Generating Failing Test Suites for Quantum Programs with Search*

Xinyi Wang[1] [0000−0001−5621−6140], Paolo Arcaini[2] [0000−0002−6253−4062], Tao Yue[1,3] [0000−0003−3262−5577], and Shaukat Ali[3] [0000−0002−9979−3519]

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China
[2] National Institute of Informatics, Tokyo, Japan
[3] Simula Research Laboratory, Fornebu, Norway

Abstract. Testing quantum programs requires systematic, automated, and intelligent methods due to their inherent complexity, such as their superposition and entanglement. To this end, we present a search-based approach, called Quantum Search-Based Testing (`QuSBT`), for automatically generating test suites of a given size depending on available testing budget, with the aim of maximizing the number of failing test cases in the test suite. `QuSBT` consists of definitions of the problem encoding, failure types, test assessment with statistical tests, fitness function, and test case generation with a Genetic Algorithm (GA). To empirically evaluate `QuSBT`, we compared it with Random Search (RS) by testing six quantum programs. We assessed the effectiveness of `QuSBT` and RS with 30 carefully designed faulty versions of the six quantum programs. Results show that `QuSBT` provides a viable solution for testing quantum programs, and achieved a significant improvement over RS in 87% of the faulty programs, and no significant difference in the rest of 13% of the faulty programs.

Keywords: Quantum Programs · Software Testing · Genetic Algorithms

1 Introduction

Testing quantum programs is essential for developing correct and reliable quantum applications. However, testing quantum programs is challenging due to their unique characteristics, as compared to classical programs, such as superposition and entanglement [11]. Thus, there is a need for the development of systematic, automated, and intelligent methods to find failures in quantum programs [11]. Such testing works have started to emerge, focusing on, e.g., coverage criteria [2], property-based testing [7], fuzz testing [12], and runtime assertions (e.g., *Proq* [9]). In contrast to existing works, we propose an approach to automatically generate test suites of given sizes with search algorithms, which are dependent on available testing budgets, with the aim of maximizing the number of failing test cases in a test suite. We call our approach *Quantum Search-Based Testing* (`QuSBT`), where the generation of test suites is encoded as a

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

search problem. We also identify two types of failures, devise statistical test-based test assessment criteria, and define the number of repetitions that are considered sufficient for test executions, by considering the inherent uncertainty of quantum programs.

`QuSBT` employs a Genetic Algorithm (GA) as its search strategy. To assess the cost-effectiveness of `QuSBT`, we compared it with Random Search (RS), the comparison baseline. We selected six quantum programs as the subject systems of the evaluation, and created 30 faulty versions of the programs (i.e., 30 benchmarks) to assess the cost-effectiveness of GA and RS in terms of finding test suites of given sizes and maximizing the number of failing tests. Results show that `QuSBT` performed significantly better than RS for testing 87% of the faulty programs, and there were no significant differences for the rest of 13% of the faulty programs.

Paper structure. Sect. 2 reviews the related work. Sect. 3 presents the background. Sect. 4 introduces definitions necessary for formalizing the `QuSBT` approach, which is discussed in detail in Sect. 5. Sects. 6-7 present our empirical evaluation. Then, Sect. 8 identifies threats that may affect the validity of `QuSBT`, and Sect. 9 concludes the paper.

2 Related Work

Ali et al. [2] proposed *Quito* – consisting of three coverage criteria defined on inputs and outputs of quantum programs, and two types of test oracles. Passing and failing of test suites were judged with one-sample Wilcoxon signed rank test, and mutation analysis was used to assess the effectiveness of coverage criteria. Results indicate that the least expensive coverage criterion, i.e., the input coverage, can manage to achieve high mutation scores, and the input-output coverage criterion (the most expensive one) could not increase mutation scores for most cases. As also acknowledged by the authors, the coverage criteria do not scale when handling quantum programs with more qubits, thus requiring the development of efficient quantum testing approaches such as `QuSBT`.

Huang and Martonosi [8] proposed a statistical assertion approach for finding bugs. They identified six bug types and their corresponding assertions. The chi-square test was used to test the hypothesis on the distributions of measurements, and determine a contingency coefficient with the confidence level of 95%. The approach tested three quantum programs followed by identifying bugs and their types in the programs.

Zhou and Byrd [10] proposed to enable runtime assertions, inspired by the quantum error correction, by introducing additional qubits to obtain information of qubits under test, without disrupting the execution of the program under test. The proposed approach was verified on a quantum simulator. Along the same lines, Li et. al. [9] proposed *Proq*, a projection-based runtime assertion scheme for testing and debugging quantum software. Proq only needs to check the satisfaction of a projection (i.e., a closed subspace of the state space) on a small number of projection measurements, instead of repeated program executions. Proq defines several assertion transformation techniques to ensure the feasibility of executing assertions on quantum computers. Proq was compared with other two assertion mechanisms [8,10] and it showed stronger expressive power, more flexible assertion location, fewer executions, and lower implementation overhead. When comparing with `QuSBT`, 1) Proq is a white-box, whereas `QuSBT` is black-box; 2) Proq requires the definition of projections and implements them as assertions, which

requires expertise and effort, while `QuSBT` does not need to change the quantum program under test to include assertions; thereby reducing cost; and 3) Same as `QuSBT`, Proq also requires repeatedly executing assertions for a *sufficiently* large number of times in order to achieve the confidence level of 95%.

QSharpCheck [7] tests Q# programs. The paper presents a test property specification language for defining the number of tests to generate, statistical confidence level, the number of measurements, and experiments for obtaining data to perform statistical tests. Moreover, QSharpCheck defines property-based test case generation, execution and analysis, and five types of assertions. It was evaluated with two quantum programs, via mutation analysis. In comparison, we focus on finding the maximum number of failing test cases in test suites with a GA based on two types of failures.

QuanFuzz [12] focuses on increasing the branch coverage with a GA. It outperformed a random generator in terms of the effectiveness of triggering sensitive branches, and achieved a higher branch coverage than the traditional test input generation method. We, instead, focus on finding failing test suites based on two types of test oracles, whereas QuanFuzz focuses on searching inputs to cover branches. Thus, the search problems of QuanFuzz are different.

3 Background

Quantum programs operate on *quantum bits* (*qubits*). Similarly as in classical computers, a qubit can take value 0 or 1. However, in addition, a state of a qubit is described with its *amplitude (α)*, which is a complex number and defines two elements: a *magnitude* and a *phase*. The magnitude indicates the probability of a quantum program being in a particular state, while the phase shows the angle of this complex number in polar form (it ranges from 0 to 2π radians). Taking a three-qubits quantum program as an example, we can represent all the possible states of the program in the Dirac notation:

$$\alpha_0 |000\rangle + \alpha_1 |001\rangle + \alpha_2 |010\rangle + \alpha_3 |011\rangle + \alpha_4 |100\rangle + \alpha_5 |101\rangle + \alpha_6 |110\rangle + \alpha_7 |111\rangle$$

$\alpha_0, \ldots, \alpha_7$ are the *amplitudes* associated to the eight program states. Note that each state is simply a permutation of the three qubits. The magnitude of a state, representing the probability of the program being in the state, is the square of the absolute value of its amplitude (α) (e.g., for $|100\rangle$, its magnitude is $|\alpha_4|^2$). Note that the sum of the magnitudes of all the states is equal to 1, i.e., $\sum_{i=0}^{7} |\alpha_i|^2 = 1$.

Fig. 1 shows a three-qubits program (*Swap Test* [6]) in the Qiskit framework [13] in Python. Its equivalent circuit is shown in Fig. 2, which also shows which line number of the code matches to which part of the circuit. It compares two qubits: *input1* and *input2*. If they are equal in terms of their states, then the value of the measure qubit (i.e., *output*) becomes 1 (as it is initialized as 0 by default) with the 100% probability; otherwise, the probability decreases when the two inputs are increasingly different. Lines 2 and 3 initialize the two input qubits (i.e., *input1* and *input2*) that are to be compared. Line 4 initializes one output qubit (*output*) which is the condition qubit for controlling the swap gate. Line 5 initializes a classical register (*outputc1*) that stores the result of the comparison. Finally, Line 6 creates the quantum circuit. After the initialization (Lines 2-6), the state of the program will be *000* with amplitude of 1 (i.e., probability of 100%).

```
1  # Initialization of the quantum program
2  input1 = QuantumRegister(1, name='input1')
3  input2 = QuantumRegister(1, name='input2')
4  output = QuantumRegister(1, name='output')
5  outputc1 = ClassicalRegister(1, name='outputc1')
6  qc = QuantumCircuit(input1, input2, output, outputc1)
7
8  # Implementation of Swap Test
9  qc.h(output)
10 qc.cswap(output, input1, input2)
11 qc.h(output)
12 qc.x(output)
13 qc.measure(output, outputc1)
```

Fig. 1: Swap Test – Qiskit Code

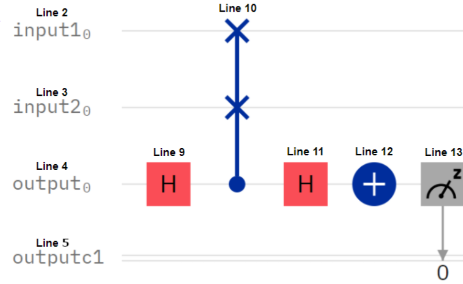


Fig. 2: Swap Test – Circuit diagram

In this execution of the program, we will compare *input1* initialized as 0 (by default) and *input2* initialized as 0 (by default). Line 9 applies the HAD gate [6] on *output* to put it in superposition. As a result, the state of the program will be 000 and 100 with amplitudes 0.707 (the 50% probability). Note that the *output* qubit is both 0 and 1 in this state of the program, whereas the other two qubits remain the same as initialized. Line 10 applies the CSWAP gate to swap the two input qubits (i.e., *input1* and *input2*). The swap only happens if the control qubit (i.e., *output*) has a value 1. Line 11 applies the second HAD gate on the *output* qubit. Due to the reversibility of the gates in quantum computing, another application of the HAD gate leads the program to its original state, i.e., *000*. Line 12 applies the NOT gate to the *output* qubit. As a result, the state in which the *output* qubit was 0 will become 1. Line 13 reads the *output* qubit and stores it in a classical register *outputc1*. The final state of the program is 1.

4 Definitions

Definition 1 (Inputs and outputs). Let Q be the set of qubits of the quantum program QP. A subset of qubits $I \subseteq Q$ defines the *input*, and a subset $O \subseteq Q$ identifies the *output*.[4] We define $D_I = \mathcal{B}^{|I|}$ as the input values, and $D_O = \mathcal{B}^{|O|}$ as the output values.

In the following, we will consider input and output values in their decimal representation. Note that input and output values are non-negative integers, i.e., $D_I = \{0, \ldots, 2^{|I|} - 1\}$, and $D_O = \{0, \ldots, 2^{|O|} - 1\}$.

Definition 2 (Quantum program). A quantum program QP can then be defined as a function QP: $D_I \rightarrow 2^{D_O}$.

Def. 2 shows that a quantum program, given the same input value, can return different output values. For each input, the possible output values occur by following a certain probability distribution. The program specification specifies the expected probabilities of occurrence of the output values.

[4] Note that I and O do not need to be disjoint, i.e., an input qubit can also be an output qubit. Moreover, there could also be qubits that are neither inputs nor outputs, i.e., $I \cup O \subseteq Q$.

Definition 3 (Program specification). Given a quantum program $\text{QP}\colon D_I \rightarrow 2^{D_O}$, we identify with $\texttt{PS}$ the *program specification*, i.e., the expected behavior of the program. For a given input assignment $i \in D_I$, the program specification states the expected probabilities of occurrence of all the output values $o \in D_O$, i.e.,:

$$\texttt{PS}(i) = [p_0, \ldots, p_{|D_O|-1}]$$

where p_h is the expected probability (with $0 \leq p_h \leq 1$) that, given the input value i, the value h is returned as output. It holds $\sum_{h=0}^{|D_O|-1} p_h = 1$. We introduce the following notation for selecting probabilities that are different from 0, i.e., those of the outputs that can occur according to the program specification:

$$\texttt{PS}_{NZ}(i) = [p \in \texttt{PS}(i) \mid p \neq 0] = [p_{j_1}, \ldots, p_{j_k}] \qquad \text{with } j_1, \ldots, j_k \in D_O$$

We further write $\texttt{PS}(i, h) = p_h$ to specify the expected probability of occurrence of output value h for input value i.

Note that, for some programs, the specifications of the expected outputs may not exist, and thus our approach would not be applicable.

5 Quantum Search-Based Testing (`QuSBT`)

We first give definitions of failure, test, and test assessment for quantum programs in Sect. 5.1, and then propose a test generation approach in Sect. 5.2.

5.1 Failures types, test, and test assessment

Any testing approach (also for classical programs) tries to trigger failures of the program under test, to reveal the presence of faults. Therefore, we need to define what a *failure* is in a quantum program. In this work, we target two types of failures:

- *Unexpected Output Failure* (*uof*): the program, for a given input i, returns an output o that is not expected according to the program specification $\texttt{PS}$, i.e., $\texttt{PS}(i, o) = 0$;
- *Wrong Output Distribution Failure* (*wodf*): the program, for multiple executions of a given input i, returns output values that follow a probability distribution significantly deviating from the one specified by the program specification.

We propose definitions of test and test assessment to reveal these types of failures. Moreover, the non-deterministic nature of quantum programs requires that a given input is executed multiple times. Therefore, we define a test input as follows.

Definition 4 (Test input). A *test input* is a pair $\langle i, n\rangle$, being i an assignment to qubits (i.e., $i \in D_I$), and n the number of times that program QP must be executed with i.

Definition 5 (Test execution and test result). Given a test input $\langle i, n\rangle$ for a quantum program QP, the *test execution* consists in running QP n times with input i. We identify with $\mathit{res}(\langle i, n\rangle, \text{QP}) = [\text{QP}(i), \ldots, \text{QP}(i)] = [o_1, \ldots, o_n]$ the *test result*, where o_j is the output value of the jth execution of the program.

Test assessment To check whether a test passes or fails, we need to check whether at least one of the two types of failures (i.e., *uof* or *wodf*) occurred.

To check *uof*, it is enough to check if some produced output is unexpected, i.e.,

$$\texttt{fail}_{\mathit{uof}} := (\exists o_j \in \mathit{res}(\langle i, n\rangle, \mathrm{QP})\colon \texttt{PS}(i, o_j) = 0)$$

If a failure of type *uof* is detected (i.e., $\texttt{fail}_{\mathit{uof}}$ is *true*), the assessment for *wodf* is not performed[5], because we can already assess that the test is not passing. Otherwise, it is performed as described in the following.

Checking *wodf* requires to check if the frequency distribution of the measured output values follows the expected distribution. We check this by doing a *goodness of fit* test with the Pearson's chi-square test [1]. The test checks whether the observed frequencies of the values of the categorical variable follow the expected distribution. In our setting, the categorical values are the possible output values of the quantum program QP for a given input i, i.e., those having their expected probabilities being non-zero, and the expected distribution is given by the program specification (i.e., $\texttt{PS}_{NZ}(i)$ in Def. 3). Concretely, given a test input $\langle i, n\rangle$ and its test result $\mathit{res}(\langle i, n\rangle, \mathrm{QP}) = [o_1, \ldots, o_n]$, we apply the chi-square test as follows:

- from the program specification, we retrieve the expected probabilities of the outputs that can occur given the input i, i.e., $\texttt{PS}_{NZ}(i) = [p_{j_1}, \ldots, p_{j_k}]$, with $j_1, \ldots, j_k \in D_O$ (see Def. 3); $j_1, \ldots, j_k$ are the categorical values of the test;
- then, from the test result, we collect, in $[c_{j_1}, \ldots, c_{j_k}]$, the number of occurrences of each possible output $j_1, \ldots, j_k$, where $c_{j_h} = |\{o \in \mathit{res}(\langle i, n\rangle, \mathrm{QP}) \mid o = j_h\}|$.[6] This is the *one-dimensional contingency table* of the chi-square test; and
- finally, we apply the chi-square test, that takes in input the contingency table $[c_{j_1}, \ldots, c_{j_k}]$ and the expected occurrence probabilities $[p_{j_1}, \ldots, p_{j_k}]$, and checks whether the recorded occurrences follow the expected probability distribution.

The null hypothesis is that there is no statistical significant difference. If the p-value is less than a given significance level α ($\alpha = 0.01$ in our experiments), we can reject the null hypothesis and claim that there is a statistical significant difference. In our case, this means that a *wrong output distribution failure wodf* occurred, which can be detected with the following predicate:

$$\texttt{fail}_{\mathit{wodf}} := (\text{p-value} < \alpha) \quad (1)$$

Note that the chi-square test requires to have at least two categories. Therefore, the assessment for *wodf* cannot be done when there is only one possible output for a given input (i.e., $|\texttt{PS}_{NZ}(i)| = 1$).[7] However, in this case, checking *uof* is enough. Indeed, if the program QP is not faulty for *uof*, it means that it always produces the unique output expected from the program specification and, so, also *wodf* is satisfied.

[5] In this case, we directly set $\texttt{fail}_{\mathit{wodf}}$ (see Eq. 1 later) to *false*.

[6] Note that the assessment for *wodf* is done only if the assessment for *uof* did not reveal any failure. If this is the case, it is guaranteed that the program returned outputs only from $j_1, \ldots, j_k$, i.e., those having their expected probabilities being non-zero. Therefore, it is guaranteed that each returned output is considered in one of the counts $c_{j_1}, \ldots, c_{j_k}$.

[7] Note that a quantum program can still be deterministic for some given inputs.

To conclude, the test is considered *failed* if one of the two failures is observed. So, we introduce a predicate recording the result of the test assessment as follows:

$$\texttt{fail} := \texttt{fail}_{\mathit{uof}} \vee \texttt{fail}_{\mathit{wodf}}$$

Remark 1. Note that the absence of a failure for an input i does not guarantee that the program behaves correctly for i. For *uof*, it could be that other additional executions would show a wrong output. For *wodf*, instead, the significance level of the test specifies the confidence on the absence of the fault. The argument is a little bit different for the case that the test fails. If $\texttt{fail}_{\mathit{uof}} = true$, we are sure that the program is faulty, because an output that should be never returned has been returned. Instead, if $\texttt{fail}_{\mathit{wodf}} = true$, it could still be that, with more executions, the observed frequencies of the output values would better align with the expected probability distribution specified in the program specification. In this case, the lower the p-value, the higher the confidence on the result.

Definition of the number of repetitions Defs. 4 and 5 say that a test must specify the number of times n that the input i must be executed for its assessment. It is well recognized that selecting such a number is difficult [9], and most approaches do not specify it nor provide a rationale for the selection of a particular number (e.g., [8]). Intuitively, the higher the number of repetitions, the better, as this gives a higher confidence on the result of the test assessment. However, a too high number of repetitions makes the assessment of tests infeasible, in particular when multiple tests must be assessed (as, for example, in a test generation approach as the one proposed in this paper).

In QuSBT, we select a number of repetitions that is sufficient to have a reasonable confidence on the result of the test assessment, but also makes the test assessment feasible to compute. We start observing that not all the inputs need the same number of repetitions: inputs for which a program specification specifies few possible output values, require fewer repetitions than those having a lot of possible outputs. Consider the case in which input *i1* has two possible outputs, while input *i2* has four possible outputs. Then, more repetitions are needed for input *i2* than for *i1*, as we need to provide comparable evidence for each of the possible outputs. On the basis of this intuition, we define a function that, given an input i, specifies the required number of repetitions:

$$\texttt{numRepetitions}(i) = |\texttt{PS}_{NZ}(i)| \times 100$$

So, the number of repetitions n of the test input i is proportional to the number of possible output values that, according to the program specification, can be obtained by executing the program with i (see Def. 3).

5.2 Test case generation

For a given program QP, QuSBT generates a test suite of M tests, being M an approach's parameter. It uses a GA, where search variables $\overline{x} = [x_1, \ldots, x_M]$ are integer variables, each one representing an input for QP taken from D_I. QuSBT finds an assignment $\overline{v} = [v_1, \ldots, v_M]$ for the M tests, such that as many of them as possible fail the program. Fitness is computed as follows. For each test assignment v_j of the jth test:

- We identify the required number of repetitions n_j, as described in Sect. 5.1;
- We execute QP n_j times with the input v_j, obtaining the result $res(\langle v_j, n_j \rangle, \text{QP})$;
- The result is assessed (see Sect. 5.1). We identify with $\texttt{fail}_j$ the assessment result.

Let $ta = [\texttt{fail}_1, \ldots, \texttt{fail}_M]$ be the assessments of all the M tests. The fitness function that we want to maximize is given by the number of failed tests, i.e.,

$$f(\overline{v}) = |\{\texttt{fail}_j \in ta \mid \texttt{fail}_i = \mathit{true}\}| \quad (2)$$

Selection of the number of tests `QuSBT` requires to select the number of tests M to be added in each searched test suite. Users can specify M, e.g., based on available budgets. However, selecting a value M without considering the program under test might not be a good practice. So, we propose to select M as the percentage β of the number of possible inputs D_I of the quantum program, i.e., $M = \lceil \beta \cdot |D_I| \rceil$. The user must then select the percentage β, rather than the absolute number M.

6 Experimental Design

We describe the experimental design to evaluate `QuSBT` in terms of research questions, benchmark programs, experimental settings, evaluation metrics, and statistical tests employed to answer the research questions. The used benchmarks and all the experimental results are available online[8].

Research Questions (RQs) We evaluate `QuSBT` using the following RQs:

- **RQ1**: Does `QuSBT` (which is GA-based) perform better than Random Search (RS)? RQ1 assesses whether GA can identify test inputs that contribute to failures, as compared to RS.
- **RQ2**: How does `QuSBT` perform on the benchmark programs? RQ2 assesses the variability on the final results, and how fast the GA converges to better solutions.

Benchmarks Programs We selected six programs with different characteristics (see Table 1): (i) cryptography programs: Bernstein-Vazirani (`BV`) and Simon (`SM`) algorithms; (ii) QRAM (`QR`) implements an algorithm to access and manipulate quantum random access memory, and invQFT (`IQ`) implements inverse quantum Fourier transform; (iii) mathematical operations in superposition, i.e., Add Squared (`AS`); (iv) conditional execution in superposition, i.e., Conditional Execution (`CE`).

Considering that there are no common known metrics available in the literature for characterizing quantum programs, we here propose to use the number of input qubits, the number of gates, and the circuit depth as the characterization metrics. The number of input qubits (i.e., $|I|$) intuitively characterizes the dimension of the input space (as in classical programs). Since we want to evaluate our approach with relatively complex programs, we selected four programs having 10 input qubits. Moreover, we selected two programs with a lower number of input qubits (i.e., `QR` and `SM`) to check whether the proposed approach is also advantageous on less complex programs.

The other two metrics, i.e., the number of gates and circuit depth, instead, to a certain extent, characterize the complexity of the program logic. The *number of gates* is

[8] `https://github.com/Simula-COMPLEX/qusbt/`

Table 1: Benchmark programs (Legend. AI: right after the inputs, MP: middle of the program, BR: right before reading the output)

Program	$\|I\|$	# gates	depth	Faulty versions (benchmarks)
AS	10	41	38	$\texttt{AS}_1$:AI, CNOT added; $\texttt{AS}_2$:AI, SWAP added; $\texttt{AS}_3$:BR, CNOT added; $\texttt{AS}_4$:BR, CSWAP added; $\texttt{AS}_5$:MP, CNOT added
BV	10	30	3	$\texttt{BV}_1$:AI, CNOT added; $\texttt{BV}_2$:AI, SWAP added; $\texttt{BV}_3$:AI, CCNOT added; $\texttt{BV}_4$:BR, CNOT added; $\texttt{BV}_5$:BR, CSWAP added
CE	10	25	20	$\texttt{CE}_1$:AI, CNOT added; $\texttt{CE}_2$:AI, SWAP added; $\texttt{CE}_3$:AI, CSWAP added; $\texttt{CE}_4$:BR, NOT added; $\texttt{CE}_5$:BR, HAD added
IQ	10	60	56	$\texttt{IQ}_1$:AI, CHAD added; $\texttt{IQ}_2$:MP, CHAD added; $\texttt{IQ}_3$:MP, replace H as CHAD; $\texttt{IQ}_4$:AI, CHAD added; $\texttt{IQ}_5$:MP, CHAD added
QR	9	15	12	$\texttt{QR}_1$:MP, CCPhase added; $\texttt{QR}_2$:MP, CHAD added; $\texttt{QR}_3$:MP, CNOT added; $\texttt{QR}_4$:AI, SWAP added; $\texttt{QR}_5$:BR, CSWAP added
SM	7	56	5	$\texttt{SM}_1$:AI, SWAP added; $\texttt{SM}_2$:AI, CCNOT added; $\texttt{SM}_3$:BR, CNOT added; $\texttt{SM}_4$:BR, CSWAP added; $\texttt{SM}_5$:BR, HAD added

the number of individual operators (e.g., HAD, NOT), while the *circuit depth* is given by the length of the longest sequence of quantum gates (the output of a gate used as the input of another gate determines a unit of the length). As shown in Table 1, IQ has the largest *number of gates*, BV has the least *circuit depth*, and AS has the longest *circuit depth*. We are aware that these metrics are coarse-grained, and in the future we plan to define and employ more fine-grained metrics.

For our selected programs, we derived the program specification (see Def. 3) to assess the passing or failing of tests. For each correct program, we produced five faulty versions of it by introducing different types of faults at different locations of the circuit. These 30 faulty programs are the *benchmarks* that we test in our experiments, which are described in details in Table 1. The benchmark name (e.g., $\texttt{AS}_1$) recalls the original correct program acting as the program specification (e.g., AS). The table also reports the location where a fault has been injected (i.e., right after the inputs, middle of the program, or right before reading the output). A short description is also provided for each benchmark program to tell which kind of gates were added. For instance, a CNOT gate is added right after the input to the original program to produce $\texttt{AS}_1$.

Experimental Settings We use Qiskit 0.23.2 [13] to write quantum programs in Python. It also provides a simulator for executing quantum programs, which we used for each evaluation of a given input i (i.e., $\mathrm{QP}(i)$).

We adopted GA from the jMetalPy 1.5.5 framework [5], and used the default settings of jMetalPy: the binary tournament selection of parents, the integer SBX crossover (the crossover rate = 0.9), the polynomial mutation operation being equal to the reciprocal of the number of variables. The population size is set as 10, and the termination condition is the maximum number of generations which is set as 50. As the baseline comparison, we also implemented a Random Search (RS) version of the approach from jMetalPy. RS has been given the same number of fitness evaluations as GA, i.e., 500. Note that there is no existing baseline with which we can compare QuSBT.

Search variables $\overline{x} = [x_1, \ldots, x_M]$ (see Sect. 5.2) of `QuSBT` represent the input values of the tests (i.e., the values i of tests; see Def. 4) of the searched test suite. So, the search interval of each variable is given by the set of possible input values D_I of the program; since inputs of a quantum program are non-negative integers (see Sect. 4), the search space is defined as $x_k \in [0, |D_I| - 1]$ for $k = 1, \ldots, M$.

`QuSBT` requires to select, as parameter, the number of tests M of each generated test suite. This can be selected as percentage β of the size of the input domain of the program (see Sect. 5.2). We here use β=5%; this results in having M=50 for programs with 10 qubits (and so 1024 input values), M=26 for the program with 9 qubits (and so 512 input values), and M=7 for the program with 7 qubits (and so 128 input values).[9]

For the fitness evaluation, assessing whether a test passes or fails requires to perform the Pearson Chi-square test for checking failures of type *wodf* (see Sect. 5.1). To this aim, we adopt *rpy2 3.4.2*, a Python interface to the R framework. We use $\alpha = 0.01$ as the significance level in the Chi-square test (see Eq. 1). Notice that correct inputs may still provide distributions slightly different from the expected ones (due to the limited number of repetitions); therefore, to be more confident on the failure of an input, we use the value 0.01 for the Chi-square test, instead of 0.05 or a higher confidence level.

Experiments have been executed on the Amazon Elastic Compute Cloud, using instances with a 2.9 GHz Intel Xeon CPU and 3.75GB of RAM. To consider the randomness in search algorithms, each experiment (i.e., the execution of `QuSBT` for a given benchmark using either GA or RS) has been executed 30 times, as suggested by guidelines to conduct experiments with randomized algorithms [3].

Evaluation Metrics and Statistical Tests In order to evaluate the quality of the results of the search algorithms (GA and RS), we directly use the fitness function defined in Eq. 2 as the *evaluation metric*, which counts the number of failing tests in a test suite (i.e., an individual of the search). We call it the *Number of Failed Tests* metric (`NFT`). The `NFT` of GA is given by the best individual of the last generation, while the `NFT` of RS is given by the best of all the generated individuals.

To answer RQ1, we selected the Mann–Whitney U test as the statistical test and the Vargha and Delaney's $\hat{A}_{12}$ statistics as effect size measure based on the guidelines [3]. Namely, given a benchmark program, we run the generation approach 30 times with GA and 30 times with RS. The Mann–Whitney U test (with the significance level of 0.05) is used to compare the 30 `NFT` values obtained by GA and the 30 `NFT` values of RS. The null hypothesis is that there is no statistical difference between GA and RS. If the null hypothesis is not rejected, then we consider GA and RS equivalent. Otherwise, if the null hypothesis is rejected, we apply the $\hat{A}_{12}$ statistics. If $\hat{A}_{12}$ is 0.5, then it means that the results are obtained by chance. If $\hat{A}_{12}$ is greater than 0.5, then GA has a higher chance to achieve a better performance than RS, and vice versa if $\hat{A}_{12}$ is less than 0.5.

[9] Note that we manually approximated the value of programs with 1024 inputs values. Indeed, the correct number of tests would be $\lceil 0.05 \cdot 1024 \rceil = 52$.

Table 2: Comparison between GA and RS (≡: there is no statistically significant difference between GA and RS. ✓: GA is statistically significantly better.)

AS_1	AS_2	AS_3	AS_4	AS_5	BV_1	BV_2	BV_3	BV_4	BV_5	CE_1	CE_2	CE_3	CE_4	CE_5	IQ_1	IQ_2	IQ_3	IQ_4	IQ_5	QR_1	QR_2	QR_3	QR_4	QR_5	SM_1	SM_2	SM_3	SM_4	SM_5
✓	✓	✓	≡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	≡	✓	✓	✓	✓	✓	✓	✓	✓	≡	≡	✓	✓	✓

7 Results and Discussions

7.1 Results and Analyses

RQ1 To assess the usefulness of using a search algorithm, in our case GA, we compared it with RS. For each experiment (i.e., the test generation for a benchmark program), we executed 30 runs with GA and 30 runs with RS. We selected the *Number of Failing Tests* NFT as the evaluation metric (see Sect. 6). Then, we compared 30 values of GA and 30 values of RS, with the Mann-Whitney U test and the $\hat{A}_{12}$ statistics as described in Sect. 6. Comparison results are summarized in Table 2.

We observe that in 26 out of 30 cases, GA is significantly better than RS. This shows that GA is able to identify failing inputs in individuals. By considering the different types of the benchmarks, (see Table 1), we notice the differences in results. For some programs such as BV, CE, and QR, GA consistently performed significantly better than RS. In other programs such as AS and IQ, instead, in one out of the five cases, there are no differences between GA and RS. Note that, even for a simple program such as SM (for which we need to generate only 7 tests), GA is still better in three out of the five cases. This means that the task of selecting qubit values leading to failures is a difficult task also for programs with small numbers of input qubits such as those of SM (with 7 input qubits), and this further motivates the need for a search-based approach.

RQ2 Fig. 3 reports, for all the benchmarks, the quality of the final results in all the 30 runs, in terms of the evaluation metric NFT, which counts the number of failing tests in the returned test suite (see Sect. 6). In almost all the cases of the four groups of the most complex benchmarks (i.e., Figs. 3a-3d) for which we built test suites of 50 tests, the variability of the final results across the runs is high. Moreover, in these four complex benchmarks, the search was almost always not able to find all 50 failing tests. Similar results can be found in QR (i.e., Fig. 3e), for which we built test suites of 26 tests, the search cannot always find 26 failing tests. This could be due to the fact that there are not so many failing tests, or the search was not given enough time.

These observations tell us that a dedicated and large-scale empirical study is needed to investigate whether such a large variability and inability to find, e.g., 50 or 26 failing tests, is due to the randomness of the search (which perhaps can be mitigated with a better fitness function), is specific to fault characteristics (such as their types and seeding locations (Table 1)) or characteristics of quantum programs under test such as their circuit depth and numbers of gates.

For the benchmark programs of SM (Fig. 3f), instead, the required test suite size is much smaller (i.e., 7). Among its five SM benchmarks, for two of them (i.e., SM_1 and SM_5), the search found, in all 30 runs, 7 failing inputs, showing that the task is relatively

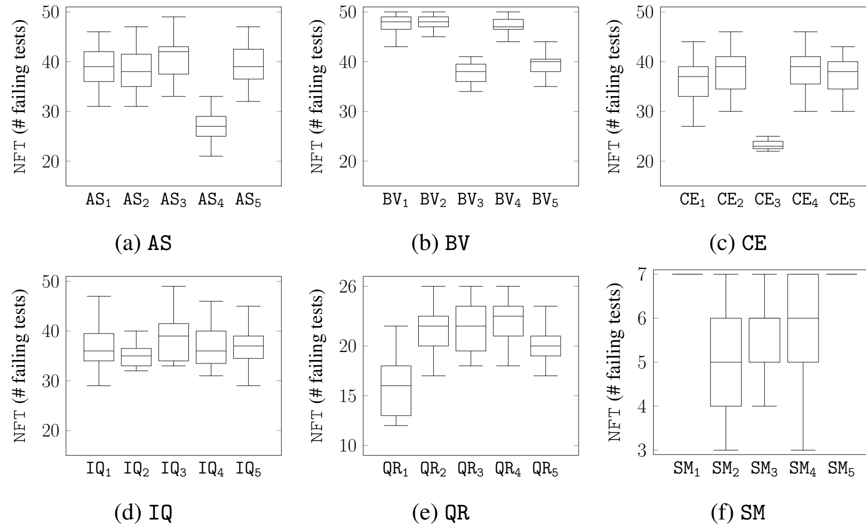


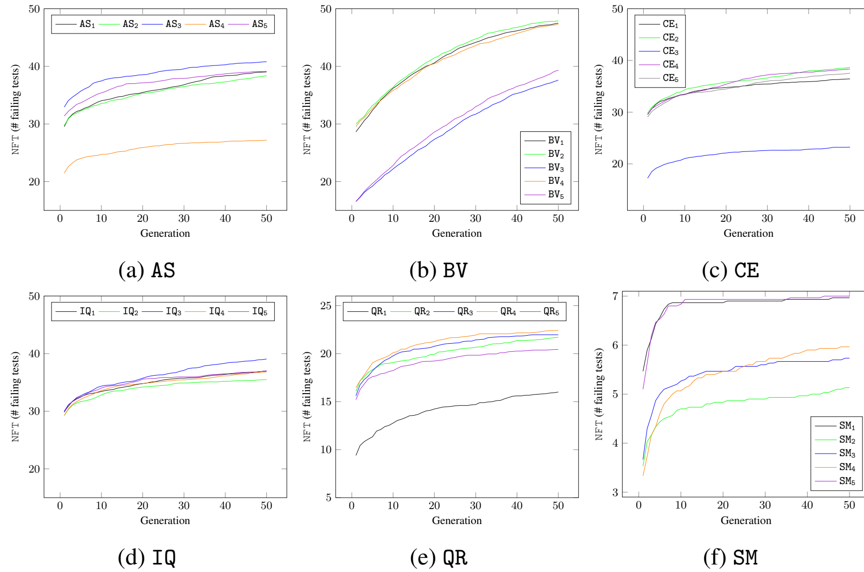
Fig. 3: Final results (# of failing tests in the final test suite) of GA across the 30 runs

easy. On the other hand, for the other three SM benchmarks, the search found less than 7 failing inputs (as low as 3 failing inputs).

We now want to assess how fast the test generation approach optimizes its objective (i.e., the maximization of the number of failing tests in a test suite). Figs. 4a-4f show, for each group of the benchmark programs, how the fitness (see Eq. 2) of the best individual in the population increases over generations. The reported plots are the averages across the 30 runs. First of all, we observe that, for all the benchmark programs, the first generation already finds some failing inputs. The number of discovered failing inputs in the first generation is positively correlated to the total number of failing inputs in the input space. Moreover, the number of identified failing inputs varies across the benchmark programs and depends on the types of faults and their locations in the benchmark programs (e.g., seeding a CSWAP gate right after the input or a HAD gate right before reading the output, see Table 1). Note that sometimes finding some failing inputs in a faulty circuit is not difficult, since RS can also do it. However, the maximization of the number of the failing tests is not trivial, as already evidenced by the observations reported for answering RQ1: GA is better than RS in finding more failing tests for most of the benchmark programs.

By observing the trends, we notice that they are increasing with different degrees of improvements. Three benchmarks of BV (i.e., $\mathtt{BV}_1$, $\mathtt{BV}_2$, and $\mathtt{BV}_4$, Fig. 4b) reach the point of almost discovering all the 50 failing tests in the final generation. $\mathtt{SM}_1$ and $\mathtt{SM}_5$ even reach a plateau after around 10 generations. Instead, all the other 25 benchmarks do not achieve high scores on detecting failing tests, possibly implying that further improvements would be still possible with additional generations.

For benchmarks of BV (see Fig. 4b), the increment in the fitness function is faster than the other benchmarks (those that must generate 50 tests). This does not necessarily



(a) AS (b) BV (c) CE

(d) IQ (e) QR (f) SM

Fig. 4: Results – Evolution of the fitness values over generations

mean that the problem is easy; indeed, for all the BV benchmarks, GA is better than RS (see Table 2). Instead, we believe that each fault of the BV benchmarks can be captured by a single well-defined pattern of input qubit values. So, once a pattern is discovered by the search, it is successfully migrated in new failing tests. We believe that, in other benchmarks, there is no such a single pattern of failing qubit values and, so, the search has more problems in finding new failing tests. As future work, we plan to perform an extensive investigation of how different types of in failing inputs, and how these failing inputs relate to each other (i.e., if they share some failing qubit values or not).

7.2 Discussion

The faults seeded in the quantum programs used for the evaluation can have different complexity (see Table 1). For instance, introducing a CNOT gate as a fault requires that the CNOT gate is applied to two qubits, which is intuitively considered more complex than introducing a HAD gate (operating on one qubit). However, seeding a fault to a *critical* location might significantly change the logic of the circuit. For instance, seeding a NOT gate *right before reading the output* might completely reverse the output of a circuit, which may make the observation of failures easy and then generate a program for which it is easy to generate failing tests. In Table 1, we classify the fault seeding locations into three categories: right after the inputs, middle of the program, and right before reading the output. This classification is coarse-grained, and a better mechanism is required to characterize fault seeding locations. So, we need larger-scale experiments, designed based on well understanding of faults characteristics, their relations to test inputs, and characteristics of quantum programs under tests. Consequently,

more comprehensive test strategies will be proposed in the future. Nevertheless, considering that quantum software testing is an emerging area, `QuSBT` contributes to building a body of knowledge in this area.

In this paper, we limit the scope of our study to identifying as many failing tests as possible. To assess passing and failing of a test, we defined two types of failures: *uof* and *wodf*, both of which do not consider the phase changes of qubits. Therefore, `QuSBT` currently can not reveal faults that only change the phases of output qubits, but not their occurrence probabilities.

The test input assessment requires to specify the number of repetitions (see Sect. 5.1). Note that, to the best of our knowledge, there is no existing criterion on how such a value should be set. So, our proposed mechanism provides a baseline for future research.

Even though our evaluation is performed on Qiskit in Python, `QuSBT` is general as it can be applied to other quantum platforms and quantum programming languages. In this paper, we performed all the experiments on the quantum computer simulator provided with Qiskit without simulating hardware faults. Thus, `QuSBT` needs to be extended in the future to deal with potential hardware faults in real quantum computers.

8 Threats to validity

External validity. We experimented only with six quantum programs and 30 (faulty) benchmark programs; thus, our results can be generalized to only quantum programs of similar characteristics. More experiments with varying characteristics of quantum programs are needed to generalize the results. Another threat is related to the selection of faults that were introduced in the quantum programs to create faulty benchmark programs. We chose a set of arbitrary faults, which could potentially affect our results. However, currently, there does not exist any bug repository for quantum programs that we could use to seed realistic faults in quantum programs.
Internal validity. We choose GA's default parameter settings. Different GA settings may produce different results that could potentially affect our results. However, the evidence has shown that even default settings of GA provide good results in search-based testing of classical software [4]. To assess the passing and failing of tests with *wodf*, we used the Pearson's Chi-square test. Other tests may be relevant; however, the Chi-square test has been used for this purpose in existing related literature [8,7].
Conclusion validity. Since GA and RS have inherent randomness, we repeated experiments 30 times for each faulty program to ensure that the results weren't obtained by chance. Followed by this, we compared the results of GA with RS with statistical tests according to the well-established guides in search-based software engineering [3].

9 Conclusion and Future Work

We presented a search-based approach for testing quantum programs that uses a Genetic Algorithm (GA) and employs a fitness function to search for test suites of a given size, containing as many failing tests as possible. We assessed the effectiveness of our approach as compared with Random Search (RS) with 30 faulty benchmark quantum

programs. The results showed that GA significantly outperformed RS for 87% of the faulty quantum programs, whereas for the rest, there were no significant differences.

Our future work includes experimenting with more algorithms and quantum programs and running them on quantum computers (e.g., by IBM). Moreover, we will perform analyses, e.g., studying the search space of solutions and the effect of search operators on the effectiveness of `QuSBT`. Finally, we will devise systematic methods to create realistic faulty quantum programs and publish a public repository.

References

1. Agresti, A.: An introduction to categorical data analysis. Wiley-Blackwell, 3 edn. (2019)
2. Ali, S., Arcaini, P., Wang, X., Yue, T.: Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 13–23 (2021). https://doi.org/10.1109/ICST49551.2021.00014
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 1–10. ICSE '11, ACM, New York, NY, USA (2011)
4. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering **18**, 594–623 (2013)
5. Benítez-Hidalgo, A., Nebro, A.J., García-Nieto, J., Oregi, I., Del Ser, J.: jMetalPy: A Python framework for multi-objective optimization with metaheuristics. Swarm and Evolutionary Computation **51**, 100598 (2019)
6. Gimeno-Segovia, M., Harrigan, N., Johnston, E.: Programming Quantum Computers: Essential Algorithms and Code Samples. O'Reilly Media, Incorporated (2019)
7. Honarvar, S., Mousavi, M.R., Nagarajan, R.: Property-based testing of quantum programs in Q#. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 430–435. ICSEW'20, Association for Computing Machinery, New York, NY, USA (2020)
8. Huang, Y., Martonosi, M.: QDB: From Quantum Algorithms Towards Correct Quantum Programs. In: 9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018). OpenAccess Series in Informatics (OASIcs), vol. 67, pp. 4:1–4:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019)
9. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime assertions for testing and debugging quantum programs. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020)
10. Liu, J., Byrd, G.T., Zhou, H.: Quantum circuits for dynamic runtime assertions in quantum computation. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1017–1030 (2020)
11. Miranskyy, A., Zhang, L.: On testing quantum programs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). pp. 57–60 (2019)
12. Wang, J., Ma, F., Jiang, Y.: Poster: Fuzz testing of quantum program. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 466–469 (2021). https://doi.org/10.1109/ICST49551.2021.00061
13. Wille, R., Van Meter, R., Naveh, Y.: IBM's Qiskit tool chain: Working with and developing for real quantum computers. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1234–1240 (2019)