# Performance Estimation Using Symbolic Data

Jian Zhang

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zj@ios.ac.cn, jian_zhang@acm.org

**Abstract.** Symbolic execution is a useful technique in formal verification and testing. In this paper, we propose to use it to estimate the performance of programs. We first extract a set of paths (either randomly or systematically) from the program, and then obtain a weighted average of the performance of the paths. The weight of a path is the number of input data that drive the program to execute along the path, or the size of the input space that corresponds to the path. As compared with traditional benchmarking, the proposed approach has the benefit that it uses more points in the input space. Thus it is more representative in some sense. We illustrate the new approach with a sorting algorithm and a selection algorithm.

## 1 Introduction

For an algorithm (or program), its efficiency is very important. This is often characterized by the worst-case or average-case complexity. But we may not know the exact complexity for every program we write. Sometimes the complexity of algorithms is difficult to analyze, even for experts. For example, the Boyer-Moore string matching algorithm was published in 1977 [1]. More than 10 years later, Richard Cole [2] showed that the algorithm performs roughly $3n$ comparisons.

Even when we know the complexity, the actual performance may be different. As a matter of fact, some polynomial-time algorithms do not have good performance in practice. For two algorithms, even though they both have polynomial-time complexity, their execution times may be different due to the difference in constant factors. Thus benchmarking or empirical evaluation is still necessary in most cases. We run the program for a certain number of times, usually with random input data; and analyze the results (e.g., execution times of the program).

In this paper, we propose a *symbolic benchmarking* approach, in which the program/algorithm is executed symbolically. A benefit of this approach is that one symbolic execution corresponds to many actual executions. The performance is measured in terms of execution times or some performance indicators (e.g., the number of comparisons of array elements).

## 2 Basic Concepts and Related Works

A program (algorithm) can be represented by its flow graph, which is a directed graph. From the graph, one may generate many (or infinite number of) paths. Given a path, it is called *feasible* or *executable* if there is some input vector which makes the program execute along the path. Otherwise, it is called *infeasible* or *non-executable.* For a feasible path, there may be many different input data which make the program execute along the path. The input data can be characterized by a set of constraints, called the *path condition.* The path is feasible if and only if the path condition is satisfiable. Each solution of the path condition, when given as the initial input to the program, will make the program execute along the path.

Let us look at a simple example. The following C program computes the greatest common divisor (gcd) of two positive integers (`m` and `n`).

```
int GCD(int m, int n)
{
    x = m;              /* 1 */
    y = n;              /* 2 */
    while (x != y) {    /* 3 */
      if (x > y)        /* 4 */
        x = x-y;        /* 5 */
      else y = y-x;     /* 6 */
    }                   /* 7 */
    gcd = x;            /* 8 */
}
```

The following are three paths:

```
P1: 1-2-3-8
P2: 1-2-3-4-5-7-3-8
P3: 1-2-3-4-6-7-3-8
```

The first path can be represented as follows:

```
    x = m;
    y = n;
  @ !(x != y);
    gcd = x;
```

Here `@` denotes a condition (e.g., loop condition or assertion); and '`!`' means the negation of a condition.

The path conditions for the three paths are: (1) $m = n$; (2) $m = 2n$; (3) $n = 2m$, respectively. Obviously, given any pair of integers satisfying the equation $m = n$ as the initial input data, the program will run along path `P1`. And, if we provide any pair of positive integers $m, n$ satisfying the second equation as the input to the program, the program will execute along path `P2`.

How can we obtain the path condition, given a path in the program? This can be done via *symbolic execution* [5]. In contrast to concrete execution, symbolic execution takes symbolic values as input, e.g., $m = m_0$, $n = n_0$; and then simulates the behavior of the program. During the process, each variable can take an expression as its value, e.g., $x = m_0 + 2n_0$.

Combined with constraint solving, symbolic execution can be very helpful in static analysis (bug finding) and test case generation [10]. For such purposes, usually we need to find just one solution to each path condition. That solution may correspond to a test case, or the input vector which shows a bug in the program.

More generally, we may consider the number of solutions to a path condition. In [9], we proposed a measure $\delta(P)$ for a path $P$ in a program. Informally, it denotes the number of input vectors (when the input domain is discrete) which drive the program to be executed along path $P$. In other words, it denotes the volume of the subspace in the input space of the program that corresponds to the execution of path $P$, or equivalently, the size of the solution space of the path condition.

For the GCD example, suppose that each of the input variables `m` and `n` can take values from the domain [1..100]. So there are 10,000 different input vectors (i.e., pairs $\langle m, n \rangle$). It is easy to see that

$$\delta(P1) = 100,\ \delta(P2) = 50,\ \delta(P3) = 50.$$

In [8], we studied how to compute the size of the solution space, given a set of constraints (where each constraint is formed from Boolean variables and linear inequalities using logical operators like negation, disjunction, conjunction). We developed a prototype tool for doing this [11]. For example, suppose the input formula is $(x \neq 0) \vee b$. Here $x$ is an integer variable and $b$ is a Boolean variable. If we use the command-line option `-v3` which specifies the word length of integer variables to be 3, the result is 15. The reason is that, when $b$ is TRUE, any value of $x$ can satisfy the formula; and when $b$ is FALSE, any non-zero value of $x$ satisfies the formula. So, there are $8 + 7 = 15$ solutions in total.

In [8], we also introduced a measurement called *execution probability.* Given a path $P$, its execution probability is $\delta(P)$ divided by the total volume of the involved data space. For path P1 in the GCD example, its execution probability is 0.01; For each of the other two paths, the execution probability is 0.005.

In [7], we proposed to move from qualitative analysis to quantitative analysis, which combines symbolic execution with volume computation to compute the exact execution frequency of program paths and branches. We argue that it is worthwhile to generate certain test cases which are related to the execution frequency, e.g., those covering cold paths.

Recently, Geldenhuys et al. [3] also discussed how the calculation of path probabilities may help program analysis (esp. bug finding). They implemented a prototype tool which can analyze Java programs.

The above works focus on finding bugs in programs, while the main purpose of the approach presented here is to estimate the performance of programs.

## 3 Performance Estimation

In practice, the performance of a program/algorithm is often measured by the running times on some machine, with respect to a set of benchmarks. This is useful. But the result often depends on the choice of machines and benchmarks.

Rather than using running times, we may also use some performance indicators or performance indices (PINDs in short) which can describe a program's performance on a family of machines. For instance, we may choose the number of memory accesses as a PIND, or the number of multiplication operations as a PIND. When studying sorting algorithms, we may choose the number of comparisons, or the number of swaps (i.e., element exchanges). Actually, Don Knuth [6] proposed to use the number of memory operations (*mems*) rather than execution times on some particular machine, when reporting the performance of some algorithm/program.

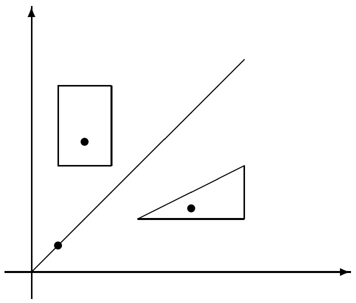
Given a program, we can estimate its performance by generating a set of program paths, denoted by $\{P_i\}$. For each path $P_i$, usually it is not difficult to determine the PIND value $pind_i$. The performance of the whole program can be computed by taking an average over the set of paths. The contribution of path $P_i$ is weighted by its $\delta$ value $\delta_i$, or its execution probability. More formally, the program's performance is estimated to be $\sum_i(\delta_i * pind_i)/\sum_i \delta_i$.

Of course, we can also extend the old-fashioned way. We may execute the program for a number of times (with different input data), remembering its actual performance (e.g., $time_1 = 5$ ms, $time_2 = 8$ ms), as with ordinary empirical evaluation. And then, for each concrete execution, we

- extract the path;
- get the path condition;
- compute the delta value of the path.

Finally, we get an estimation of the program's performance using the formula $\sum_i(\delta_i * time_i)/\sum_i \delta_i$.

Let us illustrate the approach/methodology using the following picture:



Here we assume that the input space is 2-dimensional (i.e., there are two input variables for the program, as in the case of GCD).

The above picture compares traditional benchmarking (examining the program's behavior at three points) with our approach (examining the program's behavior with respect to three sets of input data: one line, one triangle, and one rectangle). If some set is too small (as compared with others), its contribution can almost be neglected.

## 4 Examples

### 4.1 Sorting

Sorting algorithms are very important in computer science, and they have been studied extensively. For many basic sorting algorithms, we can choose the number of comparisons (or the number of swaps) as a PIND.

Let us consider the following bubble sort algorithm.

```
for(i = 0; i < N-1; i++)
  for(j = N-1; j > i; j--) {
    if (a[j-1] > a[j])
      swap(a[j-1], a[j]);
  }
```

We assume that the input array `a` is of size 4. Then there are 24 *feasible* paths. For each path $P_i$, we can get the path condition and the value of $\delta(P_i)$.

Assume that the value of each array element is between 0 and 10 (i.e., $0 \leq a[i] < 10$). Using symbolic execution and our volume computation tool [8,11], we can find that the $\delta$ value of each path is almost the same (a little more than 416.66). In this sense, the example is a bit special.

For each path, the number of comparisons is the same, i.e., 6. However, the number of swaps is different, ranging from 0 to 6. The total number of swaps for all the 24 paths is 72. So, on average, the number of swaps performed by the bubble sort algorithm is 3 (when $N = 4$). In summary, when $N = 4$, the average performance of the above program is like the following: 3 swaps of array elements and 6 comparisons between array elements.

Note that the domain [0..10) is arbitrary. We can replace 10 by 100, for instance.

### 4.2 Selection

Now we consider the selection algorithm FIND [4]. It can be described in C-like language, as follows:

```
m = 0;
n = N-1;
while (m < n) {
  r = A[f];
  i = m; j = n;
  while (i <= j) {
    while (A[i] < r)
```

```
        i = i+1;
      while (r < A[j])
        j = j-1;
      if (i<=j) {
         w = A[i]; A[i] = A[j]; A[j] = w;
         i = i+1;
         j = j-1;
      }
    }
    if (f<=j) n = j;
    else if (i<=f) m = i;
    else {
      m = 1; n = 0;
    }
  }
```

Given an array of size $N$ (denoted by `A[N]`) and a non-negative integer $f$, the algorithm tries to find the element whose value is the $f$'th in the array; and rearranges the array such that this element is placed in `A[`$f$`]`. At the end of the algorithm, the elements of the array should satisfy the following relationship:

$$A[0], A[1], \ldots, A[f-1] \leq A[f] \leq A[f+1], \ldots, A[N-1].$$

From the above source code, we can extract a number of paths from its flow graph. The following are several *short* paths:

**Table 1.** Paths in the FIND Program

| pathID | nComp | nSwap | $\delta$ |
|---|---|---|---|
| w11 | 9 | 3 | 16303680 |
| w16 | 9 | 2 | 12191040 |
| w18 | 9 | 3 | 16303680 |
| w20 | 9 | 2 | 12191040 |

Here `nComp` means the number of comparisons of array elements; and `nSwap` means the number of swaps of array elements. We assume that $f = 3$, and the size of the array is 8, i.e., $N = 8$. We also assume that each element of the array A is an integer, and it takes its value from a finite domain: $-8 \leq A[i] < 8$.

We note that the size of the whole input space is $16^8 = 2^{32}$. And the $\delta$ values in the above table are still very small, as compared with this size. But we believe that it is better to use these subspaces than a small set of single points (as with traditional empirical evaluation).

It is not true that all the paths are similar. For instance, we randomly generated the following two input vectors:

```
A[8] = { -2, 5, 6, 3, 1, 0, -7, 6 };
A[8] = { 2, 0, -2, -8, 4, -4, 5, 1 };
```

For the former, the execution path has 4 swap operations; but for the latter, the execution path has 6 swap operations.

We may execute the program for 2 times, with the above data as input. Then we track the execution traces and obtain two program paths. From each path, we can get the path condition using symbolic execution.

The path condition for the first case 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]).
```

And the $\delta$ value is 4075920. On the other hand, the $\delta$ value for the second case is just 87516. The corresponding path condition is the following set of constraints.

```
!(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]);
```

If we take an average over the paths using the traditional method, the number of swaps will be $(4 + 6) = 5$. But if we take a weighted average, the number of swaps will be $(4075920 * 4 + 87516 * 6)/(4075920 + 87516)$, which is about 4.04.

## 5 Discussion on the Limitations

One obvious difficulty with the approach is the combinatorial explosion problem. In general, a program has many (or an infinite number of) execution paths. For most programs, it is impossible to examine all the paths. We can just randomly choose a small subset of them, or choose as many as we can. Thus the result is only an *estimation* of the overall performance of the program.

Currently symbolic execution and volume computation are still expensive. But we think that the approach is practical when applied to kernel algorithms (which are not too long in terms of lines of code) or models of software systems.

One assumption that we take is that all the points in the input data space are evenly distributed. This may not be the case in certain applications. In the future, we will consider other distributions.

Sorting and selection algorithms are special in that the major operations are comparison and object movements. For a sorting algorithm like bubble sort, its behavior can often be determined by considering all possible permutations of the input vector. When the size of the input array is a small fixed positive integer, we can exhaustively examine all the paths of the algorithm, and then obtain the accurate average PIND value. But in most cases, it is not easy (if not impossible) to do this.

## 6 Concluding Remarks

In this paper, we propose to calculate or estimate the average performance of algorithms (programs) automatically, by analyzing their feasible paths and using volume computation techniques. We report some experiments with several common algorithms.

It should be emphasized that the approach is general, in that it is not restricted to a particular algorithm (or some class of algorithms). The approach can be automated. In fact, there are already some automatic tools available. As compared with traditional benchmarking, the approach can be more complete (i.e., covering a larger part of the input space).

As mentioned in the previous section, there are still some challenges for the new approach. But we believe it offers a better way to estimate the performance of programs. In the future, we will tackle the challenges, so that the approach is more practical and applicable to many algorithms.

**Acknowledgements.** The author would like to thank Peng Zhang for his comments on an earlier version of this paper.

## References

1. Boyer, R., Moore, S.: A fast string matching algorithm. Comm. ACM 20, 762–772 (1977)
2. Cole, R.: Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In: Proc. of the 2nd Symp. on Discrete Algorithms (SODA 1991), pp. 224–233 (1991)
3. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proc. of the Int'l Symposium on Software Testing and Analysis (ISSTA 2012), pp. 166–176 (2012)
4. Hoare, C.A.R.: Proof of a program: FIND. Commun. ACM 14(1), 39–45 (1971)
5. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
6. Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press (1994), `http://www-cs-faculty.stanford.edu/~knuth/sgb.html`
7. Liu, S., Zhang, J.: Program analysis: from qualitative analysis to quantitative analysis. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE 2011), pp. 956–959 (2011)
8. Ma, F., Liu, S., Zhang, J.: Volume computation for Boolean combination of linear arithmetic constraints. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 453–468. Springer, Heidelberg (2009)
9. Zhang, J.: Quantitative analysis of symbolic execution. Presented at the 28th Int'l Computer Software and Applications Conf. (COMPSAC 2004) (2004)
10. Zhang, J.: Constraint solving and symbolic execution. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 539–544. Springer, Heidelberg (2008)
11. Zhang, J., Liu, S., Ma, F.: A tool for computing the volume of the solution space of SMT(LAC) constraints, draft (January 2013)