions a *feasible* assignment. To check the satisfiability of a given formula, an SMT solver tries to find such an assignment.

[1] In this paper we represent an assignment as a set of literals.

Given a formula ϕ, we denote the set of all its feasible assignments by $Mod(\phi)$. When we talk about the volume of an assignment α, which is denoted by $volume(\alpha)$, we refer to the volume of polytope corresponding to α. The **volume** of a formula ϕ, denoted by V_ϕ, is formally defined as follows:

$$V_\phi = \sum_{\alpha \in Mod(\phi)} volume(\alpha)$$

An SMT solver typically combines SAT and theory-specific solving [16]. Most of the state-of-the-art SMT solvers are built within the DPLL(T) architecture [12], which can be adapted to compute the volume of a given formula. We ask the SMT solver to find out all feasible assignments to formula ϕ, compute the volume of each assignment and then add them up.

Algorithm 1

```
1: $V_\phi$=0;
2: for each model $\alpha$ of $PS_\phi$ do
3:    if $\hat{Th}(\alpha)$ is consistent then
4:       compute $volume(\alpha)$;
5:       sum up the volume: $V_\phi$ += $volume(\alpha)$;
6:    end if
7: end for
8: return $V_\phi$;
```

4 Volume Computation in Bunches

4.1 The Basic Idea

In the straightforward approach, each time a feasible assignment is obtained, we need to compute the volume of a conjunction of linear constraints corresponding to the literals in the assignment. Since volume computation is a time-consuming task, which is shown to be #P-hard by Dyer and Frieze [10], it is desirable to reduce the number of calls to volume computation routines.

Given an assignment α to the Boolean variables in formula ϕ, concerning the two conditions for feasible assignments, we distinguish four cases:

(i) $\alpha \models PS_\phi$, and $\hat{Th}(\alpha)$ is consistent in the specific theory. (Here α is a feasible assignment.)
(ii) $\alpha \models PS_\phi$, while $\hat{Th}(\alpha)$ is inconsistent in the specific theory.
(iii) α falsifies ϕ propositionally, while $\hat{Th}(\alpha)$ is consistent in the specific theory.
(iv) α falsifies ϕ propositionally, and $\hat{Th}(\alpha)$ is inconsistent in the specific theory.

The volume of formula ϕ is the sum of volumes of all assignments in case (i). When $\hat{Th}(\alpha)$ is inconsistent, as in case (ii) or case (iv), there is no solution to $\hat{Th}(\alpha)$, and consequently $volume(\alpha) = 0$. So when adding up the volume of feasible assignments, it will be safe to count in some theory-inconsistent assignments since they would not affect the total volume. At first glance it seems that these zero-volume assignments would give rise to additional calls to volume computation routines. However, when properly selected, they can be combined with the feasible assignments to form fewer assignments, reducing the number of volume computations.

Definition 1. *A set of full assignments $\mathcal{S}$ is called a* bunch *if there exists a partial assignment α_c such that for any full assignment α, $\alpha \in \mathcal{S} \longleftrightarrow \alpha_c \subseteq \alpha$. α_c is called the* cube *of $\mathcal{S}$.*

In other words, the assignments in a bunch $\mathcal{S}$ share a partial assignment α_c, and for the Boolean variables which are not assigned by the cube α_c, these assignments cover all possibilities of value combinations. The cardinality of $\mathcal{S}$ is exactly $2^{n-|\alpha_c|}$, where n is the number of Boolean variables, and $|\alpha_c|$ stands for the size of α_c.

For the assignments in a bunch, computing their total volume can be greatly simplified, as the following proposition reveals:

Proposition 1. *For a bunch $\mathcal{S}$ with the cube α_c, $\sum_{\alpha\in\mathcal{S}} volume(\alpha) = volume(\alpha_c)$.*

Consider the formula

$$\phi = (((y + 3x < 1) \rightarrow (30 < y)) \vee (x \leq 60)) \wedge ((30 < y) \rightarrow \neg(x > 3) \wedge (x \leq 60))$$

We first introduce a Boolean variable for each linear inequality and obtain its propositional skeleton as follows:

$$PS_\phi = ((b_1 \rightarrow b_2) \vee b_4) \wedge (b_2 \rightarrow \neg b_3 \wedge b_4)$$

where:

$$\begin{cases} b_1 \equiv (y + 3x < 1); \\ b_2 \equiv (30 < y); \\ b_3 \equiv (x > 3); \\ b_4 \equiv (x \leq 60); \end{cases}$$

Using an SMT solver, we find out that there are seven feasible assignments. Hence in the straightforward method, `vinci` is called seven times to compute the volumes of the feasible assignments, and they are added up to get the total volume of the formula. Here we list three of these assignments:

$$\begin{aligned} \alpha_1 &= \{\neg b_1, \neg b_2, b_3, \neg b_4\} \\ \alpha_2 &= \{\neg b_1, \neg b_2, \neg b_3, b_4\} \\ \alpha_3 &= \{\neg b_1, \neg b_2, b_3, b_4\} \end{aligned}$$

Now let's consider another assignment: $\alpha_4 = \{\neg b_1, \neg b_2, \neg b_3, \neg b_4\}$. It is easy to check that α_4 satisfies PS_ϕ, but $\hat{Th}(\alpha_4)$ is inconsistent in linear arithmetic theory. Thus $volume(\alpha_4) = 0$. Also, these four assignments form a bunch whose cube is $\{\neg b_1, \neg b_2\}$. Noticing this, we have

$$\begin{aligned} & volume(\alpha_1) + volume(\alpha_2) + volume(\alpha_3) \\ =\ & volume(\alpha_1) + volume(\alpha_2) + volume(\alpha_3) + volume(\alpha_4) \\ =\ & volume(\{\neg b_1, \neg b_2\}) \end{aligned}$$

As a result, we only need to call `vinci` once to compute the volume of cube $\{\neg(y + 3x < 1), \neg(30 < y)\}$ so as to obtain the total volume of α_1, α_2 and α_3. In contrast, without incorporating α_4 and combining assignments, three calls to `vinci` are needed.

Obviously, theory-inconsistent assignments, whether propositionally satisfying the formula or not, do not affect the total volume. This observation, together with Proposition 1, suggest a way to reduce number of calls to volume computing procedure. That is, we first list all feasible assignments with the help of an SMT solver and then make possible combinations to form bunches, counting in some theory-inconsistent assignments whenever necessary. However, an SMT solver doesn't provide explicitly theory-inconsistent assignments. In order to incorporate such assignments, extra calls to theory solvers are inevitable.

4.2 The Algorithm

Our method is based on the above idea. But rather than selecting theory-inconsistent assignments and then making possible combinations as a postprocessing step, we implemented it within the decision procedure of the SMT(LAC) solver. A typical SMT solver does not provide deduction procedure with respect to the specific theory for assignments that falsify the propositional skeleton of the formula. Therefore, we have to ignore the assignments in case (iv). In other words, the additional assignments possibly incorporated are those in case (ii).

A key point is that when the SMT solver finds a feasible assignment, we try to obtain a smaller one which still propositionally satisfies the formula. It is formally defined as follows:

Definition 2. *Suppose α is a feasible assignment for formula ϕ. An assignment α_{mc} is called a* minimum cube *of α if*

1. *$\alpha_{mc} \subseteq \alpha$ and $\alpha_{mc} \models PS_\phi$.*
2. *$\forall \alpha' (\alpha' \models PS_\phi \rightarrow \alpha' \not\subset \alpha_{mc})$.*

In fact, the minimum cube α_{mc} of an assignment α is the cube of a bunch $\mathcal{S}$ such that for any bunch $\mathcal{S}'$, $\alpha \in \mathcal{S}' \rightarrow \mathcal{S} \not\subset \mathcal{S}'$. Any assignment in $\mathcal{S}$ also satisfies PS_ϕ because only part of it, say α_{mc}, has evaluated PS_ϕ to be true. As we have explained before, it is pretty safe to count in such an assignment while computing the total volume, regardless of its consistency in the specific theory.

For a feasible assignment α and its minimum cube α_{mc}, from Proposition 1 we know that $volume(\alpha_{mc})$ includes $volume(\alpha)$, and possibly the volumes of other feasible assignments. Thus it seems rewarding to compute $volume(\alpha_{mc})$ instead of $volume(\alpha)$. Note that an assignment might have several minimum cubes. Currently we use a simple method to find only one minimum cube. It checks the redundancy of each literal contained in α sequentially. If α with a literal l_i removed still evaluates PS_ϕ to be true, then l_i is immediately deleted from α, and the next literal is checked with respect to the modified α. It can be easily proved that the final result is a minimum cube of the original assignment.

The SMT solving framework is adapted to compute the volume of a formula. Each time a feasible assignment is found, its minimum cube is computed and the volume of the minimum cube is added to the total volume. Then the negation of the minimum cube is added to the original formula so that a feasible assignment would not be counted more than once. It is a blocking clause, ruling out all the assignments in the bunch related to the minimum cube.

In a feasible assignment, some variables are decision variables, while others get assigned by Boolean constraint propagation (BCP). These implied literals need not be checked when finding the minimum cube of an assignment. (It will be proved in Proposition 2.) The detailed algorithm is presented as Algorithm 2.

Algorithm 2. Volume Computation in Bunches

```
Boolean Formula $PS = PS_\phi$;
$volume = 0$;
while TRUE do
  if BCP() == CONFLICT then
    backtrack-level = AnalyzeConflict();
    if backtrack-level < 0 then
      return $volume$;
    end if
    backtrack to backtrack-level;
  else
    $\alpha$ = current assignment;
    if $\alpha \models PS$ then
      if $\hat{Th}(\alpha)$ is inconsistent then
        backtrack to the latest decision variable;
      else
        for all literal $l_i \in \alpha$ do
          if $l_i$ is a decision variable or its negation then
            $\alpha' = \alpha - \{l_i\}$;
            if $\alpha' \models PS$ then
              $\alpha = \alpha'$;
            end if
          end if
        end for
        $volume$+=VOLcompute($\alpha$);
        Add $\neg\alpha$ to $PS$;
      end if
    else
      choose a Boolean variable and extend the current assignment;
    end if
  end if
end while
```

Proposition 2. *Given a feasible assignment α and a literal l_i in α, if l_i is not a decision variable or its negation, then it must appear in any minimum cube of α.*

Proof. We prove it by contradiction. Suppose α_{mc} is a minimum cube of α and $l_i \notin \alpha_{mc}$, then α_{mc} can be extended to a full assignment α', which is the same as α except that l_i is replaced with $\neg l_i$. Since $\alpha' - \{\neg l_i\} = \alpha - \{l_i\}$ together with the current Boolean formula PS implies that l_i must be true, we know that α' falsifies PS, and that α_{mc} cannot be a minimum cube. □

Theorem 1. *Algorithm 2 computes the volume of formula ϕ.*

Proof. Since the volume of a formula ϕ is the total volume of all feasible assignments, we shall show that: any feasible assignment to ϕ is counted into the total volume exactly once, and no non-empty volume of an assignment that propositionally falsifies ϕ is introduced.

Firstly, assume a feasible assignment α is missing while computing the volume of ϕ. Denote the new Boolean formula when the program terminates by PS'. Clearly $PS' = PS_\phi \wedge C$, where C stands for all the blocking clauses. In fact, $C = \bigwedge \neg c$, where c ranges over all minimum cubes discovered by the algorithm. Since α is missing, it cannot be obtained by extending any minimum cube c. Thus we have $\alpha \models \neg c$ and further $\alpha \models C$. As a feasible solution to ϕ, α propositionally satisfies ϕ, i.e., $\alpha \models PS_\phi$. Therefore, we have $\alpha \models PS'$, contradicting the termination condition that there is no assignment that satisfies PS'.

Secondly, because of the blocking clauses, a feasible assignment would not be counted more than once.

Finally, it is quite clear that a theory-consistent assignment which falsifies ϕ propositionally can't be extended from any minimum cube and is impossible to be counted in while computing the total volume. □

We would like to further clarify that

Proposition 3. *The number of calls to volume computing procedure in Algorithm 2 is not more than the straightforward method.*

This is quite clear since each minimum cube represents at least one feasible assignment, so the proof is omitted.

4.3 Incorporating Theory-Level Conflict-Driven Learning

Conflict-driven Learning with respect to the specific theory is an important technique for efficient SMT solving. It has been adopted by most of the current state-of-the-art SMT solvers [3,9,18]. When presenting Algorithm 2, we omit this technique for clarity. In this subsection, we shall give a brief introduction to this technique and explain that it can be easily integrated into the framework of our algorithm.

It is known that efficient lemma learning based on conflict analysis contributes greatly to modern SAT solvers. For an SMT solver which combines a DPLL-style SAT solver and a decision procedure for a conjunctive fragment of a theory T, lemma learning could go beyond the propositional level. When the decision procedure discovers that the current assignment α is inconsistent in T, it tries to explain the inconsistency, finding out the literals in α that lead to the conflict. The disjunction of negations of these literals is then obtained and passed to the SAT solver as a lemma. The lemma is learned by analyzing the inconsistency, and it prevents the same inconsistency from happening again. As a result, the search space is pruned.

Here we study two methods for incorporating lemma learning technique into Algorithm 2.

Method 1. Each time the current assignment α is found theory-inconsistent, a lemma is derived and added to the current Boolean formula PS. The rest of Algorithm 2 is unchanged. The modified algorithm is still correct. The proof is similar to that of Theorem 1: Suppose a feasible assignment of ϕ, denoted by α, is missing when the program terminates. At the termination the Boolean formula PS' consists of three parts: PS_ϕ, blocking clauses C, and lemmas learned from the theory inconsistencies LM. Since α is theory-consistent, for any lemma $cl \in LM$ we have α falsifies $\neg cl$ or equivalently $\alpha \models cl$. Consequently $\alpha \models LM$. From the proof of Theorem 1 we already know that $\alpha \models PS_\phi \wedge C$, thus we have $\alpha \models PS'$. So α cannot be missing when the program terminates. The rest of the proof is the same as that of Theorem 1. Furthermore, it is obvious that Proposition 2 and Proposition 3 still hold.

Method 2. When some lemmas are learnt from theory inconsistency, instead of adding them to the Boolean formula PS, we put them apart from PS. Hence these lemmas participate in the process of finding a feasible assignment α, while they do not affect finding a minimum cube of α. Proposition 2 no longer holds in this situation. For a counterexample, consider the following formula:

$$\phi = (x > 3 \vee x < 6) \wedge (x < 6 \vee x < 2) \wedge (x < 2)$$

Suppose the search procedure has made two decision assignments: $x < 2, x > 3$. Since they are theory-inconsistent, a lemma $\neg(x < 2) \vee \neg(x > 3)$ is derived and the program backtracks to the only decision assignment $x < 2$. Two literals, $\neg(x > 3)$ and $x < 6$ are then implied by BCP. A feasible assignment is now obtained, with a minimum cube $\{x < 2, x < 6\}$. Now the literal $\neg(x > 3)$ is neither a decision variable nor its negation, but it is not contained in the minimum cube, contradicting Proposition 2. As a result, in this method, all literals in the feasible assignment have to be checked for redundancy.

5 Implementation and Experimental Results

The algorithms were implemented on the basis of the SAT solver `MiniSat` 2.0 [11], which serves as the search engine for the Boolean structure of the SMT(LAC) instance. The linear programming tool `lp_solve` [5] is integrated for deciding the consistency of a conjunction of linear constraints. When a minimum cube is obtained, the program calls `vinci` to compute the volume of the corresponding polytope, or `azove` / `LattE` to count the number of integer points in the polytope.

To study the effectiveness of the aforementioned techniques, we randomly generated a number of SMT(LAC) instances whose propositional skeletons are in CNF. Since `vinci` is very slow for polytopes in more than 8 dimensions, each instance contains no more than 8 numeric variables.

Table 1. Comparison of Algorithms

		Algorithm 2				Algorithm 1	
		Method 1		Method 2			
Instance	P cls V	Time (s)	#calls	Time (s)	#calls	Time (s)	#calls
Ran1	8 50 4	0.01	19	0.02	19	0.03	41
Ran2	10 40 5	0.06	53	0.06	50	0.14	182
Ran3	15 40 5	5.35	57	2.36	47	11.12	188
Ran4	20 40 5	106.63	332	116.72	259	431.15	17158
Ran5	10 20 6	1.06	47	1.04	41	7.81	212
Ran6	10 50 6	1.07	74	2.08	74	5.32	247
Ran7	15 50 6	2.12	52	2.15	57	10.97	257
Ran8	7 40 7	1.01	16	1.01	16	2.77	39
Ran9	12 40 7	51.26	245	50.29	250	502.75	1224
Ran10	15 50 7	314.26	833	303.14	856	3872.70	5224
Ran11	20 50 7	214.13	158	143.11	140	1889.36	807
Ran12	10 20 8	13.04	39	12.04	37	150.92	235
Ran13	10 40 8	51.09	91	51.10	91	398.02	379
Ran14	16 80 8	1104.05	648	1074.48	669	4 hours	4273

`P`: number of linear constraints. `cls`: number of clauses.
`V`: number of numerical variables. `#calls`: number of calls to `vinci`.

We have implemented both methods for the theory-level conflict-driven learning mechanism described in subsection 4.3. We compare them with the straightforward approach on the random instances. The experimental results are given in Table 1. The programs are run on an Intel 1.86GHZ Core Duo 2 PC with Fedora 7 OS. For a problem instance with 7 or 8 numerical variables, if we use `azove`, the running time will be too long. So the table only contains data for the cases when `vinci` is called.

From Table 1 we can see that both the running time and the number of calls are reduced with Algorithm 2, in some cases by an order of magnitude. However, there is no clear winner between the two methods for lemma learning.

We have also tried some benchmark problems in SATLIB[2]. It turns out that some SAT instances (e.g., `Bejing`) have so many solutions that the whole search space cannot be exhausted in reasonable time. On the other hand, some other instances have just a few solutions (even one solution per instance), or the solutions are scattered in the multi-dimensional space. Table 1 does not include data for the above instances.

6 Application to Program Analysis

In [25], we proposed a measure $\delta(P)$ for a program path P. Intuitively, it denotes the volume of the subspace in the input space of the program that corresponds to the execution of path P; or, in other words, the number of input data vectors

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

which drive the program to be executed along this path. Obviously, a special case is that P is infeasible, and $\delta(P) = 0$.

For a simple example, see the following program:

```
int i, j;
if(i+j > 10)
    j = 2;
else
    j = 1;
```

The program has two paths, denoted by P_1 and P_2, which correspond to the `if-then` branch and the `else` branch, respectively. The path conditions are $(i + j > 10)$ and $(i + j \leq 10)$.

Suppose the input variables i, j take values from the interval [1..10]. We can compute that $\delta(P_1) = 55$, $\delta(P_2) = 45$. Now suppose the variables can take values from a larger interval, e.g., [1..100]. Then we have $\delta(P_1) = 9955$, $\delta(P_2) = 45$. We can see that the first path is executed much more frequently than the second path, under reasonable assumptions of the input data space.

6.1 Execution Probability

For many paths, the path condition involves just a subset of the input variables. In such a case, we can preprocess the constraints, and remove the irrelevant variables first. This will reduce the dimension of the solution space and also the volume computation time. The following is a contrived example of programs:

```
int i, j;
int a[10], b[10];
for(i = 0; i < 10; i++)
   for(j = 0; j < 10; j++)
      if(a[i] < b[j])
         return a[i];
```

There are many paths in the program's flow graph. The following is the path condition for one of them:

$$(a[0] \geq b[0]) \wedge (a[0] \geq b[1]) \wedge (a[0] < b[2])$$

The condition involves just 4 (among 20) array elements explicitly. So when computing the δ value of the path, we omit the other 16 array elements, assuming the input space is of 4 dimensions rather than 22 (20 array elements and `i`, `j`). The computation is greatly simplified.

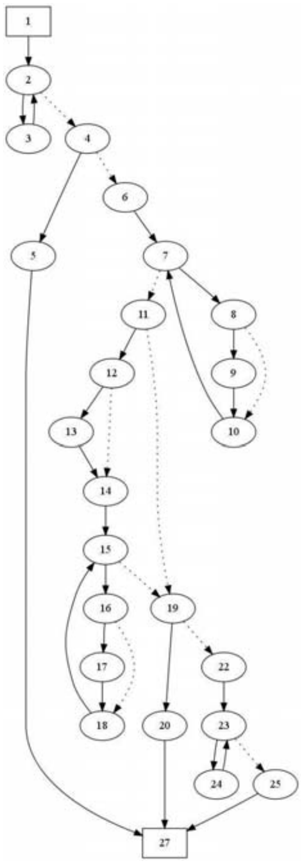
However, a problem arises when two paths involve different variables, since the δ values are no longer comparable. We have to introduce another measurement, namely *execution probability*. The execution probability of a path P, denoted by $\mathcal{XP}(P)$, is defined as $\delta(P)$ divided by the volume of the involved data space.

Suppose there are m variables in the path condition of P, and the range length or domain size of the ith variable is l_i. We have

$$\mathcal{XP}(P) = \frac{\delta(P)}{\prod_{1 \le i \le m} l_i}.$$

6.2 A Practical Example

Now we describe our experiments with a real program, i.e., a function called *getop()* which is taken from [15]. It has been used as an example in several research papers on software testing. The function fetches the next operator or operand for a calculator program. Its input variables are characters (except for one integer variable). It has `for`-loops, `while`-loops, and `if` statements. The conditional expressions in these statements contain logical operators. This makes the path conditions a bit complex. The control flow graph is demonstrated in Fig. 1. Each node in the graph represents a conditional expression or a block



```
1:  c=getchar();
2:  !(((c==32 || c==9) || c==10))
3:  c=getchar();
4:  !((c!=46 && (c<48 || c>57)))
5:  RETURN: c;
6:  s[0]=c; c=getchar(); i=1;
7:  !((c>=48 && c<=57))
8:  !(i<lim)
9:  s[i]=c;
10: c=getchar(); _temp_var0=i; i=i+1;
11: !(c==46)
12: !(i<lim)
13: s[i]=c;
14: c=getchar(); _temp_var1=i; i=i+1;
15: !((c>=48 && c<=57))
16: !(i<lim)
17: s[i]=c;
18: c=getchar(); _temp_var2=i; i=i+1;
19: !(i<lim)
20: s[i]=0; RETURN: 1000;
22: NOP
23: !((c!=10 && c!=-1))
24: c=getchar();
25: _temp_var3=lim-1; s[_temp_var3]=0; RETURN: 9999;
27: END
```

Fig. 1. getop()

of statements, as shown on the right hand side of Fig. 1. At a branching node, the dotted arrow represents the `true` branch, and the solid arrow represents the `false` branch. The flow graph and the expressions/blocks are produced by our test generation tools. In this process, some auxiliary variables like `_temp_var2` are introduced.

We obtained several paths from the program, and computed the path condition for each path. The execution probability of each path was computed by our tool quickly. The running times are within 0.03 second.

In the experiment, we assume that each character variable is an integer, taking values from the range $[0, 255]$. We can see that some paths have a larger probability of being executed, as compared with other paths. For example, see the two paths: $Path1$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 27$ and $Path2$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 11 \rightarrow 19 \rightarrow 20 \rightarrow 27$. With our tool, we found that $\mathcal{XP}(Path1) \approx 0.945$ and $\mathcal{XP}(Path2) \approx 0.021$.

7 Related Works and Discussion

So far as we know, there is little work in the literature concerning volume computation for SMT. In contrast, counting the number of models for SAT has become a hot topic in the AI community recently. However, some advanced model counting techniques (e.g., component analysis [14] and caching [22]) cannot be used directly here. The main reason is that the Boolean variables in the SMT formulas are not independent of each other.

In the software engineering community, Buse and Weimer [7] proposed a method for *estimating* path execution frequency. They give a statistical model, which is based on *syntactic* features of the program's source code. In contrast, our approach uses *semantic* information in the program paths, and it can calculate the *exact* probability of executing a path.

In our approach, the domain of linear arithmetic is separated from the logic part. An alternative is to translate the fixed-length integer variables into Boolean variables, and then using a pure Boolean logic approach. This should have some advantages when the program to be analyzed has both arithmetic operations and bit operations. However, its scalability deserves further investigation.

The main topic of our paper is the volume (or solution space size) of SMT formulas. But as illustrated in the previous section, sometimes it is desirable to talk about the probability of satisfying the formulas. So we can introduce the notion of *satisfying probability* for a formula. Assume that each variable takes values uniformly from its range. Given a formula ϕ, dividing V_ϕ by the size of the whole space, we shall get the probability that ϕ could be satisfied, which is denoted by $\mathcal{P}(\phi)$.

We have been discussing how to compute V_ϕ assuming that the linear constraints in ϕ are restricted to one data type, either integer or real. Now we extend our problem a bit further. Suppose there are two kinds of linear constraints in the formula ϕ: some are defined over p integer variables with the domain size l_I, and the others are defined over q real variables with the range length l_R. The satisfying probability of ϕ is defined as follows:

$$\mathcal{P}(\phi) = \sum_{\alpha \in Mod(\phi)} \frac{volume(\alpha_I)}{l_I^p} \times \frac{volume(\alpha_R)}{l_R^q}$$

where α_I is the partial assignment to integer constraints, and α_R to real constraints. It can be easily checked that this probability is well defined. Algorithm 2 can also be easily modified to compute $\mathcal{P}(\phi)$.

If a linear constraint in formula ϕ involves both integer and real variables, the explicit expression of $\mathcal{P}(\phi)$ is not clear. Our algorithm is not applicable to such cases, either. But we find that the volume of a polytope is quite close to the number of integer points it contains except for some special cases. Thus in practice, we can use type casting to unify the data types and get an approximate result. Despite this, we consider this issue as an interesting problem, worthy of further investigation.

8 Concluding Remarks

This paper studies the automatic computation of the volume (or solution space size) of SMT instances. We proposed a non-trivial algorithm, and augmented it with theory-level lemma learning technology. The algorithms have been implemented, and some experimental results are presented.

In this paper, we focus on just one application of volume computation techniques. Our experiences show that they can be used to compare program paths with respect to their execution frequency, for real programs. We believe that the techniques can be applied to other problems as well.

There are several ways in which the current approach might be improved. Firstly, Algorithm 2 is an on-the-fly method. The bunches are derived as the program runs, and the number of bunches is not predictable. It would be better if there were a search strategy for finding a small number of bunches without too much extra cost. Secondly, as we mentioned in Section 4, we are not able to incorporate the assignments in case (iv) at present. We plan to devise a post-processing technique to incorporate some of them when necessary. As a result, some of the bunches might be combined. Finally, we will investigate the best way of incorporating lemma learning into our framework.

Acknowledgement

The authors would like to thank the anonymous referees for their helpful comments and suggestions. Runming Lu helped us implement the prototype tool, and Hui Ruan helped in the preparation of the example in Sec. 6.2.

References

1. Andrei, S., Chin, W., Cheng, A.M.K., Lupu, M.: Automatic debugging of real-time systems based on incremental satisfiability counting. IEEE Trans. Computers 55(7), 830–842 (2006)
2. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003), pp. 340–351 (2003)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007), `http://www.cs.nyu.edu/acsys/cvc3`
4. Behle, M., Eisenbrand, F.: 0/1 vertex and facet enumeration with BDDs. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX 2007) (2007), `http://www.mpi-inf.mpg.de/~behle/azove.html`
5. Berkelaar, M., Eikland, K., Notebaert, P.: The lp_solve Page, `http://lpsolve.sourceforge.net/`
6. Büeler, B., Enge, A., Fukuda, K.: Exact volume computation for polytopes: a practical study. Polytopes–combinatorics and computation (1998), `http://www.lix.polytechnique.fr/Labo/Andreas.Enge/Vinci.html`
7. Buse, R.P.L., Weimer, W.R.: The road not taken: Estimating path execution frequency statically. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009) (2009)
8. Davies, J., Bacchus, F.: Using more reasoning to improve #SAT solving. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 185–190 (2007)
9. Dutertre, B., Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006), `http://yices.csl.sri.com/`
10. Dyer, M., Frieze, A.: On the complexity of computing the volume of a polyhedron. SIAM J. Comput. 17(5), 967–974 (1988)
11. Eén, N., Sorensson, N.: The MiniSat Page, `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
13. Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2293–2299 (2007)
14. Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence (AAAI/IAAI 2000), pp. 157–162 (2000)
15. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice-Hall, Englewood Cliffs (1978)
16. Kroening, D., Strichman, O.: Decision Procedures — An Algorithmic Point of View. Springer, Heidelberg (2008)
17. Loera, J.A.D., Hemmeckeb, R., Tauzera, J., Yoshidab, R.: Effective lattice point counting in rational convex polytopes. Journal of Symbolic Computation 38(4), 1273–1302 (2004), `http://www.math.ucdavis.edu/~latte/`

18. Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008), `http://research.microsoft.com/projects/z3/index.html`
19. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
20. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. 82(1-2), 273–302 (1996)
21. Samer, M., Szeider, S.: Algorithms for propositional model counting. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 484–498. Springer, Heidelberg (2007)
22. Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 226–240. Springer, Heidelberg (2005)
23. Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 324–339. Springer, Heidelberg (2005)
24. Zhang, J.: Specification analysis and test data generation by solving boolean combinations of numeric constraints. In: Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS 2000), pp. 267–274 (2000)
25. Zhang, J.: Quantitative analysis of symbolic execution. Presented at the 28th International Computer Software and Applications Conference (COMPSAC 2004) (2004)
26. Zhang, J., Wang, X.: A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge Engineering 11(2), 139–156 (2001)