VolCE v1.0.1

Cunjing Ge, Feifei Ma and Jian Zhang
Institute of Software
Chinese Academy of Sciences
Email: {gecj,maff,zj}@ios.ac.cn

Feb. 2015

1 Introduction

1.1 What is VolCE?

VolCE is designed for computing or estimating the size of the solution space of an SMT formula where the theory T is restricted to the linear arithmetic theory. (SMT stands for Satisfiability Modulo Theories.) The prototype tool presented in [2] computes the exact volume of the solution space. However, exact volume computation in general is an extremely difficult problem. It has been proved to be #P-hard, even for explicitly described polytopes. On the other hand, it suffices to have an approximate value of the volume in many cases. Later we implemented a tool to estimate the volume of polytopes [1] and integrated it into the framework of [2]. The new tool is called VolCE. It can efficiently handle instances of dozens of dimensions with high accuracy. In addition, VolCE also accepts constraints involving independent Boolean variables.

1.2 What can VolCE do?

VolCE uses the following three packages:

- PolyVest [1], which can be used to estimate the volume of polytopes
- Vinci [6], a software package that implements several algorithms for (exact) volume computation.
- LattE (Lattice point Enumeration) [3], a software package dedicated to the problems of counting lattice points and integration inside convex polytopes.

2 Installation

- Step 1: Make sure that g++ (version 4.8 or higher version) is installed on your machine (you can type "`g++ -v`" to check this).
- Step 2: The functionality of `VolCE` is dependent on some other libraries: `zlib`, `boost`, `lpsolve`, `glpk`, `gfortran`, `LAPACK`, `BLAS` and `Armadillo`. On Ubuntu or Debian, you can use "apt-cache search" and "apt-get install" to find and install all of these libraries. Or you can download these libraries from:

Library	URL
`zlib`	http://zlib.net/
`boost`	http://www.boost.org/
`lpsolve`	http://sourceforge.net/projects/lpsolve/
`glpk`	http://www.gnu.org/software/glpk/
`gfortran`	http://gcc.gnu.org/fortran/
`LAPACK`	http://www.netlib.org/lapack/
`BLAS`	http://www.netlib.org/blas/index.html
`Armadillo`	http://arma.sourceforge.net/

  Note. For `lpsolve`, make sure that its header files and the dynamic library (i.e., `liblpsolve55.so`) are included in the directories "`/usr/include/lpsolve`" and "`/usr/lib/lp_solve`", respectively. Besides, `LAPACK` and `BLAS` should be installed before installing `Armadillo`.

- Step 3: Open a shell (command line), change into the directory that was created by unpacking the `VolCE` archive, and type:

  ```
  sh build.sh
  ```

  When the build process is finished, you will find all binaries in the directory "`release/`".
- Step 4: Install `LattE` [3]. Build and move the executable files (`count` and `scdd_gmp`) into directory "`release/bin/`".

This release of `VolCE` has been successfully built on the following operating systems:

- Ubuntu 12.04 on 64-bit with g++ 4.8.1
- Ubuntu 13.10 on 32-bit with g++ 4.8.1

3 Input Format

The input of `VolCE` is an SMT formula where the theory T is restricted to the linear arithmetic theory. It involves variables of various types (including integers, reals and Booleans). We usually use b_i to denote Boolean variables, x_j to denote numeric variables. In the input formula, there can be logical operators (like AND, OR, NOT), arithmetic operators (like addition, subtraction, scalar multiplication) and comparison operators (like $<, \leq, >, \geq, =, \neq$).

Syntactically, there are two formats for the input file:

- `VolCE` style
- `SMT-LIBv2` [4]

We now describe them in detail.

3.1 `VolCE` Style Input Format

Let us introduce some concepts first.

- *LAC*: A linear arithmetic constraint (LAC) is a comparison between two linear arithmetic expressions. Such a constraint can be denoted by a Boolean variable (e.g., $b_3 \equiv x_1 + x_2 \leq 1$).
- *literal*: A literal is either a Boolean variable (e.g., b_3) or a negated Boolean variable (e.g., NOT b_3).
- *clause*: A clause is a set of one or more literals, connected with OR. (Boolean variables may not be repeated inside a clause.)
- $formula$: A formula is a set of one or more clauses, connected with AND.

It is well known that any Boolean expression can be converted into the conjunctive normal form (CNF) easily (e.g., using the Tseitin transformation [5]). So the input of `VolCE` is a formula in the CNF form, where each Boolean variable may stand for some LAC. The input file generally consists of two parts: LACs, and clauses in the CNF.

An example of `VolCE` style formula is the following:

$$
\begin{aligned}
& b_1 \equiv x_1 < x_2, \\
& b_3 \equiv x_1 + x_2 < 1, \\
& b_4 \equiv x_1 \leq 1, \\
& b_5 \equiv x_2 \leq 1, \\
& b_6 \equiv x_1 \geq 0, \\
& b_7 \equiv x_2 \geq 0, \\
& (b_1 \text{ OR } (\text{NOT } b3)) \text{ AND} \\
& (b_1 \text{ OR } (\text{NOT } b2) \text{ OR } b3) \text{ AND} \\
& ((\text{NOT } b3) \text{ OR } b4) \text{ AND} \\
& b_4 \text{ AND } b_5 \text{ AND } b_6 \text{ AND } b_7.
\end{aligned}
\tag{1}
$$

There are 7 Boolean variables (b_1, ..., b_7), 2 numeric variables (x_1 and x_2), 6 LACs and 7 clauses in Formula 1. Note that b_2 is an independent Boolean variable which does not represent any LAC.

Syntactically, `VolCE` accepts input in an "Enhanced DIMACS CNF Format". Every line beginning with "`c`" is a comment. The first non-comment line must be of the form:

```
p cnf v lc BOOLS CLAUSES NUMVARS LACS
```

It specifies the number of Boolean variables, the number of clauses, the number of numeric variables and the number of linear constraints.

Every line beginning with "`m`" defines a linear constraint and its corresponding Boolean variable. It must be of the form:

```
m i a1 ... an op b
```

It defines a linear inequality $a_1x_1 + \cdots + a_nx_n$ *op* b, where a_1, ..., a_n, b are constants, and *op* is a comparison operator: <, <=, >, >= or =. (The tool does not support $\neq$ directly. However, $ax \neq b \Leftrightarrow$ NOT $ax = b$.) The number i means the Boolean variable b_i represents this inequality. The space between the character "`m`" and the number i is not mandatory.

Each of the other lines defines a clause: a positive literal is denoted by the corresponding number (so `4` means b_4), and a negative literal is denoted by the corresponding negative number (so `-5` means NOT b_5). The last number in the line should be zero. Each of these lines is a space-separated list of numbers.

So the above Formula 1 would be written in the following way:

```
c It is an example, f1.vs.
p cnf v lc 7 7 2 6
c Linear Constraints part.
m1 1 -1 < 0
m3 1 1 < 1
m4 1 0 <= 1
m5 0 1 <= 1
m6 1 0 >= 0
m7 0 1 >= 0
c CNF part.
1 -3 0
1 -2 3 0
-1 3 0
4 0
5 0
6 0
7 0
```

See the file `examples/f1.vs`.

3.2 The SMT-LIBv2 Language Inputs

`VolCE` also partially supports the `SMT-LIBv2` language. For details of this language, visit the website:

`http://www.smt-lib.org/`

`VolCE` recognizes `SMT-LIBv2` format from the file name extension "`.smt2`". It automatically parses such a file into the `VolCE` style input.

Table 1 lists the commands, variable types and identifiers of `SMT-LIBv2` language that supported by `VolCE`. `VolCE` ignores some basic commands like `set-logic`, `set-info`, `check-sat`, `exit`. It directly checks all of the assertions. Besides, `assert` commands must be written after all the `declare-fun` commands.

Table 1: Supported SMT-LIBv2 Components

Commands	`declare-fun assert`
Variable Types	`Int Real Bool`

Identifiers	`let`
	`and or not => ite`
	`+ - * /`
	`= > >= < <= distinct`

In the `SMT-LIBv2` language, the above Formula 1 would be written like this:

```
(set-logic QF_LRA)
(set-info :f1.smt2)
(set-info :smt-lib-version 2.0)
(set-info :status sat)
(declare-fun x () Real)
(declare-fun y () Real)
(declare-fun b () Bool)
(assert (and (<= x 1) (<= y 1) (>= x 0) (>= y 0)))
(assert (let ((v1 (< (+ x y) 1)) (v2 (< x y)))
(and (or v1 (not v2)) (or v1 v2 b) (or (not v1) v2))))
(check-sat)
(exit)
```

See the file `examples/f1.smt2`.

4 Running `VolCE`

To run `VolCE`, you should switch your working directory to the absolute path of `VolCE`. `VolCE` has a help menu. To view it, simply type the command `"./volce --help"`. The general usage of `VolCE` is

```
% ./volce [OPTION]... <INPUT-FILE>
```

The meanings of the options are given in the following table.

Table 2: Command-line Options of `VolCE`

Option	Meaning
`-P`	Enables `PolyVest` for volume approximation. The input variables in the linear inequalities are reals. By default, `VolCE` calls `PolyVest`. It assumes that all the numeric variables are reals.
`-V`	Enables `Vinci` for volume computation. The input variables in the linear inequalities are reals.
`-L`	Enables `LattE` to count the number of integer solutions. The input variables in the linear inequalities should be integers. This option is usually enabled in the case of integer variables.
`-w=NUMBER`	Specifies the word length of numeric variables in bit-wise representations. Then each variable is automatically bounded by the range $[-2^{w-1}, 2^{w-1}-1]$. Setting the word length to 0 will disable this feature. By default, the word length is 8, which means the domain of every numeric variable is $[-128, 127]$. For example, you can change it to 3, by using the option `-w=3`.

`-maxc=NUMBER`	Sets the maximum sampling coefficient of `PolyVest`, which is an upper bound. Generally, the larger this coefficient is, the more accurate the result will be. However, the running time of the tool will be longer. The default value of `maxc` is 1600. For example, you may change it to 1000, by using the option `-maxc=1000`.
`-minc=NUMBER`	Sets the minimum sampling coefficient of `PolyVest`. The default value is 40.

To estimate the volume of the solution space of Formula 1, simply type:

```
% ./volce examples/f1.vs
```

Note that Formula 1 guarantees $0 \leq x_1, x_2 \leq 1$. So we can disable the internal bit-wise bounds of numeric variables, by setting the word length to 0:

```
% ./volce -w=0 examples/f1.vs
```

You can also enable `PolyVest`, `Vinci`, `LattE` at the same time:

```
% ./volce -P -V -L -w=0 examples/f1.vs
```

Remarks Several tools (`PolyVest`, `Vinci`, `LattE`) have been integrated which can be enabled for different situations. `Vinci` gives an accurate volume for a polytope; but it may have difficulty handling problem instances with more than 10 numeric variables. `PolyVest` gives approximate results, but it can deal with larger instances. `LattE` is good at counting the number of integer solutions. Sometimes, the first two tools can also be used for approximating the number of integer solutions.

5 Examples

Example 1 For the above Formula 1, we have two input files: `VolCE` style input (`f1.vs`) and `SMT-LIBv2` input (`f1.smt2`).

Execute the command:

```
% ./volce -P -V -L -w=0 examples/f1.smt2
```

And we obtain the result:

```
Enabled PolyVest.
Enabled Vinci.
Enabled LattE.
Set word length to 0.
Disabled default bounds since word l
ength <= 0.
VolCE Directory: ...
Working Directory: ...

===================================

Parsing smt2 file.
Reading Input.
Number of bool vars:     16
Number of clauses:       29
Number of numeric vars:  2
Number of linear constraints:  6

===================================

Branches: 2
SATISFIABLE

===================================
============ PolyVest ============
===================================

FIRST ROUND
0 0.222875 * 2
1 0.24338 * 1

SEC & LAST ROUND
0 1600 0.252037 * 2
1 1600 0.250964 * 1

Total approximation: 0.755039

===================================

The total volume (PolyVest): 0.75503
900


===================================
============== Vinci ==============
===================================

0.25000000 * 2
0.25000000 * 1

===================================

The total volume (Vinci): 0.75000000


===================================
============== LattE ==============
===================================

0 * 2
2 * 1

===================================

The total volume (LattE): 2
```

Analysis:

Figure 1 shows the linear constraints in Formula 1. The plane is splitted into 4 areas, since b_4, b_5, b_6, b_7 are always True. The pair $\{b_1, b_3\}$ determines the counted areas.

- Area I: $\{b_1 = \text{True}, b_3 = \text{True}\}$. It has no lattice points.
- Area II: $\{b_1 = \text{True}, b_3 = \text{False}\}$. It has 1 lattice point: $\{0, 1\}$.

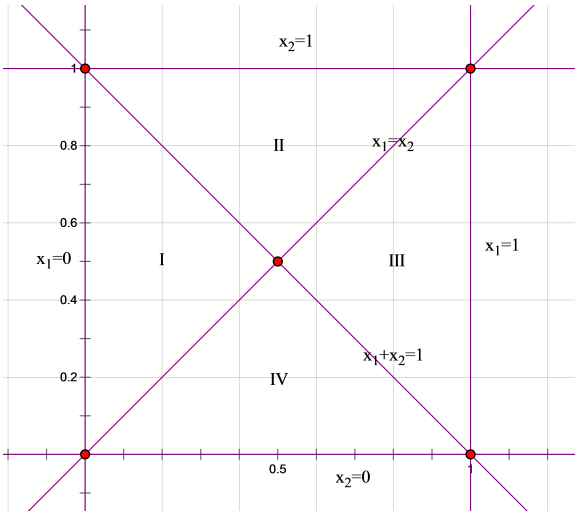
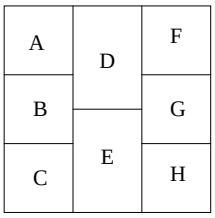


Figure 1: Solution Space of Formula 1

- Area III: $\{b_1 = \text{False}, b_3 = \text{False}\}$. It has 2 lattice points: $\{1, 0\}$ and $\{1, 1\}$.
- Area IV: $\{b_1 = \text{False}, b_3 = \text{True}\}$. It has 1 lattice point: $\{0, 0\}$.

There are 3 Boolean solutions for Formula 1: $\{b_1 = \text{True}, b_2 = \text{True}, b_3 = \text{True}\}$, $\{b_1 = \text{True}, b_2 = \text{False}, b_3 = \text{True}\}$, and $\{b_1 = \text{False}, b_2 = \text{True}, b_3 = \text{False}\}$. Thus the volume of the solution space is $2 \times vol(Area_I) + vol(Area_{III}) = 0.75$. And there are $2 \times 0 + 2 = 2$ integer solutions (lattice points).

Example 2 Here is an exercise for young pupils: In the following square, there are 8 sub-areas. Color them so that the neighboring sub-areas use different colors. How many different coloring schemes are there?



Obviously, this problem can be regarded as a solution counting problem. The input consists of the following inequalities:

$$xA \neq xB, \quad xA \neq xD, \quad xB \neq xC, \quad xB \neq xD, \quad xB \neq xE,$$

$$xC \neq xE, \quad xD \neq xE, \quad xD \neq xF, \quad xD \neq xG,$$
$$xE \neq xG, \quad xE \neq xH, \quad xF \neq xG, \quad xG \neq xH.$$

We assume that there are at most 4 colors, and execute the following command:

```
% ./volce -L -w=2 examples/coloring.smt2
```

We find that there are 768 solutions.

Example 3 In [2], we describe a program called *getop()* and analyze the execution frequency of its paths. For *Path1*, its path condition[1] is:

```
(NOT ((c = 32) OR (c = 9) OR (c = 10))) AND
((c != 46) AND ((c < 48) OR (c > 57)))
```

Here `c` is a variable of type `char`; it can be regarded as an integer variable within the domain [-128..127].

For the above path condition, we can compute the number of solutions by executing the command:

```
% ./volce -L examples/program_analysis/getopPath1.smt2
```

We find that the path condition has 242 solutions. (We do not need to use the option `-w=8`, because the default word length is 8.)

Given that the size of the whole search space is 256, we conclude that the frequency of executing *Path1* is about 0.945 (i.e., 242/256). This means, if the input string has only one character, most probably, the program will follow this path.

Another path, *Path2*, has the following path condition:

```
((c0 = 32) OR (c0 = 9) OR (c0 = 10)) AND
(NOT ((c1 = 32) OR (c1 = 9) OR (c1 = 10))) AND
(NOT ((c1 != 46) AND ((c1 < 48) OR (c1 > 57)))) AND
(NOT ((c2 >= 48) AND (c2 <= 57))) AND
(NOT (c2 = 46))
```

Given this set of constraints, and using `LattE`, our tool tells us that the number of solutions is 8085. The executed command is:

```
% ./volce -L examples/program_analysis/getopPath2.smt2
```

So the path execution frequency is $8085/(256 * 256 * 256)$ which is roughly 0.00048.

[1]The path condition is a set of constraints such that any input data satisfying these constraints will make the program execute along that path.

Example 4 Hoare's program FIND takes an array A[N] and an integer f as input. It tries to find the element of the array, whose value is the f-th in order of magnitude; and to rearrange the array in such a way that this element is placed in $A[f]$, all elements with subscripts lower than f have smaller values, and all elements with subscripts greater than f have larger values.

Assume that $N = 8$. We may extract two execution paths from the program, and generate the path conditions. The first path condition is the following:

```
(A[0] < A[3]);  !(A[1] < A[3]);  (A[3] < A[7]);
!(A[3] < A[6]);  !(A[2] < A[3]);  !(A[3] < A[5]);
!(A[3] < A[4]);  (A[0] < A[4]);  (A[6] < A[4]);  (A[5] < A[4]).
```

Setting the word length to 4, we can find that the number of solutions is 4075920. The executed command is:

```
% ./volce -L -w=4 examples/program_analysis/FINDpath1.smt2
```

The second path condition is a bit more complicated:

```
!(A[0] < A[3]);  (A[3] < A[7]);  (A[3] < A[6]);
(A[3] < A[5]);  (A[3] < A[4]);  !(A[1] < A[3]);
(A[3] < A[2]);  (A[3] < A[1]);  (A[1] < A[0]);
(A[2] < A[0]);  !(A[0] < A[7]);  !(A[4] < A[0]);
(A[0] < A[6]);  !(A[0] < A[5]);  (A[1] < A[7]);
(A[2] < A[7]);  !(A[7] < A[5]);  !(A[1] < A[5]);
!(A[2] < A[5]);  (A[5] < A[2]);  (A[2] < A[1]).
```

Executing the command:

```
% ./volce -L -w=4 examples/program_analysis/FINDpath2.smt2
```

we find that the number of solutions is 87516. So, the first path is executed much more frequently than the second one. (We assume that the input space is evenly distributed.)

Example 5 Let us try a randomly generated example (`ran_5_20_8.vs`). It has 5 Boolean variables, 8 numeric variable, 20 clauses and 5 linear constraints.

Execute the command:

```
% ./volce -P -V -w=4 examples/ran/ran_5_20_8.vs
```

And we obtain the result:

```
Enabled PolyVest.
Enabled Vinci.
Set word length to 4.
VolCE Directory: ...
Working Directory: ...

==================================

Reading Input.
Number of bool vars:     5
Number of clauses:       20
Number of numeric vars:  8
Number of linear constraints:  5

==================================

Branches: 1
SATISFIABLE

==================================
============= PolyVest =============
==================================

FIRST ROUND
0 8.38972e+06 * 1

SEC & LAST ROUND
0 1600 7.88093e+06 * 1

Total approximation: 7.88093e+06

==================================

The total volume (PolyVest): 7880930
.00000000

==================================
============== Vinci ===============
==================================

7970738.22355500 * 1

==================================

The total volume (Vinci): 7970738.22
355500
```

References

[1] C. Ge, F. Ma and J. Zhang. A fast and practical method to estimate volumes of convex polytopes. Dec. 2013. http://arxiv.org/abs/1401.0120v1

[2] F. Ma, S. Liu and J. Zhang. Volume computation for Boolean combination of linear arithmetic constraints. In: *Proc. CADE-22*, LNCS 5663, pp.453–468, 2009.

[3] LattE, available at https://www.math.ucdavis.edu/~latte/

[4] SMT-LIB: The Satisfiability Modulo Theories Library. http://www.smt-lib.org/

[5] G.S. Tseitin. On the complexity of derivation in propositional calculus. In: *Slisenko, A.O. (ed.) Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*, pp.115–125, Steklov Mathematical Institute, 1968.

[6] Vinci, available at
http://www.math.u-bordeaux1.fr/~aenge/?category=software&page=vinci