Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search

Xinyi Wang*
Nanjing University of Aeronautics and Astronautics
Nanjing, China
wangxinyi125@nuaa.edu.cn

Tongxuan Yu*
Nanjing University of Aeronautics and Astronautics
Nanjing, China
yutongxuan@nuaa.edu.cn

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

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

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

ABSTRACT

Mutation testing is often used for designing new tests, and involves changing a program in minor ways, which results in mutated versions of the program, i.e., mutants. An effective test suite should find faults (or kill mutants) with a minimum number of test cases, to save resources required for executing test cases. In this paper, in the context of mutation testing for quantum programs, we present a multi-objective and search-based approach (`MutTG`) to generate the minimum number of test cases killing as many mutants as possible. `MutTG` tries to estimate the likelihood that a mutant is equivalent, and uses this as a *discount factor* in the fitness definition to avoid keeping on trying to kill mutants that cannot be killed. We employed NSGA-II as the multi-objective search algorithm. Then, we compared `MutTG` with another version of the approach that does not use the discount factor in its fitness definition, and with random search (`RS`), over a set of open-source quantum programs and their mutants of varying complexity. Results show that the discount factor does indeed help in guiding the test generation, as the approach with the discount factor performs better than the one without it.

CCS CONCEPTS

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

KEYWORDS

Quantum Programs, Genetic Algorithms, Search-Based Testing, Mutation Testing

*Both authors contributed equally to the paper.

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

ACM ISBN 978-1-4503-9237-2/22/07.
https://doi.org/10.1145/3512290.3528869

ACM Reference Format:
Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528869

1 INTRODUCTION

Quantum software implements a desired quantum application that can be executed on quantum computers. Same as for classical software, it is important to ensure the correctness of the quantum software such that it implements expected functionalities of the quantum application, and, therefore, testing is the straightforward choice to check the correctness of quantum software [21, 40, 41].

Several testing techniques for quantum programs, from different perspectives, have been proposed in the past few years. For example, a property-based testing framework for Microsoft Q# programs is published in [13]. A fuzz testing approach [34] has also been proposed. Moreover, coverage criteria on inputs and outputs of quantum programs in a classical way have also been defined and empirically evaluated [2, 36]. Recently, a search-based approach has been applied to generate test suites with the maximum number of failing tests for quantum programs with a given budget [35].

In classical software testing, mutation testing [15, 25] has been widely used to develop new test cases based on mutated versions of a program (i.e., *mutants*), e.g., [9, 10, 12, 26]. Mutation testing can be used, for example, for regression testing, when the original program is known to be correct, and test cases that effectively kill mutants are executed on future versions of the program to hopefully identify possible faults in them. Different studies have been conducted regarding the effectiveness of mutation testing. For example, it has been shown that mutants are good for improving the fault detection capability of test suites [28], and mutation testing has higher fault revelation than structural based coverage criteria such as statement and branch coverage [7].

In this paper, motivated by the success of mutation testing for classical software, we investigate its use for quantum programs. In particular, we investigate the problem of test generation. Namely, we propose a multi-objective search-based approach (`MutTG`) for generating a minimum number of test cases to kill a maximum

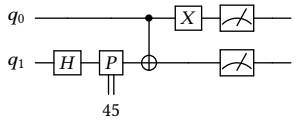

Figure 1: Increment Quantum Circuit

number of mutants of a quantum program. MutTG has two objectives. The first objective consists in minimizing the number of test cases. While this optimization is important for testing in general, it is even more critical for quantum software testing, since there is a limited access to execute test cases on publicly available real quantum computers (e.g., IBM's quantum computer). The second objective consists in minimizing the number of mutants that are not killed by the generated test suite.

A well-known problem in mutation testing is posed by *equivalent mutants* [19, 25], i.e., mutants that are equivalent to the original program and so cannot be killed. Since detecting equivalence is in general undecidable, very often it is not know which mutants are equivalent and so, during generation, resources are wasted to try to kill mutants that cannot be killed. To mitigate this problem, we introduce in the second objective a *discount factor*, which estimates the likelihood that a mutant is equivalent, such that the search can avoid keeping on trying to kill mutants that cannot be killed.

We employ NSGA-II as the multi-objective search algorithm. To evaluate our approach, we employed five open source quantum programs, based on which we further developed 20 benchmarks with three *difficulty* levels of killing mutants.

Experimental results indicate that: 1) NSGA-II significantly outperforms Random Search (the employed baseline) for all the difficult benchmarks composed of *subtle* mutants [25], i.e., those that are killed by a few inputs; 2) The *discount factor* in MutTG is effective in avoiding spending meaningless effort on keeping on trying for killing non-killable mutants; and 3) MutTG shows the potential for generating effective test cases for killing *subtle* mutants.

We organize the paper as follows. Sect. 2 introduces the relevant background. Sect. 3 presents our approach MutTG. We describe the design of our empirical evaluation in Sect. 4, and results and discussion in Sect. 5. Sect. 6 discusses threats that may affect the validity of the approach. Sect. 7 provides comparison with related work. Conclusions and future works are presented in Sect. 8.

2 BACKGROUND

2.1 Quantum Circuit Example

Currently, quantum computers are programmed at a low level as quantum circuits. A quantum circuit consists of a set of quantum gates (e.g., Hadamard). Table 1 provides the description of some quantum gates. Such gates are applied to quantum bits (or qubits in short). An example of quantum circuit is shown in Fig. 1. The example performs an increment in a two qubits program (i.e., q_0 and q_1). First, we apply the Hadamard gate to q_1 (H) followed by a phase shift of 45° with the P gate. Second, we implement an increment function that requires applying two gates, i.e., a conditional NOT gate (CNOT) followed by a NOT gate (X). Third, we measure the two qubits to read the incremented value.

Table 1: Definitions of some quantum gates

Name	Description
NOT (X)	It negates a qubit. It turns one qubit from state $\|1\rangle$ to state $\|0\rangle$, or state $\|0\rangle$ to state $\|1\rangle$.
C..C-NOT (C..CX)	It is a conditional gate applied on one or more control qubits and the target qubit. If the control qubits are all $\|1\rangle$, it performs NOT on the target qubit.
Hadamard (H)	It creates a superposition of $\|1\rangle$ and $\|0\rangle$ for one qubit allowing one qubit to have an equal chance of being in state $\|1\rangle$ or $\|0\rangle$.
C..C-Hadamard (C..CH)	It is a conditional gate, which applies the Hadamard gate on the target qubit if the conditional qubit(s) is(are) $\|1\rangle$.
Phase (P)	It rotates the relative phase of a qubit according with the angle (P).
C..C-Phase (C..CP)	It is a conditional gate. It rotates the relative phase on the target qubit with the angle (P), if conditional qubit(s) is(are) $\|1\rangle$.
CZ	It is a conditional gate applied on one control qubit and one target qubit. If the control qubit is $\|1\rangle$, the target qubit will rotate by 180°.
SWAP	It is a two-qubit gate, which swaps two qubits.
(CSWAP)	It is a conditional gate applied on one control qubit and two target qubits. If the control qubit is $\|1\rangle$, the two target qubits will be swapped.

2.2 Definitions

We provide the minimal definitions that are necessary for understanding the approach. We adopt the same theoretical framework that has been applied by other quantum testing works such as [2].

Definition 1 (Quantum program)**.** Let Q be the set of qubits of a quantum program QP. A subset of qubits $I \subseteq Q$ identifies the *input*, while a subset $O \subseteq Q$ identifies the *output*.[1] $D_I = \mathcal{B}^{|I|}$ are the *input values*, and $D_O = \mathcal{B}^{|O|}$ are the *output values*. A *quantum program* QP can be described as a function $\text{QP}: D_I \rightarrow 2^{D_O}$.

In our running example (Fig. 1), we have two qubits, both of which define inputs and outputs. Thus, we have four inputs and four outputs (from 0 to 3).

Def. 1 shows that, given an input value, a quantum program can return different output values. The program specification specifies the *expected* probability distribution followed by the output values.

Definition 2 (Program specification)**.** Given a quantum program $\text{QP}: D_I \rightarrow 2^{D_O}$, we identify with 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.,:

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

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 $\sum_{h=0}^{|D_O|-1} p_h = 1$. The probabilities of outputs that can actually occur are identified as follows:

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

For the example in Fig. 1, the program specification requires that for input 0, we must observe 1 or 3 with equal probability.

Note that the program specification can be either obtained from requirements, or can be derived from an existing program. The latter case is done, for example, in a regression testing setting in which the program specification is derived from the current correct program, and this is used for assessing future versions of the program. We consider this setting in this work.

[1] Note that I and O do not need to be disjoint.

Definition 3 (Test input)**.** A *test input* is a pair $\langle i, n \rangle$, where i is an assignment to qubits (i.e., $i \in D_I$), and n indicates how many times QP must be run with i.

A test needs to be executed multiple times to get an *estimate* of the probability distribution followed by the output values. Different inputs require different numbers of repetitions to get representative results; namely, the number of required repetitions should be proportional to the number of possible output values, as specified by the program specification. Given an input i, the number of required repetitions is defined as follows:

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

Definition 4 (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 $res = [\text{QP}(i), \ldots, \text{QP}(i)] = [o_1, \ldots, o_n]$ the *test result*, where o_j is the output value of the jth execution of the program.

The *failure* definition must consider the stochastic nature of quantum computing. We identify two types of failures:

- Unexpected Output Failure (*uof*): this occurs if the output o returned by the program for a given input i, is not allowed by the program specification PS, i.e., $\texttt{PS}(i, o) = 0$. To check *uof*, it is enough to check if some produced output is unexpected, i.e.,

$$\texttt{fail}_{uof} := (\exists o_j \in res\colon \texttt{PS}(i, o_j) = 0) \tag{1}$$

  For example, for our running example, for input 0, we must only observe 1 or 3. If we obtain 0 or 2, then there is an *uof*.
- Wrong Output Distribution Failure (*wodf*): This failure occurs when the output values returned by multiple executions for a given input i follow a probability distribution *significantly different* from the one specified by the program specification. This is assessed with a *goodness of fit* test using the Pearson's chi-square test [1]: the test compares the observed frequencies of the values of a categorical variable with the expected distribution. In our case, given a test input $\langle i, n \rangle$ and its test result $res = [o_1, \ldots, o_n]$, the chi-square test is applied using the expected probabilities $[p_{j_1}, \ldots, p_{j_k}]$ of the outputs that can occur given the input i, and the number of occurrences $[c_{j_1}, \ldots, c_{j_k}]$ of each possible output $j_1, \ldots, j_k$ in the test result. If the p-value is less than a given significance level α, we reject the null hypothesis that there is no statistical significant difference. The assessment is recorded with the following predicate:

$$\texttt{fail}_{wodf} := (\text{p-value} < \alpha) \tag{2}$$

  For example, for our running example, we run 200 times input 0, and observe 200 outputs (i.e., either 1 or 3). Then, we check with the statistical test if 1 and 3 are observed with equal probability. If the p-value is less than the chosen significance level, then there is a *wodf*.

3 PROPOSED APPROACH

In the following, we first introduce the notion of mutation analysis for quantum programs in Sect. 3.1. Then, we describe the proposed search-based test generation approach in Sect. 3.2.

3.1 Mutation Analysis for Quantum Programs

Given a quantum program QP, a mutant M is a slightly different version of QP that is obtained by introducing a syntactic fault in the circuit (e.g., using a wrong gate).

A test t is said to *kill* a mutant M if the execution of M over t produces significantly different results than the execution of t over the original program QP. In our case, we operate in a regression testing setting and derive the program specification PS from the original program QP. So, to check if a mutant is killed, we can compare it against PS. Namely, the mutant is killed if its execution leads to one of the two possible failures (see Eqs. 1 and 2):

$$\textit{isMutantKilled}(t, M) = \texttt{fail}_{uof} \vee \texttt{fail}_{wodf}$$

3.2 MutTG – Search-Based Generation

The goal of this work is to generate a test suite that is able to detect a set of mutants *Muts* of a quantum program QP. To this aim, we propose a search-based approach (called MutTG) that tries to generate the minimal test suite killing all the mutants.

In the following, we introduce the definition of individual of the search in Sect. 3.2.1. In Sect. 3.2.2, we describe how the tests identified by an individual are run and how mutation killing computed. We introduce the definitions of the objective functions in Sect. 3.2.3.

3.2.1 Individual Representation. In our approach, an individual represents a test suite for the quantum program QP (or for its future versions). In order to decide the number of variables, we start from the observation that killing $k = |Muts|$ mutants can require at most k tests (i.e., the worst case in which each test kills exactly one different mutant). Therefore, we define an individual of the search as k variables, each one representing a test. Formally:

$$\overline{x} = [x_1, \ldots, x_k]$$

where each variable x_i is defined in the domain $D = D_I \cup \textit{NoTestDom}$, with $\textit{NoTestDom} \cap D_I = \emptyset$ and $|\textit{NoTestDom}| = \beta \times |D_I|$.[2] The value v_i assumed by the variable x_i determines whether it represents a concrete test or not:

- if v_i is in D_I (i.e., the input domain of the quantum program. See Def. 1), it represents a concrete test;
- if v_i is in *NoTestDom*, it does not represent a test.

The size of *NoTestDom* determines the probability of not generating a test with a variable; in the approach, it can be customized as percentage β of the size of the input domain D_I.

Thanks to this representation, the search can generate test suites smaller than k (i.e., when some variables take values in *NoTestDom*).

3.2.2 Execution of Tests and Killing Assessment. We identify with $\overline{v}$ a concrete assignment to variables in $\overline{x}$. During the search, given an assignment $\overline{v}$, we execute the corresponding tests to check which mutants in *Muts* are killed.

The detailed description of the evaluation is given in Alg. 1. The algorithm, in addition to $\overline{v}$ and *Muts*, also takes in input *KM*, which is a dictionary that associates the tests generated so far in the search (i.e., for the previous individuals) with the mutants they kill. Moreover, the algorithm also takes in input the dictionary *NKM*, which associates the generated tests with the mutants that they

[2] Concretely, values in *NoTestDom* are the $\beta \times |D_I|$ integer values following the biggest value of D_I.

Algorithm 1 Execution of tests encoded in an individual

Require: $\overline{v}$: assignment to the individual
Require: *Muts*: mutants
Require: *KM*: dictionary of pairs $\langle t, \{M_1, \ldots, M_s\}\rangle$, where t is an executed test that kills mutants $\{M_1, \ldots, M_s\}$
Require: *NKM*: dictionary of pairs $\langle t, \{M_1, \ldots, M_r\}\rangle$, where t is an executed test that cannot kill mutants $\{M_1, \ldots, M_r\}$

1: $VT_{\overline{v}} \leftarrow \{v \in \overline{v} \mid v \in D_I\}$ ▷ Selection of valid tests
2: **for** $M \in Muts$ **do**
3: **for** $t \in VT_{\overline{v}}$ **do**
4: **if** $M \in KM[t]$ **then**
5: $killed \leftarrow true$ ▷ t is known to kill M
6: **else if** $M \in NKM[t]$ **then**
7: $killed \leftarrow false$ ▷ t is known to not kill M
8: **else**
9: $\texttt{numReps}(t) \leftarrow |\mathsf{PS}_{NZ}(t)| \times 100$
10: $res \leftarrow [M(t) \ldots, M(t)]$ ▷ test execution
11: $killed \leftarrow isMutantKilled(t, M)$ ▷ killing assessment
12: **if** *killed* **then**
13: $KM[t] \leftarrow KM[t] \cup \{M\}$
14: **else**
15: $NKM[t] \leftarrow NKM[t] \cup \{M\}$
16: **if** *killed* **then**
17: **break** ▷ other tests are not checked for M

do not kill. We write $KM[t]$ and $NKM[t]$ to retrieve the mutants associated with a test t.

The algorithm first selects from $\overline{v}$ the values that identify valid tests (line 1); recall from Sect. 3.2.1 that if a variable takes value in *NoTestDom*, it means that it does not represent a test. Then, for each mutant M (line 2), it iterates over the valid tests (line 3) and, for each test t, it performs the following actions:

- if t has already been evaluated in the past over mutant M and it killed it, the flag *killed* is set to *true* (lines 4-5);
- if t has already been evaluated in the past over mutant M and it did not kill it, the flag *killed* is set to *false* (lines 6-7);
- instead, if t has never been evaluated on M in the past, it is performed as follows:
  - the number of required repetitions $\texttt{numReps}(t)$ is calculated (line 9);
  - the mutant is executed $\texttt{numReps}(t)$ times over t, obtaining the test results *res* (line 10);
  - the results *res* are used to determined whether the mutant is killed or not as described in Sect. 3.1, and the assessment is stored in *killed* (line 11);
  - *KM* and *NKM* are updated accordingly (lines 12-15);
- if the current test t kills the current mutant M, the iteration over the tests is stopped for M, and the next mutant is evaluated (lines 16-17).

3.2.3 Search Objectives. The search is multi-objective. For each individual $\overline{v}$, the objective functions are evaluated after the execution of the tests and the assessment of the killability of mutants that have been described in Sect. 3.2.2 and Alg. 1.

The first objective is to minimize the test suite size. The corresponding fitness function is defined as follows:

$$f_{size}(\overline{v}) = |\{v_i \in \overline{v} \mid v_i \in D_I\}|$$

This fitness function measures the number of variables that are assigned with valid input values.

The second objective is to minimize the number of mutants that are *not* killed. The corresponding fitness function is defined as follows:

$$f_{not\_killed}(\overline{v}) = \sum_{M \in Muts} notKilledScore(M, \overline{v}, KM, NKM) \quad (3)$$

where *notKilledScore* is an index defined in $[0, 1]$ indicating the likelihood that the mutant M is non-killable. The index is defined as follows:

$$notKilledScore(M, \overline{v}, KM, NKM) = \begin{cases} 0 & \text{if } \exists v \in \overline{v}\colon M \in KM[v] \\ & \quad (\nexists v \in \overline{v}\colon M \in KM[v]) \wedge \\ 1 & \text{if } \left(\begin{array}{c} \exists \langle t, \{M_1, \ldots, M_s\}\rangle \in KM\colon \\ M \in \{M_1, \ldots, M_s\} \end{array}\right) \\ 1 - discFactor & \text{otherwise} \end{cases} \quad (4)$$

$$\text{with} \quad discFactor = \frac{|\{\langle t, \{M_1, \ldots, M_s\}\rangle \in NKM \mid M \in \{M_1, \ldots, M_s\}\}|}{|D_I|}$$

The intuition of *notKilledScore* is as follows:

- if the mutant M is killed by a test of the current individual (first case), its "not killed score" is 0 (the best case. Recall that we are minimizing);
- if the mutant M is not killed by a test of the current individual, but it has been killed by one of the previously generated tests (second case), its "not killed score" is 1. This is indeed the worst case in the fitness definition, as the mutant is not killed, but it is known to be killable;
- if the mutant M is not killed by the current individual, but it is also not known to be killable by the previously generated tests (third case), its "not killed score" is reduced from the worst value 1 of a given *discount factor*. This factor is the percentage of the generated tests that did not kill M over the total number of the possible inputs. The intuition is that the more tests we try that do not kill the mutant M, the more probable it is that M is an equivalent mutant that is not killable. Using this discount factor, we aim to avoid to keep on trying to kill mutants that are actually equivalent.

Remark. Note that, in our approach, all valid tests $VT_{\overline{v}}$ of an individual $\overline{v}$ contribute to the test suite. A different approach would have been to select the minimum subset of $VT_{\overline{v}}$ that is equivalent to the whole test suite $VT_{\overline{v}}$ in terms of number of killed mutants. This alternative approach has two drawbacks. First, finding the minimum test suite is known to be NP-hard [31] (however, this could be mitigated by only finding a "suboptimal" test suite that can be obtained in a liner time using a greedy algorithm). Second, it would require to run all the generated tests over each mutant, to have the complete information on killability (instead, in our approach, we stop as soon as a test kills the current mutant. See line 17 in Alg. 1). As future work, we plan to compare this alternative approach with the currently proposed approach MutTG in which the minimality of the test suite is achieved by the search algorithm.

4 EXPERIMENT DESIGN

In this section, we present the experimental design to evaluate the proposed approach. First, we present the research questions in

Table 2: Benchmarks – Quantum programs

Quantum program QP	$\|I\|$	# gates	# depth
Add Squared (AS)	10	39	37
Bernstein-Vazirani (BV)	8	24	3
Conditional Execution (CE)	10	25	20
invQFT (IQ)	10	60	56
QRAM (QR)	9	15	12

Sect. 4.1. Then, we list the benchmarks that we used in our experiments in Sect. 4.2. Experiment settings are described in Sect. 4.3, and evaluation metrics are introduced in Sect. 4.4. Code to reproduce the experiments, benchmark programs, and experimental results can be found online at [37].

4.1 Research Questions

We evaluate the proposed approach MutTG by the following RQs:

RQ1 What is the influence of the discount factor on the effectiveness of the search of MutTG?

RQ2 How does the *difficulty* of the benchmarks affect the effectiveness of MutTG?

RQ3 How does the number of equivalent mutants in the targeted mutants set affect the effectiveness of MutTG?

4.2 Benchmarks

We selected five programs with different characteristics (see Table 2) as quantum programs under test:

- *Bernstein-Vazirani* (BV) is a cryptography program [6];
- *QRAM* (QR) is an algorithm for getting access and operating quantum random access memory [11];
- *invQFT* (IQ) is an algorithm implementing inverse quantum Fourier transform [11];
- *Add Squared* (AS) demonstrates mathematical operations in superposition; and
- *Conditional Execution* (CE) demonstrates conditional execution in superposition.

The characteristics of the programs, measured in terms of number of input qubits, number of gates, and circuit depth (length of the longest sequence of quantum gates) are provided in Table 2. As shown in the table, the numbers of input qubits range from 8 to 10. The numbers of gates range from 15 to 60, and the depths of the quantum circuits from 3 to 56. The data of these two latter metrics, to a certain extent, indicate the complexity of the program logic.

MutTG takes in input a set of mutants *Muts* to be killed. We apply quantum gates in Table 1 to create these mutants. In the experiments, we use sets of ten mutants, i.e., $k = |Muts| = 10$. Some mutants are more difficult than others to kill; the difficulty depends on the percentage of inputs of the input domain D_I that can kill the mutants (the fewer inputs kill a mutant, the more difficult the task is). We built sets of mutants *Muts* with three levels of difficulty:

- *Easy* (*E*): all the mutants in this type of set are killed by at least 25% and at most 100% inputs of the input domain;
- *Medium* (*M*): all the mutants in this type of set are killed by at least 1.56% and at most 25% inputs of the input domain;
- *Difficult* (*D*): all the mutants in this type of set are killed by at most 1.56% inputs of the input domain. These are *subtle* mutants [25] that are very hard to kill.

MutTG adopts a special mechanism to estimate the likelihood that a mutant is equivalent (i.e., the *discount factor* in Eq. 4, described in Sect. 3.2.3). To assess the effectiveness of such a mechanism, we built two variants of difficult mutants sets containing, respectively, one and three equivalent mutants (identified as *D1* and *D3*).

For each program QP and difficulty X, we generated a mutants set $Muts^X_{QP}$. In our experiments, a *benchmark* is given by a quantum program and a mutant set. So, in total we generated 20 benchmarks (four mutants sets for each of the five quantum programs).

4.3 Experimental Settings

We use Qiskit 0.34.1 [38] for coding the quantum programs in Python. Qiskit is also equipped with a simulator for quantum program executions and we used it for evaluating each test case.

We selected NSGA-II as the search algorithm and used its implementation and default settings from jMetalPy 1.5.5 [5]. The settings are: binary tournament selection of parents, integer SBX crossover (the crossover rate = 0.9), polynomial mutation operation 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 1000. To check whether it is worth using NSGA-II to solve the problem, we implemented a version of the approach based on Random Search (RS) algorithm and gave RS the same number of fitness evaluations as NSGA-II, i.e., 10000.

The search variables for the search are defined as explained in Sect. 3.2.1. The definition of their domain requires to select a parameter β that specifies the relation between the size of *NoTestDom* (i.e., the part of the domain that does not represent tests) and D_I (i.e., the concrete input values). We choose $\beta = 1$, meaning that the search (when randomly generating individuals) has equal probability of producing and not producing a test.

To check whether a mutant is killed or not, we need to perform the Pearson Chi-square test for checking *wodf*, as discussed in Sect. 2.2. To do so, we used the Python interface to the R framework *rpy2 3.4.2* and selected $\alpha = 0.01$ as the significance level in the Pearson Chi-square test.

As suggested by guidelines to conduct experiments with randomized algorithms [3], to account for the randomness in the search algorithm, we ran 30 times each experiment, i.e., the execution for a given benchmark using a search approach (either MutTG, RS, or MutTG without the discount factor applied, i.e., MutTG_{ND}).

Experiments have been executed on the Amazon EC2, using instances with a 2.9 GHz Intel Xeon CPU, 3.75GB of RAM.

4.4 Evaluation Metrics and Statistical Tests

To evaluate the quality of the results of MutTG and of its variants it is compared to, we use the following evaluation metrics.

First, since the approach is multi-objective, we use a quality indicator [17] to evaluate the quality of the final Pareto fronts. We adopt *Hypervolume* [32] which is a well-known quality indicator (the higher, the better).

Moreover, we adopt two other metrics that independently assess the two desired qualities of the generated test suites. Since the

main goal of the approach is to kill all the mutants, we introduce the metric *MNNKM* that counts the *minimum number of not killed non-equivalent mutants* in the generated solutions (the lower, the better).[3] As an additional goal of the approach, we would like to obtain compact test suites. To assess this, we introduce the metric TSS_{MNNKM} that represents the *minimum size of the test suite achieving MNNKM* (the lower, the better).

Given a metric *Metr* (either *Hypervolume*, *MNNKM*, or TSS_{MNNKM}), we compare the performance of MutTG with that of a baseline approach (RS in Sect. 4.5 and $\texttt{MutTG}_{\texttt{ND}}$ in Sect. 5 - RQ1) using 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 guideline [3]. Namely, given a benchmark program, two approaches *App1* and *App2*, and a metric *Metr*, we compare the two distributions of 30 values of *Metr* (of the 30 runs) using the Mann-Whitney U test (with the significance level of 0.05). The null hypothesis is that there is no statistical difference between *App1* and *App2* in terms of *Metr*. If the null hypothesis is not rejected, then we consider the two approaches 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 *App1* has a higher chance to achieve higher values than *App2*, and vice versa if $\hat{A}_{12}$ is lower than 0.5. Which approach is "better" depends on the compared metric: for *Hypervolume*, *App1* is better if $\hat{A}_{12}$ is greater than 0.5, while, for *MNNKM* and TSS_{MNNKM}, *App1* is better if $\hat{A}_{12}$ is lower than 0.5.

4.5 Comparison with Random Search

We here assess whether the use of a search algorithm such as NSGA-II is motivated. To do so, we compared the implementation of the proposed approach MutTG based on NSGA-II with that based on Random Search (RS), which have been given the same budget. For all the 20 experiments, we executed 30 runs with NSGA-II and 30 with RS. We assessed their solutions using Hypervolume and compared them with proper statistical tests, as described in Sect. 4.4. Out of 20 experiments, NSGA-II is statistically significant better than RS in 9 cases, all *D1* and *D3* benchmarks but one, showing the advantage of NSGA-II in the difficult problems. In the remaining 11 benchmarks, there is no significant difference. So, in the following, we analyze the results of MutTG implemented using NSGA-II.

5 EXPERIMENTAL RESULTS

We discuss the results for answering each of the three RQs.

RQ1 – Influence of the Discount Factor. A characteristic feature of the proposed approach is the adoption, in the fitness function f_{not_killed} related to the number of non-killed mutants (see Eq. 3), of a *discount factor* that considers the likelihood that a mutant is equivalent (the third case in Eq. 4). The idea is that the "not killed score" of a mutant M is reduced by a discount factor that is proportional to the number of tests that have been executed over M but did not kill it; the higher the factor, the higher the likelihood that the mutant is equivalent. The adoption of the discount factor should allow the search avoiding to focus too much on mutants that cannot be killed. In this RQ, we are interested in investigating whether such discount factor does indeed provide any benefit to the search. We have implemented a version of the approach (called $\texttt{MutTG}_{\texttt{ND}}$) that does not adopt the discount factor (i.e., with the third case in Eq. 4 being 1).

Table 3: RQ1 – Influence of the discount factor – MutTG vs. $\texttt{MutTG}_{\texttt{ND}}$

(a) Hypervolume

Benchmark	Best app.
AS_E	≡
AS_M	≡
AS_{D1}	MutTG
AS_{D3}	MutTG
BV_E	≡
BV_M	MutTG
BV_{D1}	≡
BV_{D3}	MutTG
CE_E	≡
CE_M	≡
CE_{D1}	≡
CE_{D3}	MutTG
IQ_E	≡
IQ_M	≡
IQ_{D1}	MutTG
IQ_{D3}	MutTG
QR_E	≡
QR_M	≡
QR_{D1}	MutTG
QR_{D3}	MutTG

(b) *MNNKM*

Benchmark	Best app.
AS_E	≡
AS_M	≡
AS_{D1}	≡
AS_{D3}	≡
BV_E	≡
BV_M	≡
BV_{D1}	≡
BV_{D3}	≡
CE_E	≡
CE_M	≡
CE_{D1}	≡
CE_{D3}	MutTG
IQ_E	≡
IQ_M	≡
IQ_{D1}	≡
IQ_{D3}	MutTG
QR_E	≡
QR_M	≡
QR_{D1}	≡
QR_{D3}	MutTG

(c) TSS_{MNNKM}

Benchmark	Best app.
AS_E	≡
AS_M	≡
AS_{D1}	MutTG
AS_{D3}	MutTG
BV_E	MutTG
BV_M	MutTG
BV_{D1}	MutTG
BV_{D3}	MutTG
CE_E	≡
CE_M	≡
CE_{D1}	MutTG
CE_{D3}	MutTG
IQ_E	≡
IQ_M	≡
IQ_{D1}	MutTG
IQ_{D3}	MutTG
QR_E	≡
QR_M	≡
QR_{D1}	MutTG
QR_{D3}	≡

We have run $\texttt{MutTG}_{\texttt{ND}}$ 30 times on all the 20 benchmarks, and we have compared its results with those of MutTG in terms of Hypervolume[4], *MNNKM*, and TSS_{MNNKM}, as described in Sect. 4.4. Results are shown in Table 3. When looking at Table 3a, which is related to Hypervolume, MutTG is better in all the *D3* benchmarks (i.e., those with three equivalent mutants), and in three out of the five *D1* benchmarks (i.e., those with one equivalent mutant); MutTG is also better for BV_M. For all the other benchmarks, there is no statistically significant difference between the two approaches. This shows that using the discount factor as done by MutTG does indeed provide an advantage in the search when there are some equivalent mutants in the benchmark set (as in *D1* and *D3* benchmarks), in particular when the number of equivalent mutants is higher.

By looking at the results of Tables 3b-3c, we gain more insights as we can understand which part of the search is mainly improved by the use of the discount factor. From Table 3b related to *MNNKM* (i.e., the number of not killed non-equivalent mutants), we notice that MutTG is better for three *D3* benchmarks, and there is no difference for the other cases. This seems to show that, although the discount factor provides some advantage for *MNNKM* in some difficult cases with three equivalent mutants, it does not contribute much to the overall performance of MutTG.

From Table 3c related to TSS_{MNNKM}, instead, we notice that MutTG is better in almost all the *D1* and *D3* benchmarks (except

[3] We preferred this metric to the well-know *mutation score* as it is more directly related to the fitness function f_{not_killed}. In any case, the two metrics are equivalent.

[4] Note that, to be fair with $\texttt{MutTG}_{\texttt{ND}}$, the objective values obtained by MutTG for f_{not_killed} have been processed by removing the effect of the applied discount factor. Without doing so, MutTG could have a better performance simply because it has a fitness function that produces lower objective values.

for $\texttt{QR}_{D3}$), and also for other two BV benchmarks, while there is no difference in the other benchmarks. This shows that the main benefit provided by the discount factor is mostly on the optimization of the test suite size. Indeed, although $\texttt{MutTG}_{\texttt{ND}}$ is able to kill almost the same number of non-equivalent mutants (except for some difficult cases) as MutTG (as shown in Table 3b), it struggles in minimizing the test suite size, as it keeps on trying to kill the equivalent mutants. MutTG, instead, as the search progresses, can focus less on trying to kill the equivalent mutants and more on trying to minimize the test suite size (thanks to the guidance of the discount factor).

RQ2 – Influence of the Benchmark Difficulty. In this RQ, we are interested in investigating to what extent the difficulty of the benchmarks affects the performance of the proposed approach MutTG. Fig. 2 shows, for each benchmark, how *MNNKM* and TSS_{MNNKM} change over the generations (values are the average across the 30 runs). We observe that for the *E* benchmarks (the first column), all the non-equivalent mutants are killed since the early generations; indeed, these are benchmarks in which several different inputs can kill the mutants, and so it is not challenging to find them. In this case, the main effort of the search is only about reducing the test suite size.

For the *M* benchmarks (the second column), the difficulty of killing mutants is a little bit higher. Thus, the search takes few generations to kill all the non-equivalent mutants. Still, the most difficult optimization task is the minimization of the test suite size.

Regarding the benchmarks of type *D1* and *D3* (last two columns), instead, we observe that minimizing the number of killed non-equivalent mutants is more difficult, and can take more than 500 generations to reach the minimum. In some cases, e.g., $\texttt{CE}_{D1}$ and $\texttt{QR}_{D1}$, the search was not able to kill all the non-equivalent mutants (around two non-equivalent mutants were not killed on average).

By considering all the benchmarks, we observe that the test suite size can sometimes reach values as low as 1 or 2, while other times it reaches higher values around 4 and 5. This depends on the set of mutants that we want to kill: if they share more inputs that can kill several of them, this allows smaller test suites; instead, if they have disjoint killing inputs, this requires bigger test suites.

RQ3 – Influence of the Number of Equivalent Mutants. We are here interested in investigating the influence of the number of non-equivalent mutants in the benchmark set on the performance of MutTG. By looking at *D1* and *D3* benchmarks in Fig. 2, it seems that killing all the non-equivalent mutants of the *D1* benchmarks is more difficult than killing all those of the *D3* benchmarks, as evidenced by the higher *MNNKM* values of $\texttt{CE}_{D1}$ and $\texttt{QR}_{D1}$ than their counterparts $\texttt{CE}_{D3}$ and $\texttt{QR}_{D3}$. This is reasonable as, although the difficulty of killing each mutant is similar in the two types of benchmarks, *D1* benchmarks contain more mutants to kill (i.e., 10 deducted by the number of equivalent mutants). So, it is natural that they are more challenging.

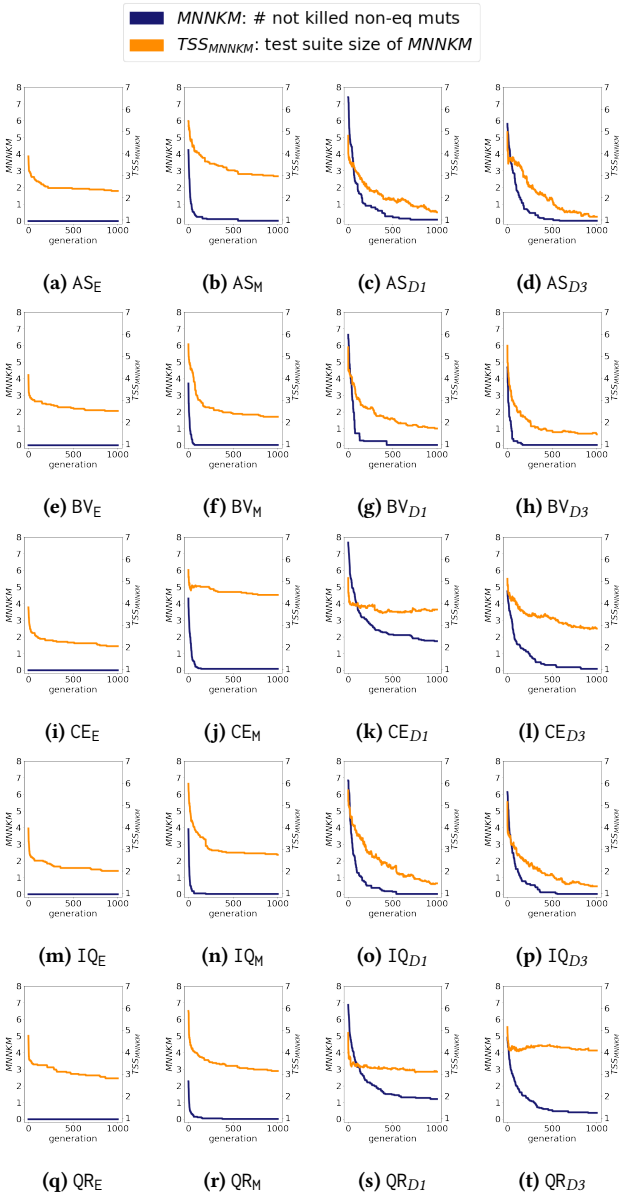

Figure 2: RQ2 and RQ3 – Evolution of *MNNKM* and TSS_{MNNKM}

6 THREATS TO VALIDITY

First, we acknowledge that the number of quantum programs and their mutants in our experiments is small. Therefore, a more extensive collection of quantum programs with diverse characteristics and their mutants of varying complexity is required to gain more confidence in our results. Moreover, currently, we created mutants manually. Such a manual process needs to be replaced with a systematic and automated approach to create a more extensive set of mutants in the future. Second, we used NSGA-II with its default parameters settings implemented in jMetal. One could argue that default parameters are not good enough. However, note that credible evidence exists suggesting that the default parameter settings are good [4]. Third, we employed the Pearson's Chi-square test to judge whether a test case passed/failed. This test is commonly used in the quantum software testing community for this

purpose [13, 35]. Fourth, we followed the well-established guide in search-based software engineering by Arcuri and Briand [3] to repeat experiments 30 times and apply the relevant statistics to compare MutTG with the two baselines.

7 RELATED WORK

We discuss the related work from the aspects of search-based mutation testing and quantum software testing.

Search-Based Mutation Testing. Search-based mutation testing (SBMT) works by formulating a test case generation problem into a search problem (solved by meta-heuristics algorithms) and applying search-based techniques to mutation testing. In recent years, various SBMT methods have been proposed. In terms of search algorithms used in these SBMT methods, Genetic Algorithm (GA) is still the most popular search algorithm used by researchers for test data generation [8, 16, 29, 30], although hill climbing (HC) [33] and particle swarm optimization (PSO) [14] have been also applied. Below, we discuss some representative SBMT methods, which, however, by no means, are complete.

Fraser and Zeller [10] proposed μTEST, a search-based unit test generation approach for object-oriented classes, which uses mutations rather than structural properties as the test coverage criterion. GA was used to breed sequences of method/constructor calls that are effective in detecting mutants.

Papadakis et al. [27] proposed an automatic framework that makes the joint use of automated tools and techniques containing symbolic execution, concolic execution, and search-based optimization techniques for automated mutation based test data generation. Motivated by the observation that it is considered expensive to apply mutation testing in practice, Papadakis et al. [26] utilized search-based testing to generate test inputs capable of killing mutants, and also proposed a dynamic execution scheme for introducing and guiding the search towards sought mutants. Souza et al. [33] later on extended [26] by introducing another search-based automated test generation approach, which combines weak and strong mutation approaches, i.e., incrementally targeting at strongly killing mutants by focusing on their propagation.

Harman et al. [12] put forwards a hybrid approach integrating dynamic symbolic execution and search-based software testing to strongly kill both first-order and higher-order mutants. Their empirical results using 17 programs show the efficiency and effectiveness of the approach for strong first-order mutation adequacy.

EvoSuite [9] is a search-based test generation tool, which avoids redundant test executions on mutants by monitoring program state infection conditions, and optimizing test suites toward killing as many mutants as possible. EvoSuite was evaluated with a random sample of 100 open source projects and results show that EvoSuite is scalable, which makes mutation testing a viable test criterion to be used in automated test case generation.

Our approach focuses on first order mutation testing; however, we would like to acknowledge that a considerable effort has been made in the context of search-based higher order mutation testing [16, 18, 22–24, 39].

Quantum Software Testing. Differently from classical software testing, testing of quantum programs is more challenging because of its inherent characteristics such as superposition and entanglement. Recently, some methods and tools have been proposed.

QSharpCheck [13] is a property-based testing tool for Q# programs supporting test case generation, test execution, and test results analysis. It has a test property specification language containing the number of tests to generate, statistical confidence level, the number of measurements, and the number of experiments. QSharpCheck has been evaluated with two example quantum programs via mutation analysis and achieved 80% and 60% of mutation scores for the two programs, respectively.

QuanFuzz [34] applied a grey-box fuzz testing model as a search-based test input generator for quantum software. It can automatically choose a better input to trigger quantum-sensitive branches. QuanFuzz was evaluated with seven benchmarks and outperformed traditional testing methods with a higher branch coverage.

Quito [36] is a quantum software testing tool having three coverage criteria defined on inputs and outputs of quantum programs, and two types of test oracles based on program specifications. Quito has been evaluated with a set of benchmarks. Evaluation results indicate that Quito is effective in terms of detecting faults.

QuSBT [35] is a search-based approach for automatically generating test suites for quantum programs. The goal of QuSBT is to find inputs that fail the original program under test; for doing so, it uses a single objective whose aim is to maximize the number of failing test cases in the generated test suite within a given budget. The goal is then different from that of our approach MutTG that aims at generating tests that targets specific faults captured by the used mutants. QuSBT can be used to test only the current program, while MutTG supports the two usages of mutation testing: generation of tests for the current program and for future evolutions of the program (in a regression testing setting).

8 CONCLUSION AND FUTURE WORK

In this paper, we presented a multi-objective search-based mutation testing approach, named MutTG, for generating a minimum number of test cases while killing as many mutants as possible in the context of testing quantum programs. We applied NSGA-II for our search problem and adopted 20 benchmarks for the empirical study. We compared NSGA-II with Random Search for sanity check. Then, we demonstrated the effectiveness of adopting the *discount factor* in MutTG to avoid meaningless effort required for keeping on trying to kill mutants that are not killable, i.e., equivalent mutants. Moreover, we introduced and controlled the *difficulty* of the benchmarks (defined as the the percentage of inputs of the input domain D_I that can kill the mutants), and demonstrated that MutTG is effective for killing mutants of the difficult benchmarks.

Future works include using other search algorithms, quantum programs, and mutant operators (e.g., those in [20]).

ACKNOWLEDGMENTS

This work 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. P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST; Funding Reference number: 10.13039/501100009024 ERATO.

REFERENCES

[1] Alan Agresti. 2019. *An introduction to categorical data analysis* (3 ed.). Wiley-Blackwell.

[2] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. 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)*. 13–23. https://doi.org/10.1109/ICST49551.2021.00014

[3] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795

[4] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

[5] Antonio Benítez-Hidalgo, Antonio J. Nebro, José García-Nieto, Izaskun Oregi, and Javier Del Ser. 2019. jMetalPy: A Python framework for multi-objective optimization with metaheuristics. *Swarm and Evolutionary Computation* 51 (2019), 100598. https://doi.org/10.1016/j.swevo.2019.100598

[6] Ethan Bernstein and Umesh Vazirani. 1993. Quantum Complexity Theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '93)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/167088.167097

[7] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 597–608. https://doi.org/10.1109/ICSE.2017.61

[8] Antonia Estero-Botaro, Antonio García-Domínguez, Juan José Domínguez-Jiménez, Francisco Palomo-Lozano, and Inmaculada Medina-Bulo. 2014. A Framework for Genetic Test-Case Generation for WS-BPEL Compositions. In *Proceedings of the 26th IFIP WG 6.1 International Conference on Testing Software and Systems - Volume 8763* (Madrid, Spain) *(ICTSS 2014)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-662-44857-1_1

[9] Gordon Fraser and Andrea Arcuri. 2015. Achieving Scalable Mutation-Based Generation of Whole Test Suites. *Empirical Softw. Engg.* 20, 3 (jun 2015), 783–812.

[10] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292. https://doi.org/10.1109/TSE.2011.93

[11] M. Gimeno-Segovia, N. Harrigan, and E.R. Johnston. 2019. *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated.

[12] Mark Harman, Yue Jia, and William B. Langdon. 2011. Strong Higher Order Mutation-Based Test Data Generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 212–222. https://doi.org/10.1145/2025113.2025144

[13] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. Property-Based Testing of Quantum Programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 430–435. https://doi.org/10.1145/3387940.3391459

[14] Nishtha Jatana, Bharti Suri, Sanjay Misra, Prateek Kumar, and Amit Roy Choudhury. 2016. Particle Swarm Based Evolution and Generation of Test Data Using Mutation Testing. In *Computational Science and Its Applications – ICCSA 2016*. Springer International Publishing, Cham, 585–594.

[15] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[16] Yue Jia, Fan Wu, Mark Harman, and Jens Krinke. 2015. Genetic Improvement Using Higher Order Mutation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 803–804. https://doi.org/10.1145/2739482.2768417

[17] Miqing Li and Xin Yao. 2019. Quality Evaluation of Solution Sets in Multiobjective Optimisation: A Survey. *ACM Comput. Surv.* 52, 2, Article 26 (March 2019), 38 pages. https://doi.org/10.1145/3300148

[18] Jackson A. Prado Lima and Silvia Regina Vergilio. 2018. Search-Based Higher Order Mutation Testing: A Mapping Study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing* (SAO CARLOS, Brazil) *(SAST '18)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/3266003.3266013

[19] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[20] Ñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1266–1270. https://doi.org/10.1109/ASE51524.2021.9678563

[21] Andriy Miranskyy and Lei Zhang. 2019. On Testing Quantum Programs. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results* (Montreal, Quebec, Canada) *(ICSE-NIER '19)*. IEEE Press, 57–60. https://doi.org/10.1109/ICSE-NIER.2019.00023

[22] Quang Vu Nguyen and Lech Madeyski. 2015. Searching for strongly subsuming higher order mutants by applying multi-objective optimization algorithm. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 391–402.

[23] Quang Vu Nguyen and Lech Madeyski. 2016. Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants. *Cybernetics and Systems* 47, 1-2 (2016), 48–68.

[24] Quang Vu Nguyen and Lech Madeyski. 2016. Higher order mutation testing to drive development of new test cases: An empirical comparison of three strategies. In *Asian Conference on Intelligent Information and Database Systems*. Springer, 235–244.

[25] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[26] Mike Papadakis and Nicos Malevris. 2013. Searching and generating test inputs for mutation testing. *SpringerPlus* 2, 1 (2013), 1–12.

[27] Mike Papadakis, Nicos Malevris, and Maria Kallia. 2010. Towards Automating the Generation of Mutation Tests. In *Proceedings of the 5th Workshop on Automation of Software Test* (Cape Town, South Africa) *(AST '10)*. Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/1808266.1808283

[28] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 537–548. https://doi.org/10.1145/3180155.3180183

[29] Shweta Rani and Bharti Suri. 2015. An Approach for Test Data Generation Based on Genetic Algorithm and Delete Mutation Operators. In *2015 Second International Conference on Advances in Computing and Communication Engineering*. 714–718. https://doi.org/10.1109/ICACCE.2015.145

[30] C Prakasa Rao and P Govindarajulu. 2015. Genetic algorithm for automatic generation of representative test suite for mutation testing. *International Journal of Computer Science and Network Security (IJCSNS)* 15, 2 (2015), 11.

[31] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249. https://doi.org/10.1002/stvr.256

[32] Ke Shang, Hisao Ishibuchi, Linjun He, and Lie Meng Pang. 2021. A Survey on the Hypervolume Indicator in Evolutionary Multiobjective Optimization. *IEEE Transactions on Evolutionary Computation* 25, 1 (2021), 1–20. https://doi.org/10.1109/TEVC.2020.3013290

[33] Francisco Carlos M. Souza, Mike Papadakis, Yves Le Traon, and Márcio E. Delamaro. 2016. Strong Mutation-Based Test Data Generation Using Hill Climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing* (Austin, Texas) *(SBST '16)*. Association for Computing Machinery, New York, NY, USA, 45–54. https://doi.org/10.1145/2897010.2897012

[34] Jiyuan Wang, Fucheng Ma, and Yu Jiang. 2021. Poster: Fuzz Testing of Quantum Program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 466–469. https://doi.org/10.1109/ICST49551.2021.00061

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

[36] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a Coverage-Guided Test Generator for Quantum Programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1237–1241. https://doi.org/10.1109/ASE51524.2021.9678798

[37] Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search – Supplementary material. https://github.com/Simula-COMPLEX/MutTG-paper.

[38] Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1234–1240. https://doi.org/10.23919/DATE.2019.8715261

[39] Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. 2016. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*. Springer, 18–33.

[40] Tao Yue, Paolo Arcaini, and Shaukat Ali. 2022. Quantum Software Testing: Challenges, Early Achievements, and Opportunities. *ERCIM News* 2022, 128 (2022). https://ercim-news.ercim.eu/en128/special/quantum-software-testing-challenges-early-achievements-and-opportunities

[41] Jianjun Zhao. 2020. Quantum Software Engineering: Landscapes and Horizons. *CoRR* abs/2007.07047 (2020). arXiv:2007.07047