Generating Test Input with Deep Reinforcement Learning

Junhwi Kim
KAIST
Daejeon, Republic of Korea
junhwi.kim23@kaist.ac.kr

Minhyuk Kwon
Suresoft Technology
Seoul, Republic of Korea
minhyuk@suresofttech.com

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

ABSTRACT

Test data generation is a tedious and laborious process. Search-based Software Testing (SBST) automatically generates test data optimising structural test criteria using metaheuristic algorithms. In essence, metaheuristic algorithms are systematic trial-and-error based on the feedback of fitness function. This is similar to an agent of reinforcement learning which iteratively decides an action based on the current state to maximise the cumulative reward. Inspired by this analogy, this paper investigates the feasibility of employing reinforcement learning in SBST to replace human designed metaheuristic algorithms. We reformulate the software under test (SUT) as an environment of reinforcement learning. At the same time, we present GunPowder, a novel framework for SBST which extends SUT to the environment. We train a Double Deep Q-Networks (DDQN) agent with deep neural network and evaluate the effectiveness of our approach by conducting a small empirical study. Finally, we find that agents can learn metaheuristic algorithms for SBST, achieving 100% branch coverage for training functions. Our study sheds light on the future integration of deep neural network and SBST.

ACM Reference Format:
Junhwi Kim, Minhyuk Kwon, and Shin Yoo. 2018. Generating Test Input with Deep Reinforcement Learning. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Search-Based Software Testing (SBST) has been shown to be effective in automatic test data generation, in particular, for structural coverage. Various metaheuristic techniques were employed in software test data generation, including hill climbing, simulated annealing, and evolutionary algorithms [14].

Metaheuristic algorithms solve problems essentially by trial-and-error. Algorithms iteratively evaluate candidate solutions and generate better solutions based on the feedback from the fitness function. This iterative process can be viewed as algorithms making a series of decisions based on the rewards (i.e. improvements of fitness values). If a decision improves the fitness value, it can be considered as getting rewards. Consequently, we can think of metaheuristic algorithm as agents following certain policy to make decisions. In this context, problem instance of metaheuristic algorithm can be described as an *environment* for the agent.

Reinforcement Learning (RL) is a machine learning technique that seeks to learn the optimal control policy for agents interacting with an unknown environment. In RL, agents try and evaluate an action, and make the next decision based on the observation of the feedback from the environment. Playing video games is one of the most widely known examples of decision-making environment in terms of RL, and recent advances in deep learning and RL have been shown to be capable of training the human level agents for a series of Atari 2600 games [16].

The analogy between the metaheuristic algorithm and RL leads us to raise an interesting question: can we train the agent to solve SBST problem? Yoo [24] proposed the idea of reformulating SBST as gaming. Considering SBST as gaming, we may learn the control policy, that is a new metaheuristic algorithm, for SBST. Most existing metaheuristic algorithms are designed manually: RL may enable us to automate the designing of a new algorithm through training of RL agents.

To answer the question about the feasibility of using RL for SBST, we formulate search-based test data generation problem as an RL environment, and subsequently train and test a Double Deep Q-Networks (DDQN) [21] on various branch predicates that take numerical inputs. We note that, to the best of our knowledge, this is the first attempt to use an RL agent in the context of SBST.

The contributions of this paper are as follows:

(1) We introduce a general, open-source framework, GunPowder, which instruments the software under test (SUT) and calculates the fitness value for given structural testing criteria. It provides a platform to apply different search methods to SBST including RL algorithms. GunPowder is compatible with the widely used RL platform OpenAI Gym [2].
(2) We reformulate SBST as a reinforcement learning environment, and subsequently train and evaluate an RL agent which can replace the metaheuristic algorithms designed by a human. We present a small empirical study that evaluates the effectiveness of our approach.

While the results are not immediately practical, we hope that our study sheds light on the future use of reinforcement learning and deep neural networks in SBST. Let us begin with the basic background on reinforcement learning.

2 BACKGROUND

2.1 Reinforcement Learning

RL aims to learn the optimal control policies for agents that interact with an unknown environment, $\mathcal{E}$. The goal of an agent is to choose a sequence of actions, by observing $\mathcal{E}$, that maximises the cumulative reward over all time steps.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This process is formally represented as a Markov Decision Process (MDP), defined by tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ is the set of states, and $\mathcal{A}$ is the set of actions. At each time step t, the agent takes an action $a_t \in \mathcal{A}$ based on observation of a state $s_t \in \mathcal{S}$ and moves to next state $s_{t+1} \sim \mathcal{P}(s_t, a_t)$ receiving feedback which is a scalar reward, $r_t \sim \mathcal{R}(s_t, a_t)$. $\mathcal{P}(s_t, a_t)$ denotes the transition probability from s_t to s_{t+1} due to action a_t. $\mathcal{R}(s_t, a_t)$ is the immediate reward after taking action a_t at state s_t. The return at time step t is defined as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step when the game terminates. The future rewards are discounted by a factor of $\gamma \in [0, 1]$. The objective of RL is to learn a policy π which maps states to a probability distribution over the actions. A policy defines the behaviour of an agent and optimal policy π^* maximises the expected return from start R_0.

The action-value function Q is the expected return starting from state s_t, after taking action a_t and thereafter following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

Bellman equation expresses the action-value function in recursive form:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1})]$$

Q-Learning [22] is a model-free, off-policy algorithm that is widely used. It uses a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q(s, a)$. Deep Q Networks (DQN) employs a large neural network as a function approximator, which is referred to as the Q-network [16]. A Q-network is trained by minimising the loss, which is defined as follows, where $y_t = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; \theta)$:

$$L(\theta) = \mathbb{E}[(y_t - Q(s_t, a_t; \theta))^2]$$

The use of Q-function approximation may result in unstable behaviours, as the same network that generates the target Q-values is also used in updating its Q-values [20]. Another problem is consecutive data samples are highly correlated, making Q-networks diverge when learning from them. To alleviate these problems, Deep Q Networks introduce two major changes: experience replay and a separate target network for calculating y_t.

First, DQN stores the agent's experience e_t at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D_t = \{e_1, ..., e_t\}$ which is called the replay memory. Then the Q-network is updated with the random samples from D, which is called the experience replay technique. This technique allows us to avoid a strong correlation between consecutive data samples. Second, DQN uses a separate target network for the generation of the targets, y_i, in the loss function. This target network is updated at every fixed step intervals by cloning the parameter of Q-network, resulting in more stable learning. With these modifications, DQN showed that stable learning using the neural network is possible. The big advantage of using the neural network is that it enables extracting high-level features from raw data, which can be directly used for RL.

In this paper, we use Double DQN (DDQN) [21], which is an improved version of DQN. The overestimation is the problem of learning unrealistic high action values. Q-learning is known to have this problem in some cases [19]. DDQN solves this problem by using a different network to select and evaluate an action. The target used by DQN is

$$y_t = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a; \theta)$$

It uses same weight θ to select an action and evaluate chosen action. Instead of it the target of DDQN is defined as

$$y_t = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta'); \theta)$$

where θ is the weights of the target network, and θ' is the weights of the online network. As in DQN, the target network is periodically copied from the online network. Note that the weights θ' is used to select an action, while θ evaluates the value of this policy. As a result, DDQN avoids overestimation of Q-learning.

2.2 GunPowder: A General Framework for SBST

Search based test data generation is a dynamic technique that involves program instrumentation. We summarise general instrumentation and analysis requirements for test data generation as follows:

(1) Instrumenting SUT to measure structural test criteria
(2) Executing SUT with candidate input generated from search algorithms
(3) Calculating the fitness value

A general framework satisfying these requirements would enable effective and efficient comparison of different search methods, which can be implemented on top of the framework. While IGUANA [15] provided a general framework for `C`, it is no longer actively maintained. We present GunPowder, a general framework for search-based test data generation for structural testing for `C`, written in `Python`.

2.2.1 Framework. GunPowder consists of three parts: instrumentation, execution, and fitness evaluation. First, GunPowder instruments the source code of SUT to allow fitness values for a specific test adequacy criterion: currently GunPowder supports `C` programs and branch coverage. Instrumentation is performed at the source code level using `Clang` front-end from `LLVM` framework [10]. Compared to `javacc` used by IGUANA, `Clang` provides more robust analysis and manipulation of `C` code.

Second, GunPowder runs SUT with a given input. In case of `C` programs, GunPowder builds the SUT as a shared library to save the effort of making additional driver codes. This library is invoked directly through foreign function interface of `Python`; GunPowder is implemented in `Python`. When instrumented SUT is executed, it returns the trace of execution. The trace is passed to the fitness function, which can be any user-defined `Python` function computed from the execution trace and the control structure of the SUT.

GunPowder is designed to be extendible. By providing the control structure analysis and instrumentation for new languages, we can extend GunPowder for languages other than `C`. Additionally, new search algorithms for SBST can be implemented and evaluated using its `Python` interface.

2.2.2 Fitness Function. Currently GunPowder supports the standard fitness function for branch coverage that consists of the approach level, A, and the branch distance, Δ. The fitness value

is obtained using the execution trace of a candidate solution and control dependency information. When a node n_i in instrumented SUT is executed, a trace record $t_i = (i, c_i(X), \Delta(X, c_i), \Delta(X, \neg c_i))$ is stored at trace $T(X) = \{t_i | n_i \in P\}$ where P is the list of executed nodes and c_i is the predicate for node n_i.

When instrumenting the SUT, GunPowder also analyses its control structures. Based on the analysis, we can obtain the desired execution path P_d for any target branch. By comparing the execution trace T and the desired path P_d, we then count how many nodes in P_d are not in actual execution path P. The number of these un-encountered nodes are assigned to approach level A for input X. At the same time, the diverging node n_c can be identified comparing T and P_d. The value $\Delta(X, c_c)$ in trace record t_c are assigned to branch distance Δ for input X. The combination of A and Δ is returned as fitness value.

3 QTIP: TEST DATA GENERATION USING Q-LEARNING

We present Qtip, a test data generation technique based on Q-learning. Qtip consists of Qtip environment which converts SUT to RL environment and Qtip agent trained with Qtip environment using Q-learning. In this paper, we use DDQN to train Qtip agent.

3.1 Formulating SBST as a Reinforcement Learning Problem

SBST considers the test data generation as optimisation problem and adopts the metaheuristic algorithms to solve it. Through iterative trial-and-error, algorithms try to find a qualifying solution following the feedback from the fitness function. This can be viewed as a continuous decision process. An algorithm, which is mapped to an agent, makes an action by creating a new candidate solution. After that, it receives a reward expressed as the improvement of the fitness value from the fitness function.

In this framework, test data generation remains an optimisation problem, but the metaheuristic algorithm is replaced by an agent trained using RL. Accordingly, an SBST problem is mapped to an environment and represented as an instance of MDP, which is a formal representation of RL problem defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$.

Depending on how we define $s_t \in \mathcal{S}$, $a_t \in \mathcal{A}$, and $r_t \sim \mathcal{R}(s_t, a_t)$, the performance of agent will change. In this paper, we propose the following formulations.

3.1.1 State and Action. Most search based test data generation depends on the concept of approach level, $\mathcal{A}$, and branch distance, Δ. The fitness function is defined as:

$$F(X) = A(X) + \mathcal{N}(\Delta(X, c_c))$$

where $\mathcal{N} : \mathbb{R} \rightarrow [0, 1)$ is a normalisation function.

The information used by a metaheuristic algorithm is the current approach level and branch distance. Similarly, we formulate the state at time step t in an RL environment as:

$$s_t = \{A(X_t), \mathcal{N}(\Delta(X_t, c_c))\}$$

where X_t is the input vector at time step t. This formulation means that an agent gives an input vector to the SBST environment then the environment evaluates the given input and returns the approach level, A, and the branch distance, Δ. The agent observes those responses and considers it as the current state of the environment.

At every time step t, an agent has to provide a candidate solution which is an input vector for SUT. It corresponds to an action from the perspective of a decision process. Therefore, an action should result in a new candidate solution. In case of hill climbing, a new solution is generated by modifying the current solution, a process that is described as moving to a neighbouring solution in the search landscape. Similarly, we define the action space as a set of possible manipulations for current input vector, X, as follows:

$$\mathcal{A} = \{a_i | i \in [1, 2 \cdot |X|]\} \quad a_i = \begin{cases} x_i \leftarrow x_i + 1 & \text{if } i \text{ is even} \\ x_i \leftarrow x_i - 1 & \text{if } i \text{ is odd} \end{cases}$$

where x_i is the i-th element of the input vector X. Since we defined two possible manipulations for each element of input vector, the size of action space is twice the length of the input vector.

As a feasibility study, we limit our study to functions with numerical inputs only. However, we believe that the action-based formulation can be subsequently extended to cater for other input types. For example, generating dynamic data structures and pointer inputs for `C` programs has been a challenge in SBST, for which various approaches have been proposed [5, 9]. The action based formulation allows defining various actions for different data types, including memory allocations or even invocations of other search heuristics.

3.1.2 Partial Observability. One inherent limitation in the above formulationm (i.e., the one based on the fitness of the current candidate input) is that the agent only has partial observation of the *trajectory* of the search. Consider the hill climbing algorithm: the algorithm is equipped with a (simple) memory mechanism, i.e., a variable that stores the fitness value of the previous candidate solution, to guide the search. Compared to this, in our decision process formulation, the agent has no information about the previous state. While it may be possible to learn to minimise the fitness function in an absolute (i.e., the current fitness should be as small as possible) rathern than relative (i.e., the current fitness should be smaller than the previous one) fashion, we posit that the information about the search trajectory may improve the performance of our RL agent.

To evaluate this hypothesis, we present an alternative formulation that includes previous observations of fitness values as part of the state. Formally, observation $o^{(t)}$ at time step t becomes

$$o^{(t)} = (a_t, A(X_t), \Delta(X_t, c_c)), s_t = \{o^{(t)}, ..., o^{(t-m+1)}\}$$

where m is the size of memory, which is referred to as window size, and $a_t \in \mathcal{A}$ is the action taken at the time step t. We train agents with both state representations: one with a window size of 200, and one without a window. Section 5.4 discusses the impact of using this window.

3.1.3 Reward. The last component of the decision process is the reward. The goal of RL is to maximise the cumulative reward from the initial time step, $R_t = \sum_{t=0}^{T} \gamma^t r_t$, where T is the time step when the game terminates. Rewards should be designed to drive agent to behave in the way expected.

In SBST, the standard optimisation goal for structural testing is to minimise the fitness value, i.e., $A + \mathcal{N}(\Delta)$. Similarly, the expected behaviour for our RL agent is to continuously decrease the fitness value: we should reward the agent when it decreases the fitness value. Consequently, the reward r_t at step t is defined as the amount of fitness decrease,

$$r_t = F(X_{t-1}) - F(X_t)$$

where X_t is the input vector and $F(X) = A(X) + \mathcal{N}(\Delta(X, c_c))$. With this, the agent gets a positive reward when it decreases the fitness value.

3.2 OpenAI Gym Compatibility of GunPowder

OpenAI Gym [2] is an open-source library written in `Python`, and aims to support the development and the comparison of different RL algorithms by providing a standardized set of environments for RL. We have developed GunPowder to be OpenAI Gym compatible to open up SBST as a future research topic for various RL algorithms.

The class `Env` in OpenAI Gym defines a standard interface for RL environment. Following is the required method of `Env`[1]

- `reset(self)`: Resets the environment's state. Returns observation.
- `step(self, action)`: Executes the environment by one time step and returns observation, reward, done, info.
- `render(self, mode='human', close=False)`: Renders one frame of the environment visually, if required.

To make a new OpenAI Gym environment, these methods have to be implemented. Since SBST has no visual representation, we only implements `reset` and `step`.

In a Qtip environment, `step` method performs the manipulation corresponding to the given action for the current input vector X. Subsequently, it executes SUT with a modified input and calculates the fitness value, which the agent observes and uses as the reward. In each episode, the agent can perform a specified number of actions, which corresponds to the fitness evaluation budget in metaheuristic algorithms. If the agent uses up all budgets or covers the target branch, the episode ends.

Both GunPowder and our Qtip environment operate at the function-level: they instruments a target function of SUT, and evaluate fitness for a specific branch in that function. Therefore, one function consists one instance of a Qtip environment. When the new episode starts, the environment sets a new target branch in the target function, which is implemented in the `reset` method.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We seek to answer following research questions with the empirical study.

- RQ1. Effectiveness: What is the average level of structural coverage that a Qtip agent can achieve?
- RQ2. Unseen Structures: Can the Qtip agent cover functions which is not seen during training?
- RQ3. Unseen Input Ranges: Can the Qtip agent cover input in ranges not seen during training?

The goal of our approach is to learn the control policy that achieves structural coverage. We answer RQ1 by measuring the branch coverage achieved by a Qtip agent. Due to the stochastic nature of the RL algorithm, we repeat 30 attempts and report the average coverage.

To replace metaheuristic algorithms, agents should be generally effective, i.e., they should be capable of achieving coverage for unseen arbitrary branches. Additionally, agents should be able to learn the policy regardless of the size of search space. RQ2 addresses the issue of generalisability by measuring the achieved branch coverage for functions that have not been seen during training. RQ3 focuses on the effect of the input range on the performance. We repeat 30 runs for both RQ2 and RQ3 for each branch.

4.2 Training

We train Qtip using DDQN algorithm to cover the set of basic binary relational operators, ==, !=, <, <=, >, and >=, as well as the unary logical negation, !. Our training program contains a series of if-statements, each of which contains one of the basic relational operators. All branches are not nested, resulting in approach level A being always zero. The target function takes two integer arguments, i.e., $|X| = 2$ and $|\mathcal{A}| = 4$. The default range of both input variables is set to $[-128, 128]$.

In each episode, a random branch is chosen as the target, to avoid overfitting to a specific branch type. Input values are initialised with random numbers that do not cover the target branch. The agent is given the budget of 2,000 actions, and the episode ends when either the agent covers target condition or it runs out of the budget. The reward is discounted by a factor of $\gamma = 0.99$ per each step, and the size of the replay buffer is set to 5,000. As stated in Section 3.1.2, we use state window of size 200 to keep the information of previous states.

4.3 Neural Network Architecture

Table 1 shows the architecture of our Q netrowk. To approximate the Q function, we use three fully-connected layers with 16 nodes. Apart from the activation_4 layer, all activation layer uses Rectified Linear Unit (ReLU) [17] activation function.

The network architecture has a significant impact on the performance of an agent. For example, DQN has adopted convolutional layers [11] to extract high-level features from screenshots of Atari 2600 games [16]. However, as an initial feasibnility study, we keep the architecture of network as simple as possible: the optimisation of the network architecture is left as future work.

Following Timothy et al. [13], we update the target network with some delay: $\theta' \rightarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$, which is called "soft" target updates. The online Q-network, however, is updated every step.

We use the Adam [7] optimisation algorithms with the mini batch size of 32. The learning rate, lr, is defined the decay as:

$$lr_i = lr_{i-1} \times \frac{1}{(1 + D \times i)}$$

[1] https://github.com/openai/gym#basics

where lr_0 is 10^{-7} and decay factor D is 10^{-6}. We Learning rate decay helps fast convergence of optimisation.

4.4 Subjects

As stated in Section 4.2, our training function contains branches with basic predicates without any nested structure. After training, we used three small functions to test the performance of QTIP: we apply the trained agent to these unseen test functions and measure the branch coverage. Table 2 presents the studied functions. All functions take two integer as input: GCD and EXP are well known algorithmic examples, whereas the function Remainder has been chosen from the IGUANA [15] benchmark.

Currently GUNPOWDER supports only `C` programs for fully automated instrumentation. However, we implemented all training and testing functions in `Python`: this is because the interprocess communication overhead became exorbitant, as QTIP requires a significantly larger number of fitness evaluations compared even to the random search. Consequently, we manually instrumented `Python` implementations of of the training function as well as those in Table 2, and wrote a small `Python` driver between GUNPOWDER and QTIP, to avoid the overhead. Despite this workaround, please note that 1) GUNPOWDER is still fully compatible with OpenAI Gym for `C` targets, and 2) GUNPOWDER can easily extended to instrument `Python` target functions as well.

5 RESULTS

5.1 Effectiveness

A common way of monitoring the training of RL is to monitor the change of total rewards received in each episode. However, in case of QTIP, the main concern is whether the target branch is covered or not, rather than the total reward in each episode. Consequently, to monitor in-training improvements, we measure the number of branches covered by the agent at every 100 episodes. Figure 1a shows that QTIP does learn a policy that results in covering more branches. The training played total 5,700 episodes, and it took 5 hours and 44 minutes.

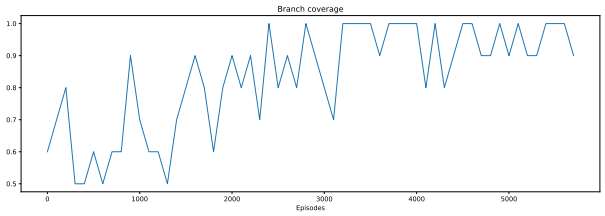
[2]https://en.wikipedia.org/wiki/Euclidean_algorithm
[3]https://en.wikipedia.org/wiki/Exponentiation_by_squaring

Table 1: The summary of architecture of Q network.

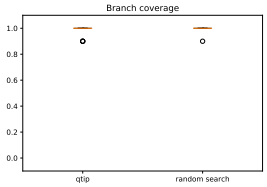
Layer (type)	Output Shape	Number of Parameters
flatten_1 (Flatten)	600	0
dense_1 (Dense)	16	9,616
activation_1 (Activation)	16	0
dense_2 (Dense)	16	272
activation_2 (Activation)	16	0
dense_3 (Dense)	16	272
activation_3 (Activation)	16	0
dense_4 (Dense)	4	68
activation_4 (Activation)	4	0
Total params.: 10,228		
Trainable params.: 10,228		
Non-trainable params.: 0		

Table 2: The summary of subject functions

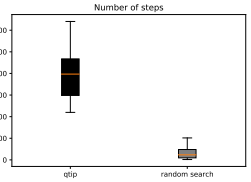
Name	Num. of branches	Purpose
Training	10	train
Greatest Common Divisor (GCD)[2]	8	test
Exponentiation (EXP)[3]	10	test
Remainder	26	test



(a) Branch coverage per 100 episodes during training



(b) Branch coverage of QTIP and random search



(c) Number of evaluations of QTIP and random search

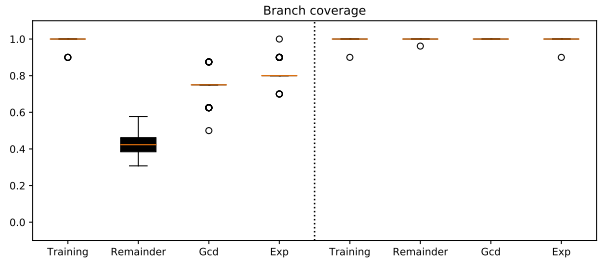
Figure 1: a: The QTIP agent learns to cover more branches in the training function. b: Left is the QTIP and right is the random search. Both methods cover all branches in most cases. c: Compared to random search, QTIP requires more evaluation budgets.

To measure the effectiveness of our approach, we set the random search as our baseline. Figure 1b shows the branch coverage of the training function achieved by QTIP and the random search. In most cases, both methods succeed to cover all branches in the function. However, to achieve the same branch coverage, QTIP requires more evaluation budgets compared to random search, as can be seen in Figure 1c. We suspect both the stochastic behaviour of the agent during the training, as well as the fact that the agent changes the input vector by step size of one only, contributes to the higher cost of QTIP.

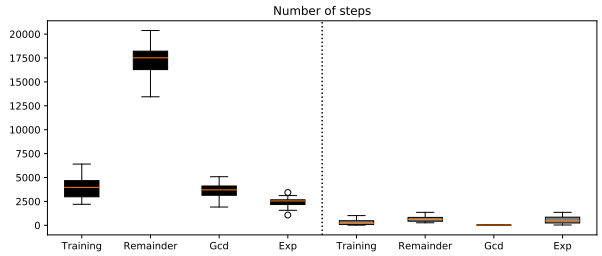
5.2 Unseen Structures

Figure 2a shows the boxplot of branch coverage achieved by the trained QTIP agent against three test functions, over 30 repeated runs. Although QTIP has never seen those functions during the training, it achieves 66.03% branch coverage on average. It suggests that QTIP has learnt a metaheuristic behaviour that works for arbitrary problem instances. Table 3 summarises the results.

In case of the function Remainder, QTIP fails to cover half of the branches. Manual inspection reavealed that all of these uncovered branches have control dependency to other branches (i.e. they are nested). Since we train QTIP agents with branches without any



(a) Branch coverage of QTIP (left) and random search (right)



(b) Number of evaluations of QTIP (left) and random search (right)

Figure 2: a: The left half is the branch coverages of QTIP and the right half is the branch coverage of random search. Although QTIP can cover some arbitrary branches, random search is more effective. b: The left half is the number of evaluations of QTIP and the right half is the number of evaluations of random search. As stated in Section 5.1, QTIP require more evaluation budgets than random search.

depth, the approach level A remains zero during the training. As a result, QTIP does not learn to use the information of approach level which is designed to help to solve nested structures in SUT. We expect that more effective training of QTIP agents in future, with a more diverse set of branch structures, may produce better results for these branches.

Table 3: Average branch coverage (μ) and standard deviation (σ) from QTIP, and Random Search over 30 runs: the highest coverage for each function is typeset in bold.

Function	QTIP		Random	
	μ	σ	μ	σ
Training	99.00	0.03	**99.67**	0.02
Remainder	42.69	0.06	**99.87**	0.01
GCD	73.75	0.09	**100.00**	0.00
Exp	81.67	0.06	**99.67**	0.02
Total	74.28	0.21	**99.80**	0.01

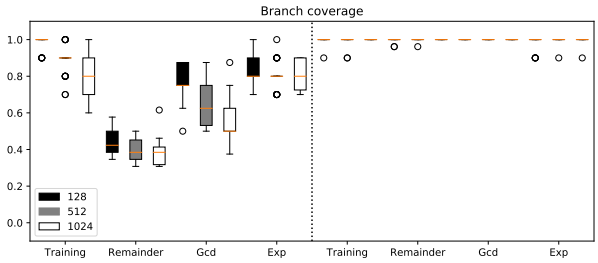
5.3 Unseen Input Ranges

The QTIP is trained with the input range of $[-128, 128]$. We test QTIP with varying input ranges, to check whether QTIP is actually learning the general metaheuristic behaviour, and not overfitting to the smaller input range. Table 4 is the summary of the size of the alternative test input spaces and the corresponding number of steps allowed.

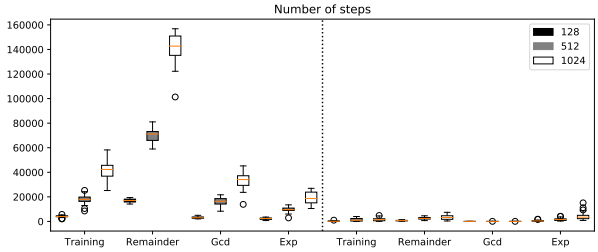
Table 4: The summary of the size of input space and corresponding evaluation budgets

Input range	Number of evaluations
$[-128, 128]$	1,000
$[-512, 512]$	4,000
$[-1024, 1024]$	8,000

Figure 3a shows the change of achieved branch coverage with respect to different sizes of input space. Regardless of the input range, average branch coverage remains consistent. For the input range of $[-512, 512]$ and $[-1024, 1024]$, QTIP achieves branch coverage of 68.12% and 64.15%, respectively. It means that the effectiveness of the QTIP does not depend on the size of the input space. Table 5 summarises the results.



(a) Branch coverage of QTIP (left) and random search (right)



(b) Number of evaluations of QTIP (left) and random search (right)

Figure 3: a: The left half is the branch coverages of QTIP and the right half is the branch coverage of random search. As the size of search space increases, QTIP fails to cover some branches. b: The left half is the number of evaluations of QTIP and the right half is the number of evaluations of random search.

5.4 State Window

Figure 4 shows the change of coverage measured at every 100 episodes. When only the current approach level, A, and the branch distance, Δ, are given as state, the agent learns little. However, if the state includes the previous approach levels and branch distances by the window size of 200, the agent covers more branches as the training continues. Without the window, we observed that the

Table 5: Average branch coverage (μ) and standard deviation (σ) from QTIP, and Random Search over 30 runs: the highest coverage for each function is typeset in bold.

Input range	128				512				1024			
Function	QTIP		Random		QTIP		Random		QTIP		Random	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Training	98.33	0.04	**99.67**	0.02	88.67	0.08	**99.33**	0.02	81.00	0.12	**100.00**	0.00
Remainder	43.97	0.06	**99.74**	0.01	40.13	0.06	**99.87**	0.01	37.95	0.07	**100.00**	0.00
Gcd	77.50	0.10	**100.00**	0.00	63.33	0.10	**100.00**	0.00	56.67	0.12	**100.00**	0.00
Exp	84.00	0.09	**98.67**	0.03	80.33	0.08	**99.67**	0.02	81.00	0.08	**99.67**	0.02
Total	75.95	0.21	99.52	0.02	68.12	0.20	99.72	0.02	64.15	0.21	99.92	0.01

agent tends to repeat only one action for any state. It suggests the possibility that the Q-network is diverging, due to the lack of sufficient reward feedback. Based on this observation, we posit that our assumption about partial observability is valid.

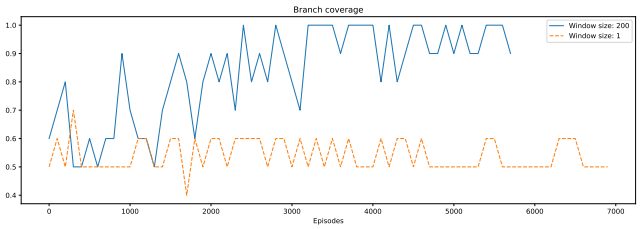


Figure 4: Solid line is QTIP with window size 200 and dashed line with window size 1. The branch coverage is measured at every 100 episodes. Although solid line increases as training continues, dashed line show no improvement at all.

6 THREATS TO VALIDITY

Threats to internal validity include the correctness of the tools used. We trained QTIP using `keras-rl` [18], which is an actively maintained open-source RL library. GUNPOWDER is based on `Clang`, which is the `C` front-end in the widely used `LLVM` framework. Both withstood extensive public scrutiny. GUNPOWDER itself is open sourced for further public inspection.

Threats to external validity include factors that may affect how well the conclusion generalises, such as the size of the empirical study. Although our conclusion may be limited by the size and choice of subject functions, we believe it provides sufficient evidence for the feasibility of formulating SBST as an RL problem. A more through cost-benefit analysis would require a larger empirical study with more target functions, as well as application of different learning algorithm.

7 RELATED WORK

Search based test data generation has been studied for decades now [14]. Korel [8] proposed the idea of using hill climbing for test data generation; later, Xanthakis et al. [23] applied global search algorithms, in particular genetic algorithm. Genetic algorithm is used by one of the most successful search based test data generation technique, EvoSuite, to generate whole test suites for Java [3]. We try to learn the behaviour of local search algorithms using RL: local search heuristics have been shown to be more effective than genetic algorithms for specific targets [4], and has been used to generate dynamic data structures as well as primitive inputs [6, 9].

Reinforcement learning has been shown to be capable of learning sophisticated control policies. Mnih et al. [16] have shown deep neural network is capable of learning human-level control policy directly from high-level observations. Li and Malik [12] proposed the idea of learning optimisation algorithm using RL. Andrychowicz et al. [1] automatically designed optimisation algorithm using a Recurrent Neural Networks (RNNs).

8 CONCLUSION AND FUTURE WORK

We formulate search based test data generation as a decision process to apply reinforcement learning. We also present GUNPOWDER, a general framework for SBST that is compatible to the standard reinforcement learning environment, OpenAI Gym. Using GUNPOWDER, we present a feasibility study of RL-based test data generation with a small empirical study. Our technique, QTIP, achieves 100% branch coverage for the training function, and 60.06% branch coverage for unseen arbitrary functions. The results suggest that learning behaviours of metaheuristic algorithm is feasible.

ACKNOWLEDGEMENT

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIT) (Grant No. R7117-16-0005: A connected private cloud platform for mission critical software test and verification).

REFERENCES

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*. 3981–3989.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. (2016). arXiv:arXiv:1606.01540

[3] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291.

[4] Mark Harman and Philip McMinn. 2010. A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247.

[5] Junhwi Kim, Byeonghyeon You, Minhyuk Kwon, Phil McMinn, and Shin Yoo. 2017. Evaluating CAVM: A New Search-Based Test Data Generation Tool for

C. In *International Symposium on Search Based Software Engineering*. Springer, 143–149.

[6] Junhwi Kim, Byeonghyeon You, Minhyuk Kwon, Phil McMinn, and Shin Yoo. 2017. *Evaluation of CAVM, Austin, and CodeScroll for Test Data Generation for C.* Technical Report CS-TR-2017-413. School of Computing, Korean Advanced Institute of Science and Technology.

[7] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[8] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[9] K. Lakhotia, M. Harman, and H. Gross. 2010. AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems. In *2nd International Symposium on Search Based Software Engineering*. 101–110. https://doi.org/10.1109/SSBSE.2010.21

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[12] Ke Li and Jitendra Malik. 2016. Learning to optimize. *arXiv preprint arXiv:1606.01885* (2016).

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[14] Philip McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (June 2004), 105–156.

[15] Phil McMinn. 2007. *IGUANA: Input Generation Using Automated Novel Algorithms. A Plug and Play Research Tool.* Technical Report CS-07-14. Department of Computer Science, University of Sheffield.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[17] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[18] Matthias Plappert. 2016. keras-rl. https://github.com/matthiasplappert/keras-rl. (2016).

[19] Sebastian Thrun and Anton Schwartz. 1993. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*.

[20] John N Tsitsiklis and Benjamin Van Roy. 1997. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*. 1075–1081.

[21] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning.. In *AAAI*. 2094–2100.

[22] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (01 May 1992), 279–292. https://doi.org/10.1007/BF00992698

[23] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. 1992. Application of genetic algorithms to software testing. In *In 5th International Conference on Software Engineering and its Applications*. Toulouse, France, 625–636.

[24] Shin Yoo. 2011. SBSE As Gaming. In *Proceedings of the 3rd International Symposium on Search-Based Software Engineering*.