Generating Failing Test Suites for Quantum Programs with Search (Hot Off the Press track at GECCO 2022)

Xinyi Wang
Simula Research Laboratory
Oslo, Norway
xinyi@simula.no

Paolo Arcaini
National Institute of Informatics
Tokyo, Japan
arcaini@nii.ac.jp

Tao Yue
Nanjing University of Aeronautics and Astronautics,
Simula Research Laboratory
Nanjing, China
taoyue@ieee.org

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
shaukat@simula.no

ABSTRACT

The inherent complexity of quantum programs, due to features such as superposition and entanglement, makes their testing particularly challenging. To tackle these challenges, we present a search-based approach, called Quantum Search-Based Testing (QuSBT), for automatically generating test suites of a given size that possibly expose failures of the quantum program under test. QuSBT encodes a test suite as a search individual, and tries to maximize the objective function that counts the number of failing tests in the test suite. Due to non-deterministic nature of quantum programs, the approach repeats the execution of each test multiple times, and uses suitable statistical tests to assess if a test passes or fails. QuSBT employs a genetic algorithm to perform the search. Experiments on 30 faulty quantum programs show that QuSBT is statistically better than random search, and is able to efficiently generate maximal failing test suites.
This is an extended abstract of the paper [1]: X. Wang, P. Arcaini, T. Yue, and S. Ali "Generating Failing Test Suites for Quantum Programs With Search", 13th International Symposium on Search-Based Software Engineering (SSBSE 2021).

CCS CONCEPTS

• **Theory of computation → Quantum computation theory**; • **Software and its engineering → Search-based software engineering**.

KEYWORDS

Quantum computing, genetic algorithms, search-based testing

ACM Reference Format:
Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Generating Failing Test Suites for Quantum Programs with Search (Hot Off the Press track at GECCO 2022). In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3520304.3534067

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3534067

1 INTRODUCTION

Quantum computing (QC) promises to solve profoundly complex computational problems, and different research and industrial efforts are being spent on it. QC frameworks and programming languages such as Qiskit are also available to develop quantum programs. As for classical programs, also for quantum programs assessing their correctness is highly required. However, testing quantum programs is not straightforward because of their unique features such as superposition and entanglement. As for classical programs, the search space of quantum programs can be particularly complex, and finding the input that triggers a failure can be challenging. Search-based approaches seem to be promising to this aim, and so, in [1], we proposed a search-based test generator for quantum programs. We here summarize it, by giving a minimal background in Sect. 2, presenting the approach in Sect. 3, and reporting some results in Sect. 4.

2 BACKGROUND

Quantum bits (*qubits*) are the basic units of quantum programs and, as classical bits, take value 0 or 1. However, a state of a qubit is also defined with its *amplitude* (α), a complex number given by a *magnitude* and a *phase*. The magnitude denotes the probability of a quantum program being in a particular state, while the phase is the angle of this complex number in polar form ranging from 0 to 2π radians. Given a quantum program QP, we identify with I the input qubits, and with O the output qubits; $D_I = \mathcal{B}^{|I|}$ are the input values, and $D_O = \mathcal{B}^{|O|}$ the output values. A quantum program QP can then be defined as a function $\mathrm{QP}\colon D_I \to 2^{D_O}$. We notice that, given the same input value, QP can return different output values that occur by following a certain probability distribution. If available, the program specification PS specifies the expected probabilities of occurrence of the output values, and can be used as oracle when testing QP. A quantum program can be specified as a *circuit* composed of different *gates* that manipulate the input values. Fig 1 shows the circuit of a simple program encoded in Qiskit. In an equivalent way, a circuit can also be specified in Python.

3 QUANTUM SEARCH-BASED TESTING

Quantum Search-Based Testing (QuSBT) is a search-based test generator for quantum programs. Given a given program QP, QuSBT generates a test suite composed of M tests, where M is a parameter

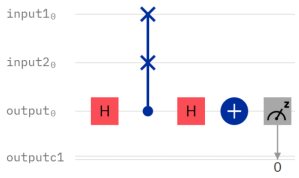


Figure 1: Circuit diagram of "swap test" program

of the approach. QuSBT is based on a genetic algorithm (GA) where the search variables are integers $\overline{x} = [x_1, \ldots, x_M]$. Each variable represents a test input for QP taken from D_I.

QuSBT searches for an assignment $\overline{v} = [v_1, \ldots, v_M]$ that maximizes the number of tests that *fail*. Namely, the fitness computation is as follows. For each test assignment v_j of the jth test:

- it identifies the number of times n_j that v_j must be repeated; indeed, due to the non-deterministic nature of quantum programs, an input must be executed multiple times to characterize its output distribution. In [1], we devised an approach to select a n_j which is suitable, i.e., that allows to have a faithful representation of the real output distribution;
- QP is executed n_j times with the input v_j, obtaining the result $res = [o_1, \ldots, o_n]$;
- two tests are used to assess if the result *res* follows the expected distribution ($\texttt{fail}_j$ is the assessment result):
  - the first test checks if *res* contains some output value that is not expected by the program specification; if it is so, $\texttt{fail}_j$ is set to *true* and the assessment terminates;
  - the Pearson's chi-square test checks if *res* follows the expected output distribution; if the null hypothesis is not rejected, $\texttt{fail}_j$ is set *false*, otherwise to *true*.

Given the assessments $ta = [\texttt{fail}_1, \ldots, \texttt{fail}_M]$ of all the tests, the fitness function that must be maximized is as follows:

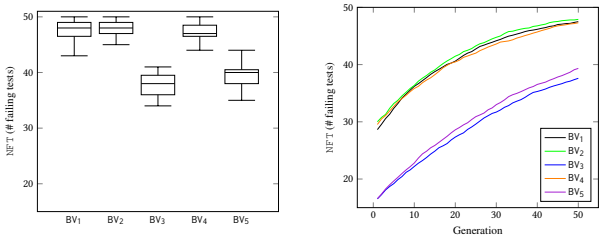
$$f(\overline{v}) = |\{\texttt{fail}_j \in ta \mid \texttt{fail}_i = true\}|$$

i.e., the number of failing tests of the test suite must be maximized.

4 EXPERIMENTS

We experimented the approach using six programs: (i) *Bernstein-Vazirani* (BV) and *Simon* (SM) are two cryptography programs; (ii) *QRAM* (QR) is a program to access and manipulate quantum random access memory (iii) *invQFT* (IQ) implements inverse quantum Fourier transform; (iv) *Add Squared* (AS) performs mathematical operations in superposition; (v) *Conditional Execution* (CE) performs conditional execution in superposition. BV, IQ, AS, CE have 10 input qubits, QR has 9, while SM has 7. The number of gates ranges from 60 to 15. *Circuit depth* goes from 3 to 56. For each of the six programs, we produced five faulty versions, by introducing different types of faults at various locations of the circuit. These 30 faulty programs are our *benchmarks* used in our experiments.

We used Qiskit 0.23.2 to specify the quantum programs. Qiskit also provides a simulator that we used to execute the programs. We implemented a version of the approach based on the GA version provided by jMetalPy 1.5.5. We set the population size as 10, and the termination condition as the maximum number of generations (i.e., 50). We also implemented a version of the search based on Random Search (RS). RS was given the same number of fitness evaluations



(a) Final # of failing tests NFT **(b) Growth of # of failing tests NFT**

Figure 2: Results of BV benchmarks

as GA, i.e., 500. The size of the test suite depends on the size of the input domain; in the experiments, it is the 5% of the size of the input domain D_I: so, we need to generate 50 tests for BV, IQ, AS, CE, 26 for QR, and 7 for SM. To account for the non-determinism of the search, we executed each experiment 30 times. Code and results are available at https://github.com/Simula-COMPLEX/qusbt/.

4.1 Results

We analyze the experimental results using two research questions.

RQ1: Does GA-based QuSBT outperform Random Search (RS)? We used the Mann-Whitney U test and the $\hat{A}_{12}$ statistics to compare the *number of failing tests* NFT found by the two approaches. We observed that for 26 out of 30 benchmarks, the GA-based implementation of the approach is statistically significantly better than the RS-based implementation, showing that the search problem is complex and deserves an advanced search algorithm.

RQ2: How is QuSBT's performance on the six benchmark programs? We observed that, for the most complex benchmarks for which we had to generate test suites of 50 tests, there is a high variability in terms of number of failing tests NFT across the 30 runs, and the maximum number of 50 failing tests was found seldomly: it could be that there are not so many failing tests, or the search was not given enough time. Fig. 2a shows the final results of BV benchmarks. For the simple program SM, instead, the search was able to find the maximum number of failing tests.

In terms of *convergence* of the search, we noticed that, for all the benchmarks, the first generation already generates some failing inputs. The speed of convergence, however, depends on the complexity of the program and on the number of failing inputs that actually exist. Fig. 2b shows the convergence of BV benchmarks.

ACKNOWLEDGMENTS

This work is supported by Qu-Test (Project#299827) funded by Research Council of Norway and the National Natural Science Foundation of China (No. 61872182). Paolo Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

REFERENCES

[1] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Generating Failing Test Suites for Quantum Programs With Search. In *Search-Based Software Engineering*. Springer International Publishing, Cham, 9–25. https://doi.org/10.1007/978-3-030-88106-1_2