Symbolic State Seeding Improves Coverage Of Reinforcement Learning

Mohsen Ghaffari
mohg@itu.dk
IT-University of Copenhagen
Copenhagen, Denmark

Cong Chen
coch@itu.dk
IT-University of Copenhagen
Copenhagen, Denmark

Mahsa Varshosaz
mahv@itu.dk
IT-University of Copenhagen
Copenhagen, Denmark

Einar Broch Johnsen
einarj@ifi.uio.no
University of Oslo
Oslo, Norway

Andrzej Wąsowski
wasowski@itu.dk
IT-University of Copenhagen
Copenhagen, Denmark

***Abstract*—Due to a limited learning budget, a reinforcement learning agent can only explore the most probable scenarios out of a potentially rich and complex environment dynamics. This may result in a limited understanding of the context and low robustness of the learned policy. A possible approach to address this problem is to explore the interactions between an autonomous agent and environment in rare but important situations. We proposes SymSeed, a method for initializing learning episodes for the class of reinforcement learning problems for which a simulation environment (model) is available. This increases the chance of exposing the agent to interesting states during learning. Inspired by techniques for increasing coverage in testing of software, we analyze the simulator implementation using symbolic execution. Then we generate initial states that ensure the agent explores the simulator dynamics well during learning. We evaluate SymSeed by feeding the generated states into well-known reinforcement learning algorithms, both tabular and approximating methods, including vanilla Q-Learning, DQN, PPO, A3C, SAC, TD3, and CAT-RL. In all test cases, the combination of SymSeed with uniform sampling from the entire state space enables all algorithms to achieve faster convergence and higher success rates than the baseline. The effect is particularly strong in presence of sparse rewards or local optima.**

***Index Terms*—Reinforcement Learning, Symbolic Execution, Initialization**

I. Introduction

Reinforcement learning is a machine learning method with applications in, e.g., self-adaptive systems [1], [2], robotics [3], gaming [4] and electronics [5]. With reinforcement learning, an agent learns a behavioral policy, while using it and iteratively self-adapting based on environment feedback [6]. In practice, the training of a reinforcement learning agent is often constrained by limited time, restricting exposure to only consider the most likely scenarios out of a large space. Limited exploration restricts the agent's understanding of the environment and poses several challenges; (1) It increases the risk of safety violations in critical systems, where a safe policy is needed for all states, not just for the most-likely-reachable states; (2) It increases the risk of terminating the learning process in a local optimum, while better policies exist; (3) It may learn policies that are overly sensitive to initial and observed states; (4) It is particularly challenged by sparse reward functions, where pockets of high reward are rare and hard to reach.

Exploration is a fundamental aspect of reinforcement learning [7], [8], [9]. Common approaches include ϵ-greedy exploration [6], count-based exploration [10], [11], [12], curiosity-based exploration [13], as well as methods specifically designed for exploring sparse rewards such as contextual Markov Decision Processes (MDPs) [14], [15]. All these methods begin by seeding initial states using a uniform distribution over a subset of the state space. Such a uniform selection of initial states can be problematic for several reasons. First, the agent may spend a considerable amount of time exploring areas of the state space that are free of immediate reward. Second, the agent may overfit to regions that it frequently encounters initially, while neglecting others that are critical in later stages of training. Third, the agent might focus on suboptimal areas, thereby reducing the utility of early policies.

Our objective is to improve the exploration process by automatically obtaining and leveraging knowledge about the dynamics of the agent's environment. In particular, we aim to identify states that are less likely to be sampled during training, and to ensure that the agent learns how to act in these states. We hypothesize that insights into the structure of the state space enables more efficient exploration. Initializing the agent in carefully selected states ensures that these states are explored, allowing the agent to learn appropriate policies for critical situations. This can also help the agent reach a goal state faster, as judicious initialization helps it to start in proximity of the high reward states and helps distributing the reward values backwards to intermediary states more effectively. This is particularly useful in scenarios with sparse rewards.

To provide reinforcement learning with relevant knowledge, we use software analysis tools, namely symbolic execution. Symbolic execution is a popular technique in program analysis, which can be used to automatically generate test inputs for programs or detect hidden problems in an implementation, guaranteeing high coverage of the analyzed code [16], [17], [18], [19]. A recent paper by the authors [20] demonstrates

that symbolic execution can be used to partition the state space of reinforcement learning. Be generating path conditions, symbolic execution effectively identifies classes of distinct initial states. We here exploit this method to improve exploration in reinforcement learning.

SymSeed, our strategy for initializing reinforcement learning episodes, is designed for problems where an environment simulator is available. SymSeed is also suited to pre-training scenarios, where the initial policy is obtained against a simulation, before reinforcement learning is used in a black-box environment, for instance in a physical environment for a robot. We use symbolic execution to analyze the environment simulator and to generate initial states. The hypothesis is that states leading to exploring different execution paths through the simulator, are also materially different from the policy perspective. Thus, ensuring a good coverage of the simulator during learning increases the signal between the environment and the agent during learning, leading to better policies faster. The main contributions are:

- SymSeed, a method for seeding reinforcement learning episodes combining symbolic execution, SMT solving, and rejection sampling.
- A Python implementation of SymSeed using Symbolic PathFinder[1] [21], Z3[2] [22], and DReal[3] [23]; integrated with the standard reinforcement learning frameworks: Stable-Baselines3[4] and Gymnasium.[5] The environment models are implemented in Java. The implementation of SymSeed will be released upon the acceptance.
- An empirical evaluation of SymSeed for well-known learning algorithms, including both tabular and approximating methods: Q-Learning, DQN, PPO, A3C, SAC, TD3, and CAT-RL. The combination of SymSeed with uniform sampling from the entire state space enables all algorithms to achieve faster convergence and higher success rates than the baseline in all the test cases.

We begin by presenting a motivating example in Sect. II. Section III reviews the relevant state of the art. Section IV recalls the required preliminaries and definitions. The SymSeed method is detailed in Sect. V. In Sect. VI, we discuss the experiment results. Finally, Sect. VII concludes the paper.

II. Overview

Consider an example problem, *safari car* (Fig. 1), a variation of the well-known *mountain car* problem [24]. This small problem exhibits relatively simple dynamics yet presents a challenge for reinforcement learning. A car can perform three actions: right, left, or skip. The objective is to learn the optimal policy to achieve the goal i.e., ascending to the rightmost hill, with the checkered flag. While ascending the first hill, the agent receives a small positive reward. This can potentially lead to the algorithm getting stuck in a local optimum.

[1]https://github.com/SymbolicPathFinder
[2]https://github.com/Z3Prover/z3
[3]https://github.com/dreal/dreal4
[4]https://github.com/DLR-RM/stable-baselines3
[5]https://gymnasium.farama.org/

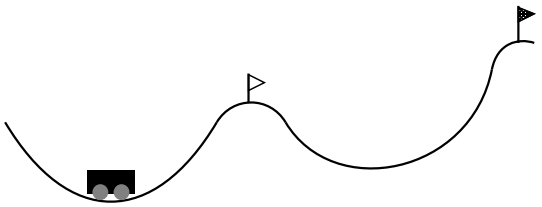


Fig. 1: A *safari car* receives a moderate reward for climbing the center hill, while the globally high reward is granted for climbing the rightmost hill.

Let us define the safari car MDP formally. The state space is $\overline{\mathcal{S}} \ni (p, v)$ where $-1.2 \leq p \leq 2.4$, $-0.07 \leq v \leq 0.07$ are the position and velocity of the car. The initial state $\overline{\mathcal{S}}_0$ is a subset of $\overline{\mathcal{S}}$. One of the well-known initializations is uniformly choosing the state from the entire state space, which means $\overline{\mathcal{S}}_0 = \overline{\mathcal{S}}$. The action space is defined as the set $\mathcal{A} = \{0, 1, 2\}$, in which 0, 1, and 2 represent accelerate to the left, skip, and accelerate to the right, respectively. For the state (p_t, v_t), representing the position and velocity of the car at time t, the output of the transition function $\mathcal{T}(p_t, v_t)$ is (p_{t+1}, v_{t+1}), which is defined as:

$$v_{t+1} = \begin{cases} v_t + (a-1)f - g\cos(3p_t), & -1.2 < p_t \leq 0.6 \\ v_t + (a-1)f - g\cos(3(p_t - 1.8)), & 0.6 < p_t \leq 2.4 \end{cases}$$
$$p_{t+1} = p_t + v_{t+1} \ , \tag{1}$$

where $f = 0.001$ and $g = 0.0025$. When the car reaches the leftmost position it stops there in state $(-1.2, 0)$. Further moves left are ignored. The reward function is:

$$\mathcal{R}(p_t, v_t) = \begin{cases} 1000, & \text{if } p_t = 2.4. \\ 1, & \text{if } 0.6 \leq p_t \leq 0.65. \\ -1, & \text{otherwise.} \end{cases} \tag{2}$$

The predicate $\mathcal{F}(p_t, v_t) = (p_t \geq 2.4)$ defines the final states.

Reinforcement learning algorithms often choose initial states by uniformly sampling from a subset of the state space. However, this approach has several significant challenges, particularly in problems where the state space is extremely large or continuous, such as robotic control or autonomous vehicles.

In continuous state spaces, the probability of selecting any specific state using a uniform distribution across the entire space is effectively zero. This makes uniform sampling impractical, if some important classes of states have very low measure. Hitting them randomly requires an infeasible number of samples. Moreover, the number of samples needed grows exponentially with the dimensionality of the state space, a phenomenon known as the "curse of dimensionality" [6]. In general, uniform sampling also fails to prioritize states that are more likely to occur during optimal or exploratory policies. As a consequence, two issues emerge. First, under-sampling of critical states; that is, important states, such as those near rewards or key decision points, may not be sampled frequently enough. Secondly, missing rare but meaningful states, which means

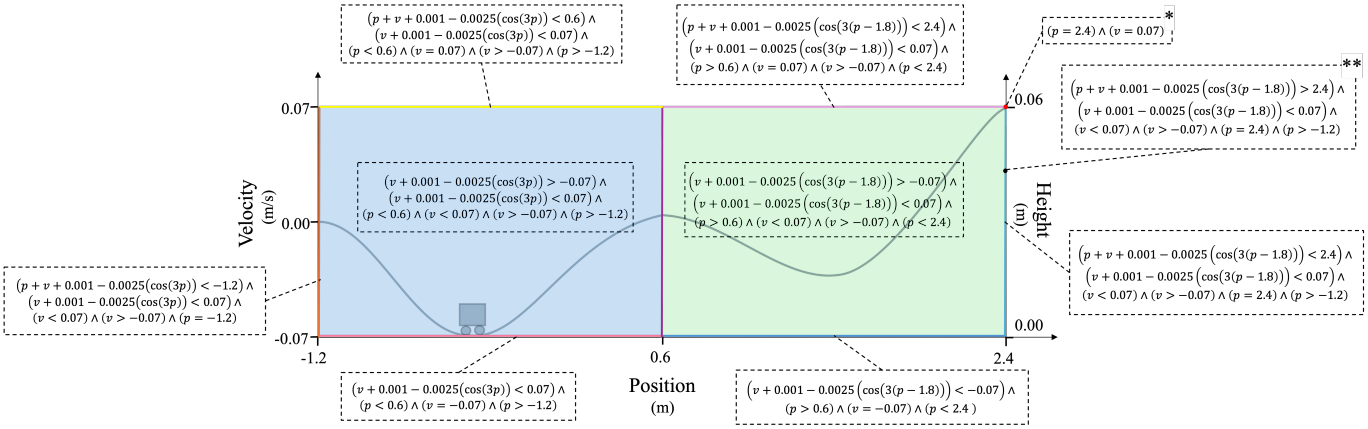


Fig. 2: Partitioning of the state space for the Safari car: 58 partitions in total. Many of the partitions are too small to visualize. Two of the small examples are marked by $*$ and $**$.

states that are rare but crucial for learning may be overlooked entirely, reducing the diversity of experience. Additionally, uniform sampling leads to inefficient resource allocation. For instance, training algorithms may waste computational effort processing states that are rarely encountered under actual policies. This inefficiency contributes to two main issues: *poor generalization*, whereby the policy may fail to prioritize frequently occurring states in the environment, and *slower convergence*, whereby the policy must adapt to the actual distribution of encountered states, which differs significantly from a uniform distribution. Finally, in environments with sparse rewards, uniformly sampling states exacerbates the difficulty of finding reward-relevant states, making it even harder for the agent to learn an effective policy.

Figure 2 shows a possible partitioning of the state space of the safari car into important classes of states. We explain how such a partitioning can be obtained in Sect. V. There are 58 classes in the partitioning, but only seven partitions are sufficiently large to be readily observed in the figure. The rest are mostly single state partitions. An example of a large partition is $(v + 0.001 - 0.0025\cos(3p) > -0.07) \wedge (v + 0.001 - 0.0025\cos(3p) < 0.07) \wedge (p < 0.6) \wedge (v < 0.07) \wedge (v > -0.07) \wedge (p > -1.2)$, shown in blue. It captures the states of the car in the first valley. The formula restricts the positions (horizontal in the figure) and velocities (vertical in the figure). The trajectory of the car in this partition is overlaid as a line, for convenience. Note that it aligns with the horizontal axis, and the right vertical scale (height). However, as height is not part of the state, the colored partitions only relate horizontal position and velocity (left). The height is purely informational—the agent does not know about it.

Given the size of the partitions, it is reasonable to assume that the agent may start from the left valley with the probability of 50%. Ascending the first mountain is technically identical to solving the classic mountain car problem. Upon reaching the summit of the first mountain, the agent tends to remain in that position. It is also possible that the agent learns to ascend the center mountain even if initialized in the right-most valley, and then remains in the local optimum for a period of time.

Similarly, the logical expression shown in green refers to the car movement in the right valley. Two other interesting examples of partitions are marked with asterisks ($*, **$). As these constraints use equality, they have zero surface (zero measure). The partition $p = 2.4 \wedge v = 0.07$ is one of the goal states. Notice that uniform sampling from the entire state space is likely to hit one of the two large areas (green and blue), but extremely unlikely to hit zero measure areas, even if these states are interesting.

The main idea of SymSeed is to represent partitions as logical predicates obtained from a program analysis tool. Then, use solutions for the corresponding logical predicate to seed a rejection sampler, which in turn produces initialization values for the learning episodes. SymSeed first ensures that at least one solution from each partition is generated using a solver. Then inflates each solution to a set by adding noise. With such hierarchical generative approach, the probability of initializing an episode from any partition is $1/58$, independently of the size of the partition. When using uniform sampling the probability is about $1/2$ for the big partitions and zero for all the others.

III. Related Work

Although the initial states of a trained model play an instrumental role in performance of reinforcement learning [25], [26], state seeding has received much less attention than policy initialization [27], [28], [29], [30]. Policy initialization, which is useful for policy optimization, makes the agent learn a policy that is roughly analogous to an initial policy, but does not trigger a more comprehensive exploration of the environment. Thus, the exploration of the state space remains limited to a specific range. A reinforcement learning algorithm may get stuck in local optima, impeding its ability to explore the state space effectively. Furthermore, states that are less likely to be observed may be entirely bypassed with policy initialization. This poses a significant risk for safety-critical systems. Another line of work that also emphasizes the importance of the initial states, optimizes an ensemble of policies over different "slices" of the

```
1 def step(p, v, a):
2   v += a+math.cos(3*p)
3   p += v
4   r = -1.0
5   if p == 0.5:
6     r = 100
7   return p, v, r
```

Fig. 3: A simple step function

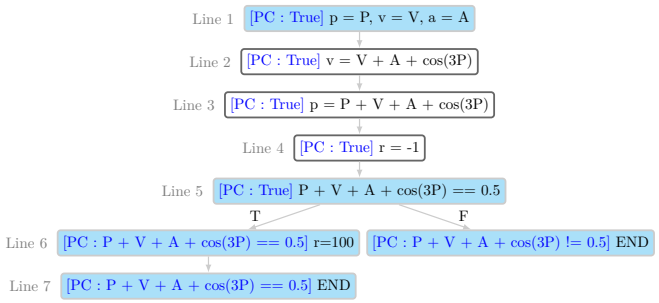


Fig. 4: Execution tree for the symbolic input p = P, v = V, a = A and the function of Fig. 3

state space [31]. Their objective is to partition the state space in order to simplify the complex tasks, which has no impact on the exploration of the state space. The work is constrained to problems with multiple tasks, due to the necessity of an MDP with a contextual structure. Furthermore, the partitioning of the state space is based on a limited number of samples, which may still result in the exclusion of small but critically important partitions of the state space. In contrast, we develop a method for seeding initial states for a broader class of reinforcement learning problems that leads to better state exploration.

In robotics research, where agents are commonly trained on simulators, there is interest in mapping the trained initial states from the simulation level to states in the real world [32], [33]. These works do not address the initial states that an agent would take at the training level in the simulator but focus on the initial states that an agent, such as a robot, should adopt when retrained or tested in the real world. In robotics, another branch of study attempts to learn the reset function or initial states [34]. Similarly, Messikommer et al. select states from previous experiences and use them to initialize the agent in the environment, thereby guiding it toward a more informative state [35]. The use of past experiences to improve policy has been the subject of extensive study in the literature [36], [37], [38]. These works demonstrate the significance of maintaining sufficiently limited initial states to facilitate the repetition of previously explored states. Moreover, the set of initial states should be sufficiently expansive to guarantee that the agent has adequately explored the state space, which is not considered in the above-mentioned works.

IV. Background

Symbolic Execution is a software analysis technique used to automate software testing to find program errors [39]. It extends normal execution by running the basic operators of a language using symbolic inputs, i.e. logical variables, instead of concrete values [40]. The values of program variables become mathematical expressions over the symbolic inputs. For each path executed through the program, the analysis maintains a symbolic *path condition* which encodes the conditions on the inputs for the execution to follow that path. Each path condition is built by accumulating the branch conditions encountered during the execution of the program. Subsequently, the symbolic executor performs a satisfiability check of the path condition, thereby determining whether the corresponding branch to the path condition can continue to grow or whether it should be terminated. Additionally, in the presence of loops and recursion, symbolic execution does not terminate. To halt symbolic execution, one may set a predefined timeout, an iteration limit, or a program statement limit [20]. This results in an under-approximation of the set of path conditions. The symbolic execution engine maintains an internal representation, a symbolic execution tree, to keep track of the observed conditions during the execution.

Figure 3 shows an example of a simple *step function*, and Fig. 4 the corresponding symbolic execution tree for this function, generated by symbolic execution in a step-by-step manner, with each level of the tree corresponding to the same line in the code. For example, there is a condition in line 5 of the program, which, when executed symbolically, results in two branches in the execution tree: one for the true case of the condition and the other for the false case. As illustrated, the execution tree carries a path condition PC in each node. The tree's leaf nodes show the logical expressions that must be satisfied to execute the corresponding execution path in the program. This use of symbolic execution in reinforcement learning is discussed in detail in a recent paper [20].

Markov Decision Processes (MDPs) are discrete-time stochastic control structures, which assume that the distribution of the future states is only dependent on the present state and independent of the past execution history [41]. Formally, an MDP is a tuple $(\overline{\mathcal{S}}, \overline{\mathcal{S}}_0, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, where $\overline{\mathcal{S}}$ is a set of states, $\overline{\mathcal{S}}_0 \in$ pdf $\overline{\mathcal{S}}$ is a probability density function for initial states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T} \in \overline{\mathcal{S}} \times \mathcal{A} \to$ pdf $\overline{\mathcal{S}}$ is the transition probability function for successor states for transitions from a given state with a given action, $\mathcal{R} \in \overline{\mathcal{S}} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\mathcal{F} \in \overline{\mathcal{S}} \to \{0, 1\}$ is a predicate defining final states.

Reinforcement Learning is a data-driven controller synthesis method for tasks where an agent interacts with an environment through actions, observations and rewards [6]. A reinforcement learning problem can be modeled using an MDP, in which the task is to find a *policy* π that selects actions in different states to maximize the expected accumulated reward (so reinforcement learning is a statistical method for solving MDPs). To learn an optimal policy π^*, the action-value function can be represented

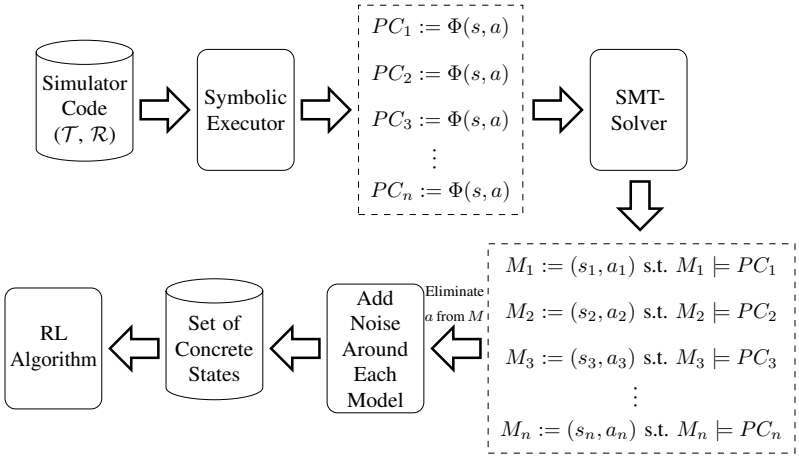


Fig. 5: The SymSeed method

in, e.g., a Q-table [42], which we update using the equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big] , \quad (3)$$

where r_{t+1} is the reward received after taking action $a_t \in \mathcal{A}$ in state $s_t \in \overline{\mathcal{S}}$, $0 < \alpha \leq 1$ is the *learning rate*, $0 < \gamma \leq 1$ is the *discount factor*, $\max_{a'} Q(s_{t+1}, a')$ is the maximum Q-value for the next state $s_{t+1} \in \overline{\mathcal{S}}$ across all possible actions $a' \in \mathcal{A}$. The optimal action-value function $Q^*(s, a)$ can then be obtained by $Q^*(s, a) = \max_\pi Q^\pi(s, a)$. We give a brief overview of the reinforcement learning algorithms that will be used for the evaluation of the methodology proposed in this paper.

Deep Q-Network (DQN) learning overcomes challenges arising when the dimensions of the state space increase [43]. Unlike traditional Q-learning, which relies on a Q-table, DQN leverages Deep Neural Networks (DNNs) to approximate value functions for continuous state spaces. In each episode, the DQN updates its parameters by learning from the agent's experiences, allowing it to estimate the expected cumulative reward more efficiently. The DNN model is optimized to approximate the optimal action-value function by adjusting its parameters based on observed state transitions and rewards.

Asynchronous Advantage Actor-Critic (A3C) is a reinforcement learning algorithm in which multiple agents (or copies of the environment) work in parallel to learn. These agents operate independently and asynchronously, meaning they collect experiences at different times [44]. Each agent uses an *actor* network to decide on actions and a *critic* network to evaluate how good those actions are, based on the current state. By working together, the agents share their learned experiences with a global model, allowing it to learn more efficiently and effectively.

Proximal Policy Optimization (PPO) is a policy gradient algorithm that combines ideas from Asynchronous Actor-Critic (A2C-having multiple workers) and Trust Region Policy Optimization (TRPO; using a trust region to improve the actor) [45]. It iteratively learns a parameterized policy π_θ. In standard implementations, PPO regularizes policy updates with clipped probability ratios, and parameterizes policies with either continuous Gaussian distributions or discrete Softmax distributions [46].

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an improved variant of Deep Deterministic Policy Gradient (DDPG) designed to enhance stability and performance in continuous action spaces [47]. TD3 addresses several critical limitations of DDPG, particularly its vulnerability to overestimated Q-values, which can lead to unstable policies. TD3's improvements include clipped double Q-learning, which reduces overestimation by training two separate Q-networks and using the minimum of their estimates for policy updates. Furthermore, it uses delayed policy updates, i.e., it reduces instability by updating the policy network at a slower rate than the Q-networks, allowing for more accurate value estimations before each policy change. Finally, target policy smoothing introduces noise to the target actions, reducing the likelihood of exploiting minor errors in Q-value estimation.

Soft Actor-Critic (SAC) is a reinforcement learning algorithm which optimizes a stochastic policy using an off-policy method, bridging the gap between stochastic policy optimization and DDPG approaches [48]. SAC performs well in environments that require a delicate balance between exploration and exploitation, making it well-suited for continuous control tasks. To stabilize learning, SAC employs the clipped double-Q trick to stabilize learning, and its stochastic policy benefits from target policy smoothing, which enhances performance. A core aspect of SAC is entropy regularization, where the policy maximizes a trade-off between expected return and entropy.

Conditional Abstraction Trees for Sample-Efficient Reinforcement Learning (CAT-RL) is a top-down approach for constructing state abstractions while learning [49]. Starting with state variables and a simulator, it dynamically computes an abstraction based on the dispersion of temporal difference errors in abstract states as the agent continues acting and learning.

V. Symbolic State Seeding (SymSeed)

We propose a methodology for generating a set of initial states for reinforcement learning algorithms by analyzing the environment dynamics, which we assume to be simulated by a

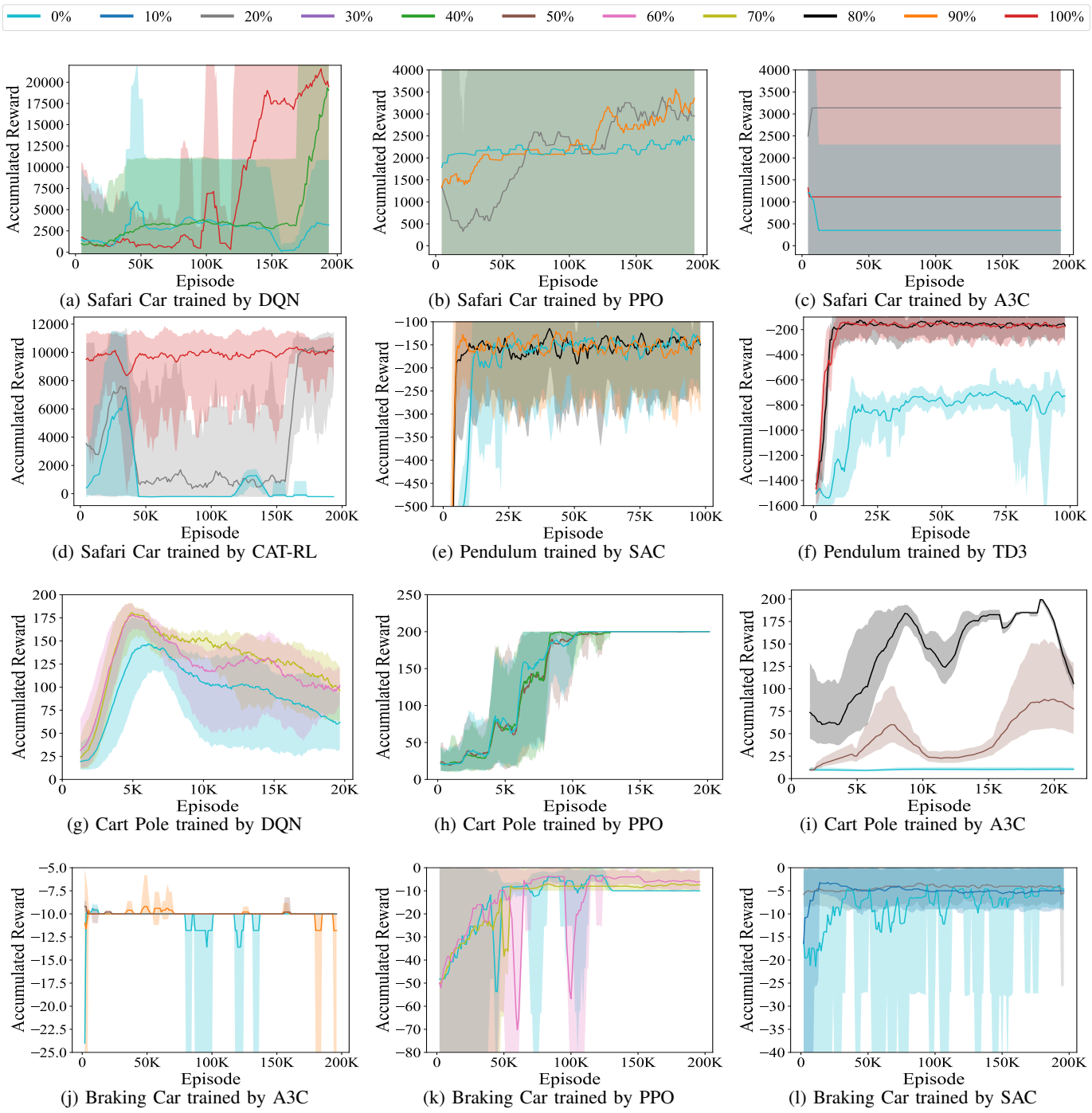


Fig. 6: Reinforcement learning algorithms evaluated per 100 training episodes, for ten uniformly sampled initial states. Each plot shows the accumulated reward (overall performance), minimum reward, and maximum reward (range of outcomes) for the baseline (0%, sky blue), pure SymSeed (100%, red), and a mixture of the two (varying color and percentage).

computer program. To this end, the environment simulator is executed symbolically to extract path conditions (PCs), using an off-the-shelf tool. Each PC is a logical expression over the input variables of the program, in this case the state and the action of the reinforcement learning agent. We can then solve each PC using an SMT solver, resulting in a concrete state and action that together satisfy the given PC. Finally, we introduce noise around these states (by a rejection sampling technique) in order to increase the number of samples, and use all the obtained states as initial states for a reinforcement

learning algorithm, see Fig. 5.

The environment simulator is a program that takes the current state of the agent and its action, and computes the next state that the agent reaches and the immediate reward obtained for the current transition, a so-called *step function*. In other words, the environment simulator implements a single-step transition of the MDP for the given problem, reflecting the environment dynamics. Remark that the simulator does not necessarily need to be a faithful rendition of the real environment, in full detail. On the contrary, abstract environment models can be used for pre-training simulations.

The symbolic execution of a program explores the program's feasible execution paths and generates a set of PCs; each PC characterizes a specific execution path within the program. To execute a program symbolically, two key elements must be in place: the program itself (or the byte code of the program) and the variables within the program that are to be treated as symbolic variables. The symbolic execution of a given step function captures the environment dynamics in a set of PCs, where each PC becomes a logical expression over the state and action variables. The specific techniques used to solve the PCs may vary depending on the chosen SMT solver. When asked for the solution to a PC expression, the solver returns concrete values for the variables of the formula that can satisfy this logical expression. These concrete values can be used as initial states for learning. It should be noted that the symbolic executor verifies the satisfiability of PCs; it follows that each PC has at least one solution.

SMT solvers typically yield one solution for a given formula, even if multiple solutions exist. Although feasible, obtaining more solutions from an SMT solver is time-consuming, as the formula to be solved typically grows with each iteration. Instead, we obtain a single solution from the SMT solver and then add a small amount of noise to this solution to obtain multiple states. The samples are selected via an accept/reject sampling technique [50]. In this manner, for each noisy sample, we ascertain whether it remains within the same partition. This process is repeated until either k states are obtained from each partition or the new samples are rejected k times. This is due to the fact that the size of the partitions may be very small, which makes it impossible to generate additional states from them. It is highly probable (heuristically) that this approach increases the number of samples for each PC.

The result of solving each PC using the SMT solver, is a set of concrete values for state and action variables that can satisfy the given PC. Notice that although these solutions include values for both state and action variables, actions are not initialized in reinforcement learning. Consequently, the action values are excluded from the answers, yielding a set of concrete values for state variables. Next, we introduce noise around each state value to avoid sample bias. Ultimately, this set can now be fed into learning algorithms, either in tabular or deep methods.

We implemented SymSeed using Symbolic PathFinder[6] [21] to calculate the path conditions. As PathFinder works with JVM, we need to provide the simulation programs in this format (we use Java in the experiments). Z3[7] [22] is used as the main SMT-Solver. We switch to DReal[8] [23] to handle problems with non-linear functions such as trigonometric functions. We will release the implementation in a public repository upon the acceptance of the paper.

[6]https://github.com/SymbolicPathFinder

mixing	states	max freq.	mean freq.	std.	mean rew.	max rew.
0%	159028	35662	5.64	127.03	724.32	6923.4
10%	176573	29131	10.82	112.74	10570.62	10785.5
20%	194564	20766	11.15	80.69	3321.27	10454.0
30%	153059	27072	12.84	249.43	2095.24	3798.7
40%	178155	14351	11.93	87.79	2699.39	4279.4
50%	177352	21108	10.82	117.84	5472.33	6786.0
60%	149587	21543	4.99	118.35	2975.95	7685.4
70%	109722	47648	11.7	307.09	5393.39	5685.9
80%	160338	40829	11.72	264.9	6294.42	8073.2
90%	138977	45349	11.74	329.14	7135.45	9260.9
100%	145279	40533	9.31	267.8	9811.23	10377.0

TABLE I: States visited by Safari Car trained by CAT-RL

VI. Evaluation

A. Experiment Setup

We ask the following research questions to evaluate the performance and efficacy of SymSeed.

RQ1. *Can SymSeed decrease the number of visited states and yet improve the reward?*

RQ2. *Does SymSeed help to avoid local optima?*

RQ3. *How much does SymSeed improve the performance in the presence of sparse rewards?*

To answer these questions, we apply SymSeed to well-known reinforcement learning algorithms: Q-Learning [6], DQN [43], A3C [44], TD3 [47], SAC [48], PPO [46], using the Stable-Baselines3 implementations [51], and CAT-RL [49]. For each algorithm, we conduct a series of experiments with several classic case studies. The training of each agent is conducted using three distinct initialization strategies: *(a)* a uniform sampling over the entire state space, *(b)* a solving-and-sampling using SymSeed, and *(c)* a mixture of (a) and (b), whereby the percentage of the mix is controlled.

To answer **RQ1**, we collect the visited states during training for each initialization strategy. Additionally, we measure the mean, minimum, and maximum of the accumulated rewards for ten randomly selected states at specified episodes.

To answer **RQ2**, we designed a series of examples, e.g., the Safari Car test case, which demonstrate how local optima may impede an agent's progress if the environment is not adequately explored. We say that the agent has succeeded if it identifies the global optimum. In these experiments, the success rate was measured during training for each initialization strategy.

To answer **RQ3**, we designed a series of examples based on Office World, in which the agent only obtains a reward in final states (i.e., no intermediate rewards). The goal is to show how SymSeed helps the agent to find goal states in early episodes

[7]https://github.com/Z3Prover/z3

[8]https://github.com/dreal/dreal4

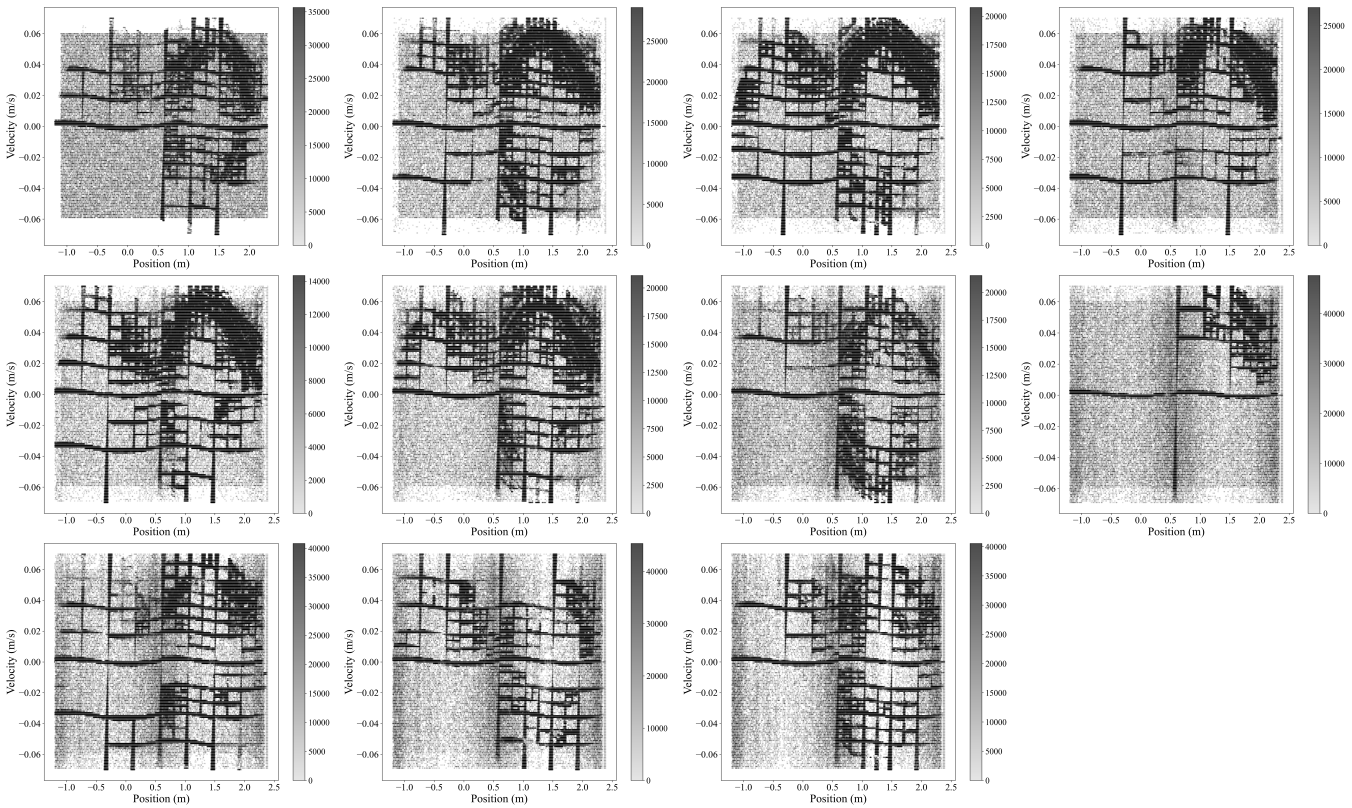


Fig. 7: Visited states of Safari Car example during 100K episodes training of CAT-RL. Each plot corresponds to a specific strategy of using SymSeed, starting from 0% in the top-left corner and increasing in 10% increments, progressing row by row from left to right, until reaching 100% in the bottom-right one.

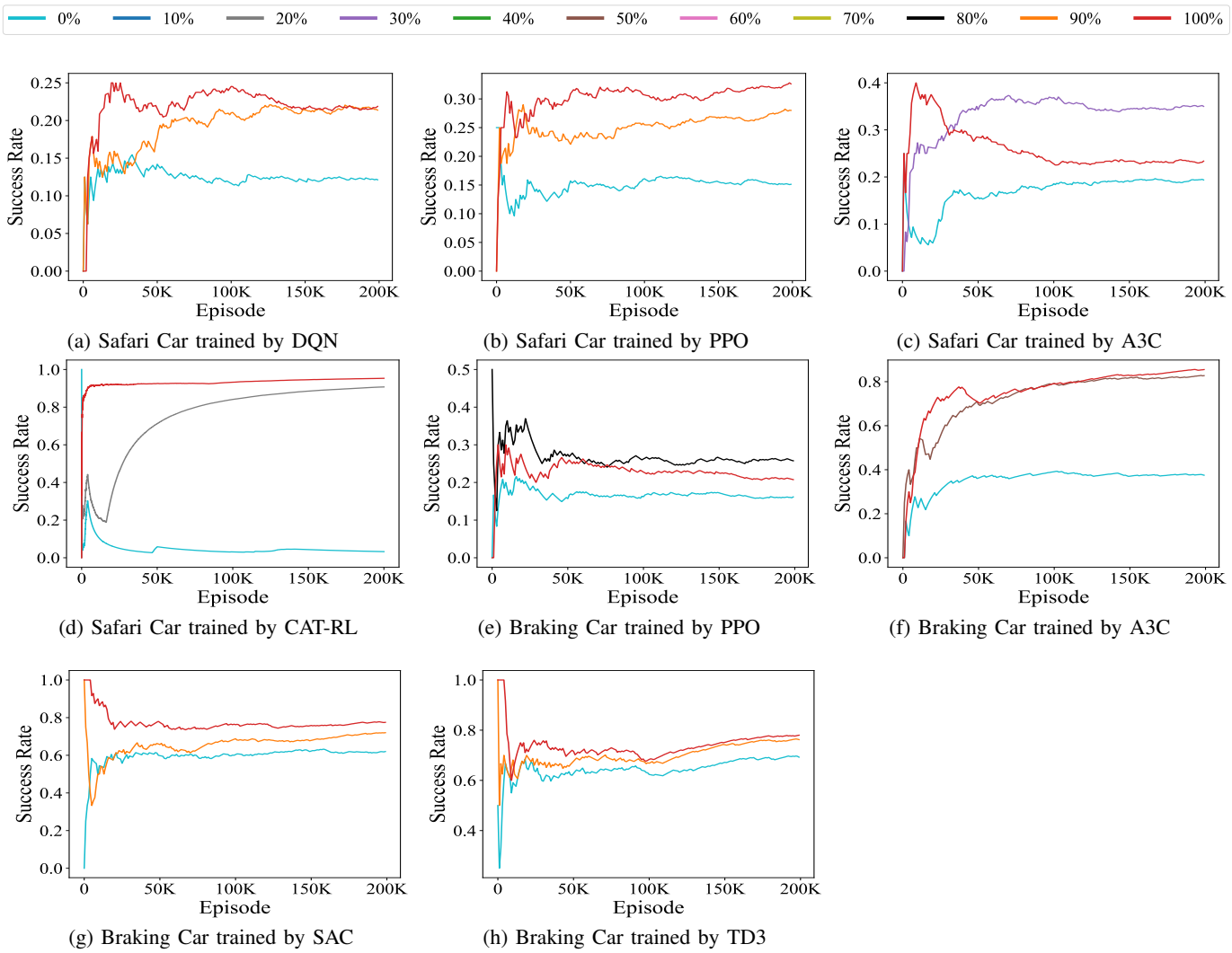
and to expand awareness of these goal states to the rest of states. We measure the accumulated reward to evaluate the trained policy every ten episodes for ten randomly selected states.

B. Test Problems

We use three modifications of the *Office World* environment [49]. *Office World 1* problem is a grid map comprising four distinct rooms at its corners. The objective is to collect and deliver mail and coffee to the designated office location, resulting in a positive reward; otherwise, no reward is given. *Office World 2* is analogous to Office World 1, with the exception that the location of the goal is situated at the farthest distance from the start position of the agent. *Office World 3* is a combination of the first and the second one. It has two distinct goal states. The agent only succeeds in one of the two possible outcomes, while the other goal acts as a local optimum. *Braking Car* describes a car moving towards an obstacle with a given velocity and distance. The goal is to stop the car to avoid a crash with minimum braking pressure [52]. *Safari Car* aims to learn how to obtain enough momentum to move up two steep slopes and has similar dynamics to the mountain car [24], cf. Sect. II. The *Pendulum* environment comprises a pendulum attached at one end to a fixed point, and the other end is free to move. The goal is to apply a torque on the free end to swing it into an upright position. *Cart Pole* problem concerns a pole attached to a cart by an unactuated joint, which is moved along a frictionless track. The pendulum is placed in an upright position on the cart, and the objective is to balance the pole by applying forces in the left and right directions on the cart.

C. Results

RQ1 (efficiency of learning). Figure 6 summarizes the performance of reinforcement learning algorithms when seeded with SymSeed. Three scenarios are considered: initialization purely with SymSeed (100%), not using SymSeed at all (0%, baseline uniform initialization), and a mixture of SymSeed and uniform. The baseline (0%) is shown against the two best mixtures in all the plots. The comparison of achieved accumulated rewards for each of these strategies (Fig. 6) shows that SymSeed has enhanced the performance of all the studied reinforcement learning algorithms, resulting in higher rewards. Furthermore, SymSeed obtains higher rewards faster than uniform state initialization. It is noteworthy that the cart pole environment only branches on the final states. Consequently, the symbolic execution generates only few path conditions. For this reason, we did not anticipate the observed significant improvement over the uniform sampling (figures 6g to 6i). Another interesting observation is that mixing uniform sampling over the state space with SymSeed yields a more



(a) Safari Car trained by DQN (b) Safari Car trained by PPO (c) Safari Car trained by A3C

(d) Safari Car trained by CAT-RL (e) Braking Car trained by PPO (f) Braking Car trained by A3C

(g) Braking Car trained by SAC (h) Braking Car trained by TD3

Fig. 8: The success rate is calculated during the learning by counting success for all training episodes. The baseline (0%, sky blue), all initial states generated by SymSeed (100%, red), and a mixed method are selected.

notable improvement for almost all the cases, so the method works best when combined with a random initialization.

An analysis of the visited states is presented in Table I for CAT-RL and Safari Car. The table aggregates the number of new states observed by the agent during training (states), the maximum number of times that a state is visited (max freq.), the mean of the frequency of visited states (mean freq.), the standard deviation of the frequency of visited states (std.), the mean of the accumulated reward during training (mean rew.), and the maximum accumulated reward (max rew.). The results show that even though the initial states provided by SymSeed lead to CAT-RL exploring a smaller number of new states than the uniform baseline (0%), CAT-RL still achieves higher accumulated reward with SymSeed. Figure 7 displays heatmap plots of visited states in the same experiment. The weight of points in the plots refers to the frequency with which a given state has been visited. As the training for all strategies is fixed to 100,000 episodes for all the ten runs, a higher number of visits to a single state/region in this plot indicates a lower number of new states visited. The plots show that SymSeed concentrates learning around different classes, which seems to accelerate the reward accumulation and enhances the efficiency of exploration.

RQ2 (local vs global optima). The success rate (the rate of achieving the globally optimal reward in policy evaluation) for each of the aforementioned initialization strategies is summarized in Fig. 8. It shows that each of those algorithms learns policies that achieve the global optimum more often when using SymSeed-generated initial states than otherwise. However, there is no clear trend in the observed outcomes when the percentage in the mixture is increased. In some cases, including more SymSeed states leads to better performance, while in others the opposite is true. Furthermore, while

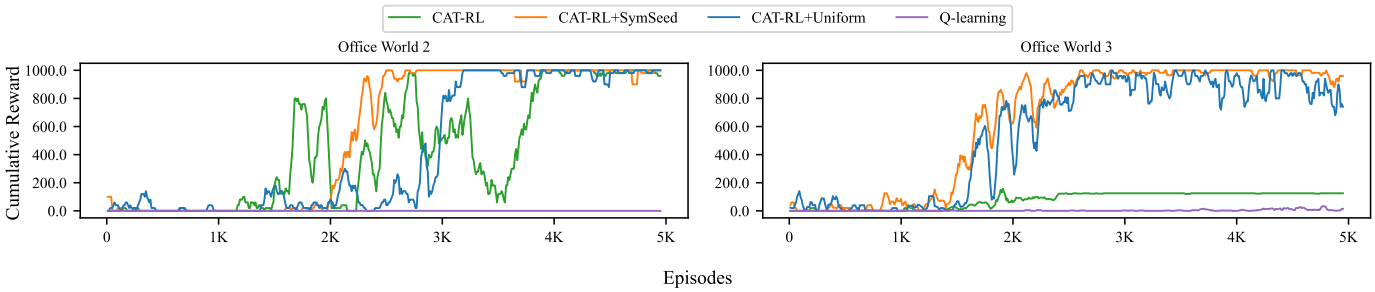


Fig. 9: Cumulative reward of evaluating tabular reinforcement learning algorithms on Office World 2 and Office World 3 with 5K episodes. The evaluation has been run ten times every 100 episodes starting from the same ten randomly selected initial states.

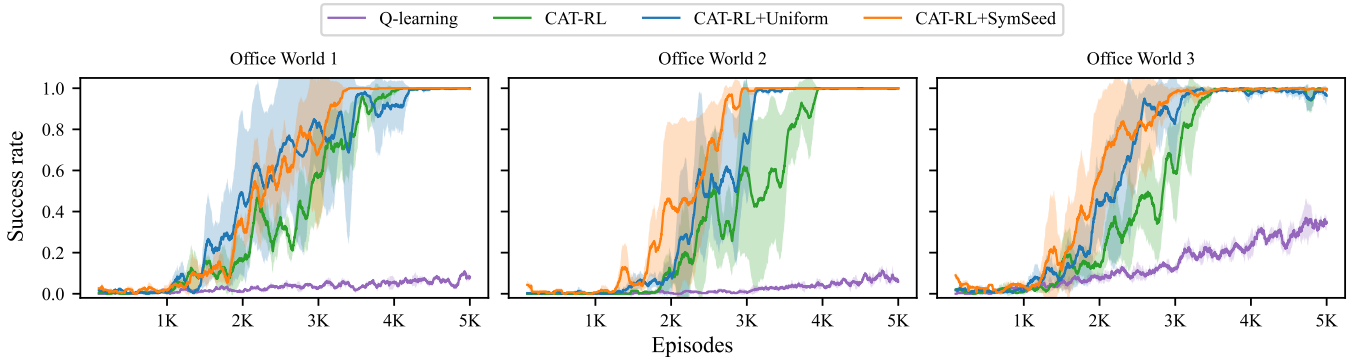


Fig. 10: The success rate is calculated during the learning by counting success for all training episodes over three runs of tabular reinforcement learning algorithms on three Office World domains with 5K episodes.

SymSeed-generated initial states generally enhance success rates across reinforcement learning algorithms, the optimal balance of SymSeed states versus random states varies depending on the specific algorithm and environment. This indicates that the integration of SymSeed states should be carefully calibrated, as an increase does not always correlate with improved performance, suggesting a nuanced interaction between state initialization and algorithm dynamics.

RQ3 (sparse rewards). We are using Office World with two goals at the different levels of reward and CAT-RL to investigate this question. We chose CAT-RL as the Office World is a discrete state environment, so approximate methods do not naturally apply. Moreover, CAT-RL (with a fixed initialization) has been shown to perform well in this kind of environment already. Recall that Office World 2 has sparse rewards, and Office World 3 has a local optimum and sparse information about the global optimum. Figure 9 shows that CAT-RL initialized by SymSeed, converges to an optimal policy faster than with other initialization strategies; the green line labelled CAT-RL uses a single initial state, and the blue line uses uniformly random initialization. Q-Learning is plotted to underline that the problem is hard for classic algorithms. The faster convergence with SymSeed can be attributed to the fact that the agent may start from states that are in close proximity to the final states (given that in many cases, the final states are conditionally defined, and SymSeed is capable of acquiring this knowledge through symbolic execution). This allows the agent to observe the final state earlier. This also leads to succeeding earlier in reaching the global optimum (Fig. 10), as it enables the distribution of the reward across the neighboring states of the final state. Consequently, the impact of the reward is distributed rapidly, which is equivalent to an environment with a non-sparse reward. Subsequently, initializing with SymSeed-seeded states provides a strategic advantage by facilitating early exposure to rewarding states, thereby accelerating policy convergence. This advantage highlights the potential of SymSeed to optimize reinforcement learning training efficiency by transforming sparse reward environments into effectively denser ones, improving both learning speed and stability.

D. Limitations

SymSeed can only handle environments that are implemented as programs. The worst case for SymSeed are problems for which the environment has very limited branching; e.g., Cart Pole discussed above. The simulation of Cart Pole only branches on final states; its dynamics is a physical formula over the position and velocity of the cart, and the angle and angular velocity of the pole. The path conditions found by symbolic execution are of little help here. In these cases, SymSeed acts similar to sampling uniformly from the entire state space. As our experiments show, combining SymSeed with uniform sampling in some cases outperform standalone SymSeed. This

may be because the coverage of SymSeed is sensitive to the selected standard deviation of the added noise. Furthermore, the SMT solver is not guaranteed to provide solutions (or different unique solutions that enable sufficient coverage); this issue can be mitigated by means of SMT sampling [53]. Finally, the partitioning using symbolic execution is sensitive to the level of granularity [20], which can influence the results of SymSeed.

VII. Conclusion

This paper shows that reinforcement learning algorithms are sensitive to how initial states are selected for each learning episode. The acquisition of additional knowledge about the environment can facilitate more optimal seeding of the algorithms. Furthermore, we have introduced a method SymSeed that facilitates more effective exploration and performance of reinforcement learning algorithms. In this context, a pre-analysis with an environment simulator allows for the generation of a set of initial states for reinforcement learning algorithms that can enhance exploration and facilitate more effective response to sparse rewards and local optima in the policy state. SymSeed is using a simple idea for adding noise around the models that are obtained from path conditions of a symbolic executor with help of an SMT-Solver. Other sampling techniques exist that may lead to a better set of initial states, for instance using SMT-samplers [53] or MCMC inference methods for guiding in the reward space [54]. At the same time, examining the impact of the number of noisy samples and the magnitude of the noise can provide valuable insights, enabling informed parameter selection. From software engineering perspective on reinforcement learning, it would be interesting to integrate better adaptive learning and testing of the reliability of policies, to harden the reliability guarantees.

Acknowledgments

This work was partially funded by DIREC (Digital Research Centre Denmark), a collaboration between the eight Danish universities and the Alexandra Institute supported by the Innovation Fund Denmark.

References

[1] M. Li, D. Yang, Y. Xu, and T. Ji, "Hierarchical deep reinforcement learning for self-adaptive economic dispatch," *Heliyon*, vol. 10, no. 14, 2024.

[2] A. Metzger, C. Quinton, Z. Á. Mann, L. Baresi, and K. Pohl, "Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration," *Computing*, vol. 106, no. 4, pp. 1251–1272, 2024.

[3] B. Singh, R. Kumar, and V. P. Singh, "Reinforcement learning in robotic applications: a comprehensive survey," *Artificial Intelligence Review*, vol. 55, no. 2, pp. 945–990, 2022.

[4] M. Mekni, C. S. Jayaramireddy, and S. V. V. S. S. Naraharisetti, "Reinforcement learning toolkits for gaming: A comparative qualitative analysis," *Journal of Software Engineering and Applications*, vol. 15, no. 12, pp. 417–435, 2022.

[5] M. Ghaffari and M. Afsharchi, "Learning to shift load under uncertain production in the smart grid," *Intl. Transactions on Electrical Energy Systems*, vol. 31, no. 2, p. e12748, 2021.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[7] J. Tarbouriech, E. Garcelon, M. Valko, M. Pirotta, and A. Lazaric, "No-regret exploration in goal-oriented reinforcement learning," in *Proc. 37th Intl. Conference on Machine Learning (ICML 2020)*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 9428–9437. [Online]. Available: http://proceedings.mlr.press/v119/tarbouriech20a.html

[8] R. Fruit and A. Lazaric, "Exploration-exploitation in MDPs with options," in *Proc. 20th Intl. Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, ser. Proceedings of Machine Learning Research, A. Singh and X. J. Zhu, Eds., vol. 54. PMLR, 2017, pp. 576–584. [Online]. Available: http://proceedings.mlr.press/v54/fruit17a.html

[9] J. Z. Kolter and A. Y. Ng, "Near-bayesian exploration in polynomial time," in *Proc. 26th Annual Intl. Conference on Machine Learning (ICML 2009)*, ser. ACM International Conference Proceeding Series, A. P. Danyluk, L. Bottou, and M. L. Littman, Eds., vol. 382. ACM, 2009, pp. 513–520. [Online]. Available: https://doi.org/10.1145/1553374.1553441

[10] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS 2016)*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 1471–1479. [Online]. Available: https://proceedings.neurips.cc/paper/2016/hash/afda332245e2af431fb7b672a68b659d-Abstract.html

[11] A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for Markov decision processes," *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008.

[12] M. C. Machado, M. G. Bellemare, and M. Bowling, "Count-based exploration with the successor representation," in *Proc. 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press, 2020, pp. 5125–5133. [Online]. Available: https://doi.org/10.1609/aaai.v34i04.5955

[13] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *Proc. 34th Intl. Conference on Machine Learning (ICML 2017)*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 2017, pp. 2778–2787. [Online]. Available: http://proceedings.mlr.press/v70/pathak17a.html

[14] R. Raileanu and T. Rocktäschel, "RIDE: rewarding impact-driven exploration for procedurally-generated environments," in *Proc. 8th Intl. Conference on Learning Representations (ICLR 2020)*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rkg-TJBFPB

[15] D. Zha, W. Ma, L. Yuan, X. Hu, and J. Liu, "Rank the episodes: A simple approach for exploration in procedurally-generated environments," in *Proc. 9th Intl. Conference on Learning Representations (ICLR 2021)*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=MtEE0CktZht

[16] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[17] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[18] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Irlbeck, D. A. Peled, and A. Pretschner, Eds. IOS Press, 2015, vol. 40, pp. 26–41.

[19] Y. Chen, W. Tsai, W. Wu, D. Yen, and F. Yu, "PyCT: A python concolic tester," in *Proc. 19th Asian Symposium on Programming Languages and Systems (APLAS 2021)*, ser. Lecture Notes in Computer Science, H. Oh, Ed., vol. 13008. Springer, 2021, pp. 38–46. [Online]. Available: https://doi.org/10.1007/978-3-030-89051-3_3

[20] M. Ghaffari, M. Varshosaz, E. B. Johnsen, and A. Wąsowski, "Symbolic state partitioning for reinforcement learning," in *Proc. 28th Intl. Conference on Fundamental Approaches to Software Engineering (FASE 2024)*, ser. Lecture Notes in Computer Science. Springer, 2025, to appear.

[21] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[22] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Proc. 14th Intl. Conf. on Tools and Algorithms for the Construction*

and Analysis of Systems (TACAS 2008), ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[23] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Proc. 24th Intl. Conf. on Automated Deduction (CADE-24)*, ser. Lecture Notes in Computer Science, vol. 7898. Springer, 2013, pp. 208–214.

[24] A. W. Moore, "Efficient memory-based learning for robot control," Ph.D. dissertation, University of Cambridge, UK, 1990.

[25] M. Andrychowicz, A. Raichuk, P. Stanczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters for on-policy deep actor-critic methods? A large-scale study," in *Proc. 9th Intl. Conference on Learning Representations (ICLR 2021)*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=nIAxjsniDzg

[26] S. Jang and H.-I. Kim, "Entropy-aware model initialization for effective exploration in deep reinforcement learning," *Sensors*, vol. 22, no. 15, p. 5845, 2022.

[27] D. Abel, Y. Jinnai, Y. S. Guo, G. D. Konidaris, and M. L. Littman, "Policy and value transfer in lifelong reinforcement learning," in *Proc. 35th Intl. Conference on Machine Learning (ICML 2018)*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 20–29. [Online]. Available: http://proceedings.mlr.press/v80/abel18b.html

[28] I. Uchendu, T. Xiao, Y. Lu, B. Zhu, M. Yan, J. Simon, M. Bennice, C. Fu, C. Ma, J. Jiao, S. Levine, and K. Hausman, "Jump-start reinforcement learning," in *Proc. Intl. Conference on Machine Learning (ICML 2023)*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 2023, pp. 34 556–34 583. [Online]. Available: https://proceedings.mlr.press/v202/uchendu23a.html

[29] A. Barreto, S. Hou, D. Borsa, D. Silver, and D. Precup, "Fast reinforcement learning with generalized policy updates," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 117, no. 48, pp. 30 079–30 087, 2020.

[30] L. Kraemer and B. Banerjee, "Reinforcement learning of informed initial policies for decentralized planning," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 4, pp. 1–32, 2014.

[31] D. Ghosh, A. Singh, A. Rajeswaran, V. Kumar, and S. Levine, "Divide-and-conquer reinforcement learning," in *Proc. 6th Intl. Conference on Learning Representations (ICLR 2018)*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rJwelMbR-

[32] W. Montgomery, A. Ajay, C. Finn, P. Abbeel, and S. Levine, "Reset-free guided policy search: Efficient deep reinforcement learning with stochastic initial states," in *Proc. Intl. Conference on Robotics and Automation (ICRA 2017)*. IEEE, 2017, pp. 3373–3380.

[33] A. Sharma, A. Gupta, S. Levine, K. Hausman, and C. Finn, "Autonomous reinforcement learning via subgoal curricula," *Advances in Neural Information Processing Systems*, vol. 34, pp. 18 474–18 486, 2021.

[34] J. Kim, J. Hyeon Park, D. Cho, and H. J. Kim, "Automating reinforcement learning with example-based resets," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6606–6613, 2022.

[35] N. Messikommer, Y. Song, and D. Scaramuzza, "Contrastive initial state buffer for reinforcement learning," in *Proc. Intl. Conference on Robotics and Automation (ICRA 2024)*. IEEE, 2024, pp. 2866–2872.

[36] M. Andrychowicz, D. Crow, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS 2017)*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5048–5058. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/453fadbd8a1a3af50a9df4df899537b5-Abstract.html

[37] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[38] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, pp. 293–321, 1992.

[39] C. S. Păsăreanu, *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security*. Morgan & Claypool Publishers, 2020.

[40] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[41] R. Bellman, "A Markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[42] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[43] V. Mnih, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[44] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. 33nd Intl. Conference on Machine Learning (ICML 2016)*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html

[45] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. 32nd Intl. Conference on Machine Learning (ICML 2015)*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html

[46] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[47] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proc. 35th Intl. Conference on Machine Learning (ICML 2018)*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1582–1591. [Online]. Available: http://proceedings.mlr.press/v80/fujimoto18a.html

[48] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proc. 35th Intl. Conference on Machine Learning (ICML 2018)*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1856–1865. [Online]. Available: http://proceedings.mlr.press/v80/haarnoja18b.html

[49] M. Dadvar, R. K. Nayyar, and S. Srivastava, "Conditional abstraction trees for sample-efficient reinforcement learning," in *Proc. Thirty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI 2023)*, ser. Proceedings of Machine Learning Research, R. J. Evans and I. Shpitser, Eds., vol. 216. PMLR, 2023, pp. 485–495. [Online]. Available: https://proceedings.mlr.press/v216/dadvar23a.html

[50] J. A. Fill, "An interruptible algorithm for perfect sampling via markov chains," in *Proc. 29th Annual Symposium on the Theory of Computing (STOC 1997)*, F. T. Leighton and P. W. Shor, Eds. ACM, 1997, pp. 688–695. [Online]. Available: https://doi.org/10.1145/258533.258664

[51] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-Baselines3: Reliable reinforcement learning implementations," *J. Mach. Learn. Res.*, vol. 22, pp. 268:1–268:8, 2021. [Online]. Available: https://jmlr.org/papers/v22/20-1364.html

[52] M. Varshosaz, M. Ghaffari, E. B. Johnsen, and A. Wąsowski, "Formal specification and testing for reinforcement learning," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, aug 2023. [Online]. Available: https://doi.org/10.1145/3607835

[53] M. Peled, B. Rothenberg, and S. Itzhaky, "SMT sampling via model-guided approximation," in *Proc. 25th Intl. Symposium on Formal Methods (FM 2023)*, ser. Lecture Notes in Computer Science, M. Chechik, J. Katoen, and M. Leucker, Eds., vol. 14000. Springer, 2023, pp. 74–91. [Online]. Available: https://doi.org/10.1007/978-3-031-27481-7_6

[54] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*, ser. Springer Texts in Statistics. Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-1-4757-4145-2