# Testing for Fault Diversity in Reinforcement Learning

Quentin Mazouni Simula Research Laboratory Oslo, Norway quentin@simula.no

Arnaud Gotlieb Simula Research Laboratory Oslo, Norway arnaud@simula.no

#### **ABSTRACT**

Reinforcement Learning is the premier technique to approach sequential decision problems, including complex tasks such as driving cars and landing spacecraft. Among the software validation and verification practices, testing for functional fault detection is a convenient way to build trustworthiness in the learned decision model. While recent works seek to maximise the number of detected faults, none consider fault characterisation during the search for more diversity. We argue that policy testing should not find as many failures as possible (e.g., inputs that trigger similar car crashes) but rather aim at revealing as informative and diverse faults as possible in the model. In this paper, we explore the use of quality diversity optimisation to solve the problem of fault diversity in policy testing. Quality diversity (QD) optimisation is a type of evolutionary algorithm to solve hard combinatorial optimisation problems where high-quality diverse solutions are sought. We define and address the underlying challenges of adapting QD optimisation to the test of action policies. Furthermore, we compare classical QD optimisers to state-of-the-art frameworks dedicated to policy testing, both in terms of search efficiency and fault diversity. We show that QD optimisation, while being conceptually simple and generally applicable, finds effectively more diverse faults in the decision model, and conclude that QD-based policy testing is a promising approach.

#### **CCS CONCEPTS**

 Software and its engineering → Software verification and validation; • Computing methodologies → Reinforcement learning.

# **KEYWORDS**

Software Testing, Reinforcement Learning, Quality Diversity

### **ACM Reference Format:**

Quentin Mazouni, Helge Spieker, Arnaud Gotlieb, and Mathieu Acher. 2024. Testing for Fault Diversity in Reinforcement Learning. In 5th ACM/IEEE

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

AST '24, April 15-16, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 979-8-4007-0588-5/24/04...\$15.00 https://doi.org/10.1145/3644032.3644458

Helge Spieker Simula Research Laboratory Oslo, Norway helge@simula.no

Mathieu Acher Univ Rennes, Inria, INSA Rennes, CNRS, IRISA Rennes, France mathieu.acher@irisa.fr

International Conference on Automation of Software Test (AST 2024) (AST '24), April 15-16, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644458

#### INTRODUCTION

In the last decade, Reinforcement Learning (RL) combined with neural networks (NNs) was shown to be able to effectively solve complex sequential decision problems in various fields, such as planning, game playing or system control [10, 21, 28]. Deployment of such action policies in real-world applications involves strong software validation and verification practices. Among them, testing, which trades exhaustiveness for efficiency helps build trust and confidence in the decision model. To that regard, a lot of work to test learnt policies have recently emerged [14, 18, 29, 31]. Some of them [18, 31] further investigate the test results after the search. For example, Pang et al. [18] retrain the policy and Zolfagharian et al. [31] extract interpretable rules from a Decision Tree to characterise the fault-revealing inputs. However, none of them proposes to consider how the policy under test solves (or fails) the problem at each test case and look for diversity. Instead, they all maximise the number of faults found, whatever they reveal or mean.

Still, fault diversity is important: to improve the model, to assess the range of possible incorrect decisions (policy explainability) and one can also see the diversity as a test coverage measure, the latter being especially difficult to assess in NN-based policies. That way, stakeholders could accurately assess the safety of the NN-based policy and eventually build trust and trustworthiness. Fortunately, an evolutionary search technique known as Quality Diversity (QD, or Illumination) addresses this very same issue.

Indeed, QD optimisation finds both diverse and high-quality solutions for a given task. To do so, diversity is not computed in the search space but rather in a behaviour space that describes how a solution actually solves the task. Typical QD applications are in robotics, where the objective is to find the best action policies (i.e., that successfully solve the task, like getting out of a maze) while discovering as many behaviourally different policies as possible, e.g., how the robot leaves the maze.

In this paper, we propose to address the problem of fault diversity in policy testing with Quality Diversity. In other words, we investigate if QD optimisation can find diverse faults in trained policies. Especially, we define and address the underlying challenges of adapting QD optimisation to policy testing. We compare how two existing QD optimisers solve the subsequent policy testing task as a QD problem to a state-of-the-art framework dedicated to policy

testing, both in terms of search efficiency and fault diversity. Our results show that QD-based testing finds diverse solutions, without additional test budget cost. The contributions of this paper are thus:

- We propose the first reformulation of policy testing as a Quality Diversity optimisation problem.
- We implement our method with two classical Illumination algorithms.
- We compare the two QD-based frameworks to a SOTA framework dedicated to policy testing on three use-cases.

The rest of the paper is organised as follows. Section 2 describes the current policy testing techniques and positions our study. Section 3 introduces Reinforcement Learning for decision making and Quality Diversity optimisation. Section 4 presents the challenges to solve the problem of fault diversity in policy testing in the Quality Diversity framework. Section 5 describes the empirical evaluation of our QD-based policy testing framework implemented with 2 QD optimisers to a dedicated policy testing technique. We discuss the current limitation in Section 6. Eventually, Section 7 concludes the paper and draws some perspectives.

#### 2 RELATED WORK

Policy testing has been recently addressed in numerous ways [15]. In the following, we describe the different testing objectives studied alongside the corresponding testing techniques proposed.

Zolfagharian et al. [31] test RL policies with a genetic algorithm [9], where test cases are individuals of a population. Here, the individuals are episodes and their genes, state-action pairs. Starting from historical data (e.g., training data of the policy tested), the search consists of evolving the population to generate faulty episodes which are likely to be consistent with the policy under test (i.e., the trajectories match policy's decision). While this approach lets it avoid executing the policy, it also requires that resulting faulty episodes be validated with respect to the policy. Lu et al. [14] and Ul Haq et al. [29] investigate active policy testing, which consists of dynamically changing the simulator (the policy under test interacts with) during executions. As such, the test cases are defined as the sequences of the environmental changes applied to the simulator. They both turn the search problem into a RL task, where an agent learns to perturb the simulation to generate hazard decisions in the policy. More precisely, Lu et al. [14] address the case of an autonomous driving system (where the possible modelled perturbations include changing weather conditions and dynamics of pedestrians and vehicles - e.g. their position or velocity -), while Ul Haq et al. [29] consider the case of multiple testing requirements (i.e., many-objective search). Tappler et al. [24] look for states that trigger unsafe decisions, called boundary states. The crucial difference with all the other methodologies is that they do not look for those boundary states but, rather, retrieve the latter from the state space explored by an initial backtracking-based, depth-first search for a solution of the decision-making problem. This approach can thus be computationally expansive (depending on the difficulty of the decision task) and provides no guarantee of finding boundary states from the search.

Fuzzing frameworks have also been recently proposed [4, 18, 22]. Pang et al. [18] consider seeds as initial situations of the decisionmaking task. Similarly to genetic searches, the search space is explored by mutating used seeds. Even though the search does not look for diversity, it accounts for novelty by maintaining the pool of seeds which produce uncovered state sequences. Precisely, they compute the likelihood of the latter (collected after each execution) with Gaussian Mixture Models [6] and keep a seed only if the likelihood is lower than a defined threshold. As for [22] and [4], they investigate the bug confirmation problem for NN-based action policies, which corresponds to finding avoidable failures. To bypass the oracle problem in such a situation, Steinmetz et al. [22] use heuristics known in classical AI planning, and Eniser et al. [4] rely on metamorphic testing. To do so, the authors design the metamorphic operations around state relaxation, a well-studied concept also taken from the AI planning community. Their idea is that a relaxed version of a given environment should represent an easier problem than the original one. Therefore, the policy under test contains a bug if it solves the original problem but fails to solve its "relaxed" counterpart. Besides, Tian et al. [25] and Pei et al. [19] consider white-box testing of image input-based NNs. In their work, the objective is to find behaviour inconsistencies, which is approached as a neuron-coverage-guided greedy search. They differ in their test oracles: Tian et al. [25] use metamorphic testing [1] to check that the model tested outputs the same action for morphed images, while Pei et al. [19] rely on differential testing [16] (i.e., several NNs are simultaneously tested and inconsistencies are detected when their decisions differ).

# 3 BACKGROUND

In this section, we introduce Reinforcement Learning to approach sequential decision-making and Quality Diversity optimisation.

# 3.1 Reinforcement Learning for sequential decision-making

Informally, sequential decision-making refers to tasks that can be solved by any decision model in a step-by-step manner and which accounts for the dynamics of the environment [5]. This definition is very broad, and in the following we consider tasks that involve a single decision entity. As such, solving a sequential decision-making problem consists of initialising the world (or environment) to a particular starting situation and letting the decision model (or agent) interact with the former (e.g., simulations) through a step-wise observation-decision-action process until a final state is reached: if the latter is satisfying, the agent has solved the task; otherwise, the agent fails. A typical example is the case of path planning in robotics, where the agent is expected to safely reach a given position from an initial situation. Formally, those kind of problems are formulated as Markov Decision Processes (MDPs), defined as 4-tuples  $\langle S, A, R, T \rangle$  where:

- S is a set of states. Referred to as the observation space, it corresponds to what the agent can know about its environment.
- A is the set of actions. Referred to as the action space, it specifies how the agent acts on its environment.

- R: S × A → ℝ is the reward function. It reflects the agent's
  performance by associating any pair of state-action with a
  numerical value.
- T: S × A × S → [0, 1] is the transition function, which is a
  probability distribution over the observation and the action
  space. It depicts which state the environment will transit to
  after an action is executed. The function is not known by
  the agent and governs the environment's dynamic.

Solutions to MDPs are called policies and noted  $\pi$ , which are functions mapping from the observation space S to the action space A. A common approach to training policies is Reinforcement Learning, a sub-field of Machine Learning which consists in learning from rewards/penalties [23]. Precisely, RL learns an optimal, or nearly-optimal, policy  $\pi: S \times A \mapsto [0,1]$  that maximises the total expected discounted cumulative reward  $R_t = \sum_{t>0} \gamma^{t-1} r_t$ , where  $0 < \gamma \le 1$  is the discount factor. This parameter controls how the agent takes into account the future rewards. Precisely, a small value encourages the agent to maximise short-term rewards, whereas high values (usually close to 1) lead the agent to focus on maximising long-term rewards.

In this work, we consider black-box testing, i.e., without access to the internals of the policy nor the simulator, of *deterministic* decision models, changing thus the previous definition to  $\pi: S \mapsto A$ . In the following, we introduce Quality Diversity optimisation.

# 3.2 Quality Diversity

Informally, Quality Diversity optimisation stems from the evolutionary algorithms, but provides a shift in methodology by not only considering the maximisation of a fitness function, but explicitly targeting the discovery of diverse solutions as characterised by their behaviour, i.e. the way a problem is solved. Formally, it assumes the objective function f now returns a fitness value  $f_x$  and a behavioural descriptor  $b_x$  for any parameter x, i.e.,  $f_x, b_x \leftarrow f(x)$ . As previously mentioned, the behavioural descriptor describes how the solution solves the problem, while the fitness value quantifies how well it is solved. If we assume that we want to maximise the fitness function and define the behaviour space as B, then the goal of QD optimisation is to find for each point  $b \in B$  the parameters  $x \in X$  with the maximum fitness value:

$$\forall b \in B, x^* = \arg\max_{x} f_x \mid b_x = b$$

The goal of QD is thus to return a collection of behaviourally distinct solutions, whose individuals are the best performers (also called *elites*) in their local behaviour area (or behaviour *niches*).

The first method in the QD paradigm is *Novelty Search* (NS) [11, 12], which entirely abandons the search for the objective function and only explores behavioural novelty. While later developments in QD reconsider the inclusion of the objective function into the search by competition of behaviourally similar solutions with different quality [7, 13], we see a parallel between the pure search for novelty and fault-triggering inputs for policies. In the search for fault-triggering inputs, there is no clear objective function to maximise besides the binary test verdict of a successful or failing episode. There is no indication of the closeness to a failure, even though some existing works introduce surrogate objectives to help guide the search [31]. We therefore consider novelty search as a

potentially interesting approach to explore the behaviour space without the necessity for guidance through an objective function.

In its recent variants, QD tends to discover as many diverse behaviours as possible, while improving their elite, i.e., the individual with the highest quality in that niche. The result of QD optimisation is a set of solutions, which is usually referred to as *collection*, *archive* or *container* and is structured through the niches of the behaviour space. During the optimisation process, the collection is filled with any candidate whose behaviour is novel enough and lets the latter compete with the current collection's elite if its behaviour is deemed to belong in a niche already covered. As such, the collection of any QD algorithm structures its search, since it defines the behavioural neighbourhood of the parameters evaluated (i.e., how to consider if two solutions have close enough behaviours to belong to the same behavioural area or niche).

In the QD literature, one can distinguish two types of collections, namely structured and unstructured ones. In the former case, the behaviour space is discretised with a segmentation pattern into a grid, and each cell represents a niche (or behaviour descriptor location). This approach is for example implemented in the MAP-Elites algorithm [17], one of the most used QD optimisers. Here, the collection is a regular grid and the search aims at filling every cell of that grid with the best possible solution. On the other hand, unstructured archives do not define niche locations before the optimisation starts and relies on distance thresholds and/or local densities to assess the behavioural similarities and neighbourhoods between solutions.

Cully and Demiris [2] propose a unified definition of QD optimisation that we follow to introduce the general, high-level optimisation process in QD (see Algorithm 1). In their formulation, algorithms vary depending on (i) the type of container (i.e., how the data is gathered and ordered into a collection); (ii) the selection operator (i.e., how the solutions to be modified in the next generation are selected) and (iii) the type of scores computed for the container and the selection operator to work. Given those parameters and operators, the execution of a QD algorithm follows a 4-step loop until the budget is consumed:

- A new batch of candidates is produced from the individuals selected by the selection operator.
- The candidates are evaluated and their performance and descriptors, recorded.
- Each candidate is possibly added to the container (according to the solutions already in the collection).
- The scores maintained by the container are eventually updated (if needed). Common scores are the novelty, the local competition, or the curiosity score.

For more information about the different variants studied and the most widely used scores, see [2].

#### 4 QD OPTIMISATION FOR POLICY TESTING

This section describes the challenges in optimising policy testing for fault diversity with Quality Diversity and how we address them. We consider a black-box setting, where neither the MDP nor the policy under test  $\pi$  can be directly inspected, but only inputs and outputs can be observed. The only assumption we take is, that it

#### Algorithm 1 4-step QD-algorithm for Policy Testing

```
Input: N: iteration budget, N_{init}: number of initial iterations
Output: A: archive of diverse and high-performing solutions
 1: A \leftarrow \emptyset
                                              ▶ empty archive of solutions
 2: for I \leftarrow 0 to N do
       ▶ start with random parents and offspring
 3:
       if I < N_{init} then
 4:
          X_{parents} \leftarrow \text{random\_solutions}()
 5:
          X_{offspring} \leftarrow \text{random\_solutions()}
 6:
 7:
          ▶ 1. select individuals from the archive and/or the previous
 8:
          X_{parents} \leftarrow select(A, X_{offspring})
 9:
          \triangleright 2. create randomly modified copies of X_{parents} (mutation
10:
          and/or crossover)
          X_{offspring} \leftarrow variation(X_{parents})
11:
12:
13:
       for x \in X_{offspring} do
          ▶ 3. record the behaviour and quality of the candidate ▷
14:
          b_x, f_x \leftarrow f(x)
15:
          x.store\_score(b_x, f_x)
16:
          \triangleright 3*. initialise MDP with x and characterise x with (1) the be-
17:
          haviour and fitness of \pi; (2) the test oracle result
          b_x, f_x, o_x \leftarrow Evaluate(MDP, \pi, x)
18:
          x.store\_score(b_x, f_x, o_x)
19:
          ▶ 4. attempt to add the candidate to the archive (local
20:
          competition)
          attempt_to_add(x, A)
21:
          update\_scores(parent(x), A) \triangleright parent's scores might be up-
22:
23:
       end for
       ▶ possibly update the scores of all solutions (e.g., curiosity scores) ▷
24:
       update_scores(A)
25:
26: end for
```

is possible to instrument the MDP's simulator with an initial state and random seed.

27: **return** *A* 

Solution Behaviour. Since we aim to find fault-triggering initial states of the MDP (under which  $\pi$  fails), the search space corresponds to the parameter (or input) space of the MDP's simulator. The exact definition of such parameter spaces depends on the implementation of the simulator and the decision-making problem (see Section 5). Therefore, at first glance, the search space does not come with any behavioural definition directly: parameters like gravity or objects' positions do not behave in a specific way. This is not an issue though, as we can still use the traditional behaviour space definition of QD. Indeed, we similarly want to characterise how the policy under test solves the MDP. The main difference is that in QD, the solutions evaluated are policy's parameters (since the goal is to find good policies) whereas in our reformulation,  $\pi$  is the same and its behaviours depends on the initial scenario described by the solution evaluated. This allows us to leverage the QD's literature richness of behavioural policy analysis and use already proposed behaviour space definitions (see Section 5) to describe the solutions. That way, finding diverse solutions will exercise as many different policy behaviours as possible and, similarly, the fault-triggering ones will imply diverse hazard decisions.

Solution Quality. The second challenge in the adaption of QD for policy testing lies in the definition of the solutions' quality (or fitness). Indeed, in policy testing the quality of a test case (i.e., the evaluation of an execution) boils down to the boolean value of the test oracle (i.e.,  $\pi$  solves the task or fails), which is not enough informative. We instead define the quality of solutions as the accumulated reward of the policy under test (like in common QD applications, i.e., as if we were finding policies) and define the optimisation task of QD with minimisation (i.e., accumulated reward minimisation).

Assumptions. In this first work, we fix the randomness effects during the simulation, letting thus the *MDP*, its simulator and all their executions be deterministic. As such, every input test (i.e., solution) generates a single trajectory and thus, a single behaviour (and test result).

QD-based Policy Testing. Thanks to the flexibility of Quality Diversity, our proposal to optimise a fault-triggering simulator's inputs search for policy testing with QD involves only few changes to the high-level framework proposed by Cully and Demiris [2] as shown in Algorithm 1. The modifications are highlighted at lines 18-19, which mostly consists of changing how solutions are evaluated. First, a solution x is used to initialise the MDP's simulator. Then, we let  $\pi$  solve/fail the decision task and characterise s with (1) the behaviour and fitness of  $\pi$ ; (2) the test oracle result. After the search, fault-triggering solutions (faults for short) can be retrieved from the archive by filtering it with the test oracle results.

#### 5 EXPERIMENTAL EVALUATION

#### 5.1 Research Questions

To evaluate QD optimisation for fault diversity in policy testing, we conduct experiments to answer three research questions (RQs):

- **RQ1** How efficient is QD optimisation compared to dedicated policy testing techniques?
- **RQ2** How does QD optimisation improve diversity?
- **RQ3** How does the behaviour space definition impact the performance of QD-based policy testing?

Answering the first two research questions will let us assess the benefits and cost of prioritising diversity. Here, we compare the number of faults revealed and their diversity among QD-based testing and testing without consideration of behaviours. Finally, the last research question investigates the impact of the definition of the behaviour space, a key configuration parameter for QD optimisation. We expect that the number of faults found by QD-based testing still be competitive and that it improves fault diversity by accounting for the behaviour of the policy under test.

# 5.2 Experiments

To answer the RQs, we conduct experiments with three standard environments [27]: Lunar Lander, Bipedal Walker and Taxi. We compare Random Testing with MDPFuzz [18], a recent black-box policy testing technique for MDPs and two implementations of

our QD-based testing framework. The first one uses the QD optimiser MAP-Elites [17], and the second one, Novelty Search [11]. Random Testing will assess the difficulty of the testing task and act as a baseline to compare the other methodologies. We choose the policy-dedicated testing framework MDPFuzz since it addresses complex environments (which has not been shown by other frameworks such as STARLA [31]) and drives its fuzzing search towards uncovered state sequences, accounting thus for diversity. Finally, as part of our QD-based testing framework, we study MAP-Elites (ME) since it is one of the first and most conceptually simple illumination algorithms, while Novelty Search (NS) will let us assess the relevance of accounting for the quality of the executions, as this algorithm emphasises diversity only.

*5.2.1 Environments.* The three selected environments are commonly used benchmarks in the RL literature.

Lunar Lander. This control problem consists in safely landing a spacecraft. We chose this environment since it has been used in QD optimisation and RL. In particular, a behaviour space has already been studied [26]. The spacecraft always starts at the top centre of the space and, similarly, the landing pad is always at the centre of the ground. The initial situations differ in the shape of the landscape (around the landing pad) and the initial force applied to the spacecraft. The policy controls the main and orientation engines of the spacecraft. Precisely, there are four possible actions: do nothing, fire the left orientation engine, fire the main engine and fire the right orientation engine.

Bipedal Walker. This problem consists in piloting a 4-joint walker robot across an uneven landscape composed of obstacles like steps, pits, and stumps. We chose this environment to follow the evaluation of MDPFuzz [18] (enabling thus a fair comparison), but also because behaviour spaces have already been proposed [8]. The impact of these definitions on the results are studied in RQ3. In this problem, the walker always starts at the same position. The initial situations differ in the shape of the landscape (positions of the steps, the stumps and the pits). The action space is continuous. Precisely, the action of the policy is the motor speed values at the 4 joints of the robot, which are localised at the hip and its knees.

Taxi. In this classical environment, the policy navigates in a grid world to pick up passengers and safely drop them off at their destinations [3]. Every test initiates a particular initial situation as the position of the passenger, the position of the taxi and the passenger's destination. The six actions possible by this policy are the next taxi's direction (going north, south, east or west) and interactions with the passenger (pick it up, drop it off). We use a version of the Taxi environment with an enlarged 18x13 map, thus disabling the simple enumeration of all MDP's possible states for the standard 5x5 grid.

#### 5.3 Metrics

To answer RQ1, we measure what we call the *test efficiency* as the number of distinct faults found over time. To answer RQ2, we study the diversity of testing (i.e., how a test methodology exercises the policy under test) and diversity of the faults. We consider two metrics to measure diversity. First, we compare the behaviour coverage,

that is: how many behaviours are discovered during testing. To do so, we follow the QD literature and count the number of bins filled in each result archive. This archive corresponds to the regular grid used by MAP-Elites. For fault diversity, only the bins filled by at least one fault-triggering solution are counted. To complement this method space-based measure, we also analyse the diversity with the final states of the simulations. The idea of the final state comparison is to make the result analysis possibly more accessible and not only do a comparison in the method space (i.e., behaviour space). Indeed, the definition of the behaviour space is domain-dependent and can therefore vary. Complementing thus the behaviour space coverage with final state diversity analysis will provide us with conclusions that are not bound to how behaviours are actually computed from the trajectories. We report final state diversity as the average distances of the 3 nearest neighbours in the solution sets, since this metric captures the sparseness of a data set (i.e., if the set is composed of dense points or if the points are smoothly distributed). Similarly as the first metric, fault diversity only considers the final states of fault-revealing executions (failure states). Eventually, to answer RQ3, we compare the effect of four behaviour spaces for the Bipedal Walker use-case on all the previous metrics mentioned above.

# 5.4 Implementation

We run all methods with a budget of 5000 tests, and an initialisation phase of 1000 for MAP-Elites and MDPFuzz (following the evaluation in [18]). For Novelty Search, we use a population size of 100 and let the search iterate for 50 iterations. The result archives used to collect data rely on a regular grid of 50x50 bins. All the experiments were executed on a Linux machine (Ubuntu 22.04.3 LTS) equipped with an AMD Ryzen 9 3950X 16-Core processor and 32GB of RAM. We accounted for randomness effects by repeating all the experiments with 10 seeds and report the median results as well as the first and third quantiles. The source code of the experiments is available online 1.

Test oracles. In Lunar Lander, a failure occurs if the lander crashes into the ground or moves outside the viewport. In Bipedal Walker, failure occurs if the body of the robot collides with the ground. For the Taxi environment, a fault occurs in case of an illegal action (for instance, dropping the passenger even though the taxi is still empty) or collision (by moving into a wall).

Input/Parameter sampling and mutation. For the Lunar Lander environment, the shape of the landscape is fixed by our assumption to consider deterministic environments. As such, the parameter space is the two-dimensional space  $\left[-1000,1000\right]^2$  that describes the possible initial forces applied to the spacecraft. The mutation operator slightly perturbs the original parameter (clipped, if needed). For the Bipedal Walker use-case, we follow the experimental settings of MDPFuzz [18] for both the parameter space and the mutation operator. Precisely, the parameter space encodes the type of obstacles (flat, pit, steps or stump) of the landscape as 15-size vectors whose values  $\in [0,1,2,3]$ , while the mutation operator randomly changes at least one of those values (clipped if needed). As mentioned in the aforementioned paper, the obstacles are sufficiently well away from

 $<sup>^1</sup>https://github.com/QuentinMaz/QD\_Based\_Testing\_RL$ 

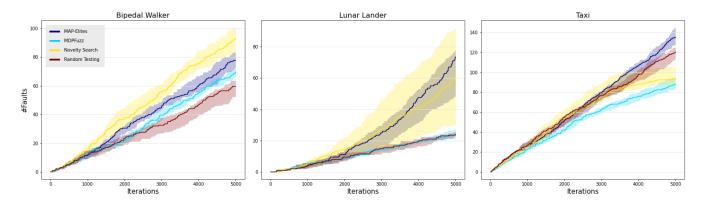


Figure 1: Evolution of the number of fault-triggering solutions found for each framework evaluated. The lines show the median results over 10 executions, and the shaded areas correspond to the first and third quantiles.

each other so that they can be passed by an optimal policy  $\pi^*$  (i.e., all the problems are solvable). As for the last use-case, the parameter space encodes the initial position of the taxi and the passenger as well as its destination. The map is static in this environment. The mutation operator increments or decrements one value (clipped if needed). For instance, the taxi would now start from a cell in the grid world close to the initial one.

Behaviour Spaces. All the behaviour spaces studied are twodimensional spaces. For the Lunar Lander use-case, we use the behavioural definition proposed by [26]. It describes how the policy lands the spacecraft as its horizontal position and vertical velocity when it first touches the ground. If the lander moves outside the viewport, we consider the last values observed. For Bipedal Walker, we follow a previous work that studies policy behaviour for this very same use-case [8]. In particular, they define a set of hand-designed behaviour descriptors (averaged over the observation state sequence) such as Distance, the walker's position relative to the goal, Hull angle, the body's angle of the agent, Torque, the force applied to the agent's hip and knee joints, Jump, that describes when both legs are simultaneously not in contact with the ground and Hip angle/speed, that are the angle and speed values of the agent's hip joints, respectively. We define the behaviour space as the pair of the descriptor Distance and Hull angle. The effect of different descriptor pairs as behaviour spaces is addressed in RQ3 (see Subsection 5.7). As for the last use-case, the behaviour is defined as the two dimensional point that counts the number of actions to 1) pick-up the passenger and 2) to drop them off afterwards.

Models Under Test. For the Bipedal Walker and Lunar Lander experiments, we use the models freely available on the Stable-Baselines3 repository [20]. For the customised Taxi use-case, we train the agent via Q-Learning [30].

*Hyperparameters.* We follow the guidelines indicated in [18] to configure the Gaussian Mixture Models used by MDPFuzz. As for Novelty Search, it computes the novelty scores as the average distance of the 3 nearest neighbours and the novelty threshold for updating its novelty archive has been set to t = 0.9 for the Taxi usecase and t = 0.005 for Bipedal Walker and Lunar Lander. Similarly

to all the previously mentioned experimental parameters, they have been obtained in a prior study.

# 5.5 RQ1: Effective Fault Detection

We first investigate the efficiency of the approaches evaluated. Figure 1 shows the evolution of the number of distinct fault-triggering solutions.

Results. All the frameworks evaluated find more faults than Random Testing for the Bipedal Walker use-case. Precisely, at the end of testing, we report an average improvement of 30% and 56% for ME and NS, respectively, while MDPFuzz finds 17% more faults. Greater improvements of QD optimisation are observed for the Lunar Lander use-case (up to 206% and 152%), while MDPFuzz matches the baseline. We note that in the case of NS, the latter vary a lot (see the shaded areas in Figure 1). The results for the last use-case show a different picture though. Here, only ME beats Random Testing (up to 12%), while Novelty Search and MDPFuzz find 23% and 27% fewer faults, respectively.

Analysis. Searching for diversity can impede test efficiency, as Random Testing outperforms Novelty Search in the Taxi experiments. Furthermore, NS is the most sensitive framework to randomness, which is likely to be caused by its high dependency to its initial population. It is therefore difficult to recommend as a general optimiser for efficient QD-based policy testing. However, accounting for both quality and diversity lets MAP-Elites systematically beat Random Testing, while showing lesser sensitivity to its initialisation. Besides, MDPFuzz does not seem to be suited for all the use-cases studied, since it is only able to compete with the efficiency of QD-based policy testing on the Bipedal Walker environment. While being deceptive, those results are not completely surprising. Indeed, this framework drives its search with Gaussian Mixture Models (GMMs), whose parameters were studied for several applications, including Bipedal Walker. We therefore suspect that MDPFuzz ends up sharing its results with Random Testing on Lunar Lander because of none-optimal GMM configuration. As for the Taxi use-case, we recall that the MDPFuzz aims at enabling policy testing solving complex MDPs [18], which is definitely not the case of this toy problem.

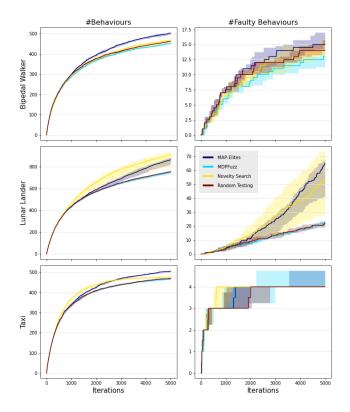


Figure 2: Evolution of the behaviour space coverage over time as the number of behaviour niches (bins) illuminated during testing. In the second column, only bins filled by faulttriggering solutions are counted, i.e., faulty behaviours. The lines show the median results over 10 executions, and the shaded areas correspond to the first and third quantiles.

Conclusion. The complexity of dedicated policy testing techniques such as MDPFuzz can hurt their efficiency – especially for smaller MDPs –, while QD optimisation consistently finds the greatest number of faults in the decision model. About that, discarding solution quality in favour of novelty can lead to better, yet unstable results. QD-based policy testing does not come with poorer efficiency (as we expected) but rather with significant increase the number of functional faults found in the model.

# 5.6 RQ2: Testing and Fault Diversity

Next, we study how QD optimisation improves diversity in terms of behaviours and final states. Figure 2 shows the behaviour and faulty behaviour coverage, and Figure 3 the average distances of the 3 nearest neighbours of the final states. Since the latter are distances between observation states, Figure 3 shows the relative performance of the methodologies evaluated to the Random Testing baseline.

Results. ME systematically improves behaviour discovery, ranging from 7% (Taxi) to 14% (Lunar Lander). MDPFuzz matches RT's performance for all the use-cases, as Novelty Search does, except for Lunar Lander for which NS covers up to 20% more behaviour

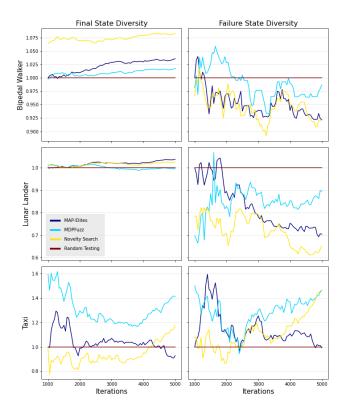


Figure 3: Final state diversity as the average distances of the 3 nearest neighbours. Since their scale depends on the observation space of each use-case, we report the relative performance of the methodologies to Random Testing. The lines show the median results over 10 executions.

niches (slightly outperforming MAP-Elites). Regarding faulty behaviours, our results only show significant differences to Random Testing for MAP-Elites and Novelty Search on the Lunar Lander experiments. Precisely, they stand out after 2000 iterations and end up with 189% and 138% more faulty behaviours discovered, respectively. Similarly to the previous results though, we note that NS's performance vary a lot. If we now look at final state diversity, we find mixed-bag results. For Bipedal Walker, we observe that Novelty Search explores around 7.5% more diverse final states than the random baseline throughout testing. For Lunar Lander, none of the testing technique significantly improves the baseline's results. Actually, the failure state distribution tends to be worse, especially for QD optimisation (up to a 30% decrease). For the last use-case, MDPFuzz outperforms all the other techniques with significant margins, with 40% greater distances in the final states (averaged throughout testing) and up to 50% sparser failure states. Meanwhile, if QD optimisation does not show significant difference with the baseline for the final state diversity, we observe great, yet unstable improvements in their failure state results (as shown in the second chart of the bottom row in Figure 3). In particular, Novelty Search catches up with MDPFuzz's performance on the last iterations.

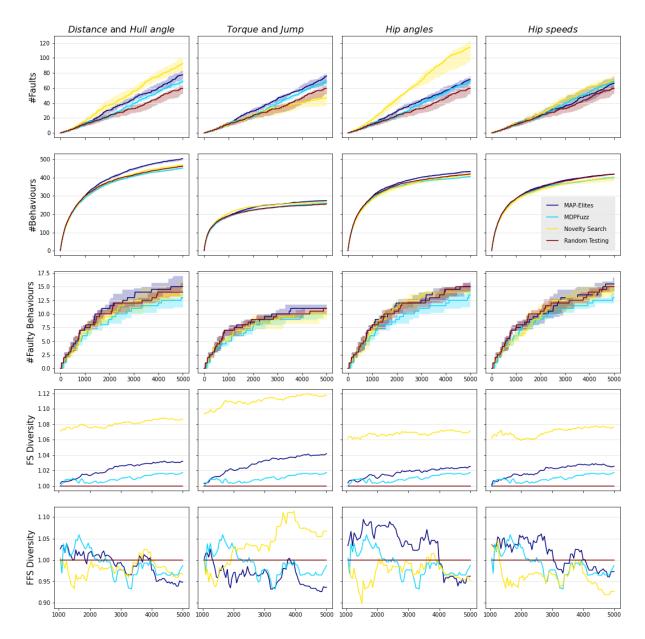


Figure 4: Impact of the behaviour space parameter for the Bipedal Walker experiments. The four spaces are different pairs of hand-designed descriptors studied in [8]. Each column displays the results for a behaviour space. From top to bottom: number of faults, number of behaviours and faulty behaviours, final and failure state diversity (FS and FFS) relative to Random Testing. The results are the medians of 10 executions.

Analysis. QD-based policy testing discovers more behaviours than both MDPFuzz and Random Testing. More importantly, in some cases (Lunar Lander), this also applies for fault-triggering solutions. In other words, QD-based policy testing finds more diverse faults in the model under test than dedicated testing techniques. It is interesting to note that the SOTA framework MDPFuzz does not cover less behaviours than Random Testing; something that one could have expected. Complementing the diversity measurement of our behavioural evaluation with final state analysis lets us shade

interesting details. Indeed, discovering more behaviours does not translate to sparser final state distribution, as the two QD-based frameworks either matches or slightly improves Random Testing. It is especially the case for Lunar Lander, for which both ME and NS significantly improve faulty behaviours coverage (i.e., find diverse faults in the model tested) while showing poor failure state distributions. As such, the use of behaviour spaces helps policy testing, as simply looking at final states is not enough to cover diverse faults in the model. If we consider the actual descriptors

used to define the behaviour spaces, we can explain the lack of a general correlation between increased behaviour and final state discovery. Indeed, none of the descriptors solely depends on the termination of an execution to define the behaviour. For instance, in the Bipedal Walker experiments, the behaviours are averages of some observations' features.

Interestingly though, the descriptors defined for the Lunar Lander use-case let us better understand what type of faults every methodology tends to find. In particular, behaviours are based on the *first* contact of the spacecraft with the ground; but the policy can later fail if the landing wasn't on the targeted pad (and that the lander has thus to be safely glided towards the latter). Therefore, by facing the results of our bi-metric evaluation, an interpretation is that here the QD-based frameworks find solutions for which  $\pi$  lands the spacecraft at various positions (i.e., diverse behaviours) but later fails the task by colliding the lander's body on the same edge of the landscape (i.e., dense failure states), while MDPFuzz either finds slightly more different edges (around 10% more distributed failure states) but the policy actually lands at similar positions (i.e., poor faulty behaviour discovery).

Conclusion. The ability of QD optimisation to find diverse highquality solutions applies for policy testing, that is, fault-triggering inputs that exercise the model such that it fails with varied behaviours. As found in the previous analysis, discarding solution quality leads to unstable results, which is fixed by more balanced Quality Diversity optimisers such as MAP-Elites.

# 5.7 RQ3: Behaviour Space Impact

In this last research question, we investigate the effect of different behaviour spaces on the proposed QD-based policy testing framework. As previously mentioned, Gupta et al. [8] define several descriptors to characterise the behaviour of a policy. We use the ones introduced in Subsection 5.4 to define three additional behaviour spaces (as descriptor pairs) to study how the results of our appraoch can differ. Figure 4 summarises our findings, where each column corresponds to a behaviour space. Note that the first column corresponds to the results found above. In the following, we analyse the impact on each metric.

Test efficiency. ME improves the number of faults found compared to Random Testing, regardless of the behaviour space used, to a extent that can decrease to 11.5% (down from 30%, the first column on Figure 4). However, Novelty Search shows significantly higher sensitivity to the behaviour space. In particular, NS can almost double the number of faults found by Random Testing with the third behaviour definition (90%), but has around 20% poorer performance with an inappropriate space (as shown in the second column of Figure 4). The drastic difference in behaviour sensitivity of the two implementations again lies in the balance between how quality and diversity of solutions are accounted. Since MAP-Elites mostly relies on quality (by focusing on mutating its elites), the results are steadier and less sensitive to their exact behaviour. In other words, as long as the behaviour space is able to capture diversity - which is the case here, as all the spaces are based on meaningful, hand-designed descriptors - MAP-Elites seems to be an efficient optimiser for QD-based policy testing. On the other

hand, by discarding quality to only account for behaviour novelty, Novelty Search becomes more sensitive to the behaviour space used.

Behaviour Diversity. The number of behaviours discovered by all the methodologies are for the most part steady across the spaces evaluated, which match the performance of the baseline. We only observe ME discovers around 8% more behaviours with two of the spaces (first and third columns in Figure 4). As for the diversity of faulty behaviours, we report no significant change compared to our previous findings, that is: all frameworks share similar figures. One can possibly note MDPFuzz has consistent lower numbers (10% in averaged over the spaces). While QD-based policy testing never impedes behaviour discovery, its ability to significantly improve the random baseline depends on the behaviour space. For instance, we find that the behaviour of the policy tested is best captured with the first pair of descriptors.

Final State Diversity. The behaviour space definition does not affect the sparseness of the final states, which is little surprising since behaviours are computed as feature averages. Similarly, the relative performance of the methodologies to the baseline does not significantly fluctuate. In particular, Novelty Search shows a 8% to 12% smoother state distribution. Finally, if we only consider failure states (bottom row), we can see that there is no general trend in the results neither, with minor fluctuations around the baseline lower than 11% throughout testing.

#### 5.8 Summary

Our first and most important conclusion is that QD manages to find diverse faults in the decision model, despite the simplicity of our approach. In particular, it shows that complex, dedicated policy techniques such MDPFuzz are not always needed. Second, we systematically observe a lack of consistency in the results of our framework optimised with Novelty Search, especially when running close to our set test budget. As such, despite its outstanding performances for some cases, we recommend using our proposal with QD optimisers that account for quality and diversity in more balanced ways. Finally, the selection of the precise behaviour space can boost the performance of QD-based policy. We observe this for the Bipedal Walker use-case, where a well-chosen behaviour space substantially increases the number of faults detected. At the same time, the other behaviour spaces are all competitive and do not negatively impact the ability to reveal diverse faults.

# 6 THREATS TO VALIDITY

In the following, we discuss the limitations of our proposal as well as the threats of our experimental evaluation.

External Threats. A first threat to our evaluation is the policy under test used, that we mitigate by using the same model as previous works [18]. Similarly, there is some inherent bias to the results from the selected use-cases. To that regard, we consider three environments of various natures, two of them having already been studied in the QD and RL literature. Finally, we only evaluate one SOTA policy testing technique, namely MDPFuzz. Given the space limitation, we decided to prioritise the number of use-cases, since

this work primarily introduces Illumination optimisation in policy testing.

Construction Threats. In this first work, we assume to have deterministic executions with fixed randomness effects. While deterministic executions are a common approach for testing policies and their deployment, the challenge of handling MDP stochasticity lies in the fact that multiple executions of a particular solution would generate different trajectories and thus, possibly different behaviours. Similarly, some of these executions might fail depending on the selected policy actions or state transitions in the MDP. We acknowledge that it is thus an important challenge in QD-based policy testing, that needs to be addressed in future work. Practically speaking, by fixing the random effects we reduce the search space to a subset of inputs that reveal a fault with the given random seed. We thereby limit the experimental evaluation to a subset of all possibly detectable faults in the policy.

#### 7 CONCLUSION

This work introduces Quality Diversity (QD) for policy testing. QD is a flexible, black-box optimisation framework that optimises a population of individuals by considering both their behaviour, i.e. how they solve a given problem, and their quality, i.e. how well they solve it.

We illustrate how to adapt QD to policy testing and propose the first formulation of diversity-oriented policy testing as an Illumination task. Precisely, we characterise test inputs with the behaviour of the policy under test, that is, how it solves/fails the test case. We implement our QD-based testing framework with two commonly used QD optimisers of different paradigms: the elitist MAP-Elites and the divergent Novelty Search algorithms. We perform experiments on three use-cases from the reinforcement learning literature, and compare QD-based testing to state-of-the-art policy testing and random testing as the state-of-the-practice. Our results show that QD optimisation, while being a conceptually straightforward and easy-to-apply approach, not only improves fault diversity but also fault detection. We further assess the impact of the behaviour space definition, what we consider as the most decisive parameter of our approach. With this first work, we open a new application area for Quality Diversity.

In future work we will address the inclusion of generic respectively learned behaviour spaces to reduce the initial effort to setup QD and the handling of stochastic MDPs in the search space to further guide the search.

# **ACKNOWLEDGEMENTS**

This work is funded by the Norwegian Ministry of Education and Research, the Research Council of Norway under grant number 324674 (AutoCSP), and is part of the RESIST\_EA Inria-Simula associate team.

#### **REFERENCES**

- T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report. Department of Computer Science, Hong Kong University of Science and Technology.
- [2] Antoine Cully and Yiannis Demiris. 2018. Quality and Diversity Optimization: A Unifying Modular Framework. IEEE Transactions on Evolutionary Computation 22, 2 (2018), 245–259. https://doi.org/10.1109/TEVC.2017.2704781

- [3] Thomas G Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of artificial intelligence research 13 (2000), 227–303
- [4] Hasan Ferit Eniser, Timo P. Gros, Valentin Wüstholz, Jörg Hoffmann, and Maria Christakis. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. https: //doi.org/10.1145/3533767.3534392
- [5] Keith Frankish and William M. Ramsey (Eds.). 2014. The Cambridge Handbook of Artificial Intelligence. Cambridge University Press, Cambridge, UK.
- [6] D. N. Geary. 2018. Mixture Models: Inference and Applications to Clustering. Journal of the Royal Statistical Society Series A: Statistics in Society 152, 1 (12 2018), 126–127. https://doi.org/10.2307/2982840 arXiv:https://academic.oup.com/jrsssa/article-pdf/152/1/126/49758778/jrsssa\_152\_1\_126.pdf
- [7] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. 2015. Devising Effective Novelty Search Algorithms: A Comprehensive Empirical Study. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (Madrid, Spain) (GECCO '15). Association for Computing Machinery, New York, NY, USA, 943–950. https://doi.org/10.1145/2739480.2754736
- [8] Vikas Gupta, Nathanael Aubert-Kato, and Leo Cazenille. 2020. Exploring the BipedalWalker Benchmark with MAP-Elites and Curiosity-Driven A3C. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (Cancún, Mexico) (GECCO '20). Association for Computing Machinery, New York, NY, USA, 79–80. https://doi.org/10.1145/3377929.3389921
- [9] John H. Holland. 1992. Genetic Algorithms. Scientific American (1992).
- [10] Rushang Karia and Siddharth Srivastava. 2020. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. CoRR (2020).
- [11] Joel Lehman and Kenneth O. Stanley. 2008. Exploiting Open-Endedness to Solve Problems Through the Search for Novelty. In *IEEE Symposium on Artificial Life*. https://api.semanticscholar.org/CorpusID:2367605
- [12] Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. Evolutionary computation 19, 2 (2011), 189–223.
- [13] Joel Lehman and Kenneth O. Stanley. 2011. Evolving a Diversity of Virtual Creatures through Novelty Search and Local Competition. In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (Dublin, Ireland) (GECCO '11). Association for Computing Machinery, New York, NY, USA, 211–218. https://doi.org/10.1145/2001576.2001606
- [14] Chengjie Lu, Yize Shi, Huihui Zhang, Man Zhang, Tiexin Wang, Tao Yue, and Shaukat Ali. 2023. Learning Configurations of Operating Environment of Autonomous Vehicles to Maximize their Collisions. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2022.3150788
- [15] Quentin Mazouni, Helge Spieker, Arnaud Gotlieb, and Mathieu Acher. 2023. A Review of Validation and Verification of Neural Network-based Policies for Sequential Decision Making. In Rencontres des Jeunes Chercheurs en Intelligence Artificielle (RJCIA). https://pfia23.icube.unistra.fr/conferences/rjcia/Actes/RJCIA2023\_paper\_5.pdf
- [16] William M. McKeeman. 1998. Differential Testing for Software. Digit. Tech. J. (1998).
- [17] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs.AI]
- [18] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2022. MDPFuzz: Testing Models Solving Markov Decision Processes. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. https://doi.org/10. 1145/3533767.3534388
- [19] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In proceedings of the 26th Symposium on Operating Systems Principles. 1–18.
- [20] Antonin Raffin. 2020. RL Baselines Zoo. https://github.com/DLR-RM/rl-baselines Zoo.
- [21] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science (2018). https://doi.org/10.1126/science.aar6404
- [22] Marcel Steinmetz, Daniel Fiser, Hasan Ferit Eniser, Patrick Ferber, Timo P. Gros, Philippe Heim, Daniel Höller, Xandra Schuler, Valentin Wüstholz, Maria Christakis, and Jörg Hoffmann. 2022. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. Proceedings of the International Conference on Automated Planning and Scheduling (2022). https://doi.org/10.1609/icaps.v32i1.19820
- [23] Richard S Sutton and Andrew G Barto. 2018. Reinforcement learning: An introduction. MIT press.
- [24] Martin Tappler, Filip Cano Córdoba, Bernhard K. Aichernig, and Bettina Könighofer. 2022. Search-Based Testing of Reinforcement Learning. In Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022, Luc De Raedt (Ed.). ijcai.org, 503-510. https://doi.org/10.24963/IJCAI.2022/72

- [25] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In Proceedings of the 40th International Conference on Software Engineering. https://doi.org/10.1145/ 3180155.3180220
- [26] Bryon Tjanaka, Sam Sommerer, Nikitas Klapsis, Matthew C. Fontaine, and Stefanos Nikolaidis. 2021. Using CMA-ME to Land a Lunar Lander Like a Space Shuttle. pyribs.org (2021). https://docs.pyribs.org/en/stable/tutorials/lunar\_lander.html
- [27] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. 2023. Gymnasium. https://doi.org/ 10.5281/zenodo.8127026
- [28] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. 2020. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research* 68 (2020).
- [29] Fitash Ul Haq, Donghwan Shin, and Lionel C. Briand. 2023. Many-Objective Reinforcement Learning for Online Testing of DNN-Enabled Systems. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 1814– 1826. https://doi.org/10.1109/ICSE48619.2023.00155
- [30] 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
- [31] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, Mojtaba Bagherzadeh, and Ramesh S. 2023. A Search-Based Testing Approach for Deep Reinforcement Learning Agents. IEEE Transactions on Software Engineering 49, 7 (2023), 3715–3735. https://doi.org/10.1109/TSE.2023.3269804