Symbolic State Partitioning
for Reinforcement Learning

Mohsen Ghaffari[1] , Mahsa Varshosaz[1] ,
Einar Broch Johnsen[2] , and Andrzej Wąsowski[1]

[1] ITU, Copenhagen, Denmark
{mohg, mahv, wasowski}@itu.dk
[2] University of Oslo, Oslo, Norway
einarj@ifi.uio.no

Abstract. Tabular reinforcement learning methods cannot operate directly on continuous state spaces. One solution to this problem is to partition the state space. A good partitioning enables generalization during learning and more efficient exploitation of prior experiences. Consequently, the learning process becomes faster and produces more reliable policies. However, partitioning introduces approximation, which is particularly harmful in the presence of nonlinear relations between state components. An ideal partition should be as coarse as possible, while capturing the key structure of the state space for the given problem. This work extracts partitions from the environment dynamics by symbolic execution. We show that symbolic partitioning improves state space coverage with respect to environmental behavior and allows reinforcement learning to perform better for sparse rewards. We evaluate symbolic state space partitioning with respect to precision, scalability, learning agent performance and state space coverage for the learned policies.

Keywords: Reinforcement Learning · Symbolic Execution · State Space Partitioning

1 Introduction

Reinforcement learning is a form of active learning, where an agent learns to make decisions to maximize a reward signal. The agent interacts with an environment and takes actions based on its current state. The environment rewards the agent, which uses the reward value to update its decision-making policy (see Fig. 2). Reinforcement learning has applications in many domains: robotics [22], gaming [44], electronics [15], and healthcare [54]. This method can automatically synthesize controllers for many challenging control problems [43]; however, dedicated approximation techniques, hereunder deep learning, are needed for continuous state spaces. Unfortunately, despite many successes with continuous problems, Deep Reinforcement Learning suffers from low explainability and lack of convergence guarantees. At the same time, discrete (tabular) learning methods have been shown to be more explainable [27,37,51,55] and to yield policies for

which it is easier to assure safety [13, 18, 48], for instance using formal verification [1, 20, 45]. Thus, finding a good state space representation for discrete learning remains an active research area [3, 9, 17, 26, 28, 35, 52].

To adapt a continuous state space for discrete learning, one exploits partial observability to merge regions of the state space into discrete partitions. Each part in a partition represents a subset of the states of the agent. Ideally, all states in a part capture meaningful aspects of the environment—best if they share the same optimal action in the optimal policy. Consequently, a good partitioning is highly problem specific. For instance, in safety critical environments, it is essential to identify small "singularities"—regions that require special handling—even if they are very small. Otherwise, if such regions are included in a larger part, the control policy will not be able to distinguish them from the surrounding parts, leading to high variance in operation time and slow convergence of learning.

The trade-off between the size of the partitions and the optimality and convergence of reinforcement learning remains a challenge [3, 9, 26, 28, 35, 52]. Policies obtained for coarse partitions are unreliable. Large fine partitions make reinforcement learning slow. The dominant methods are *tiling* and *vector quantization* [26, 28, 35, 52]; neither is adaptive to the structure of the state space. They ignore nonlinear dependencies between state components even though quadratic behaviors are common in control systems. So far, the shape of the state space partitions has hardly been studied in the literature.

In this work, we investigate the use of *symbolic execution* to extract approximate adaptive partitions that reflect the problem dynamics. *Symbolic execution* [8, 21] is a classic foundational technique for dynamic program analysis, originating in software engineering and deductive verification research and commonly used for test input generation [49] and in interactive theorem provers (e.g. [2]). A symbolic executor generates a set of *path conditions* (PC), constraints that must hold for each execution path that the program can take. These conditions partition the state space of the executed program into groups that share the same execution path. Our hypothesis is that *the path conditions obtained by symbolic execution of an environment model (the step and reward functions) provide a useful state space partition for reinforcement learning*. The branches in the environment program likely reflect important aspects of the problem dynamics that should be respected by an optimal policy. We test this hypothesis by:

- Defining a symbolic partitioning method and establishing its basic theoretical properties. This method, SymPar, is adaptive to the problem semantics, general (i.e., not developed for a specific problem), and automatic (given a symbolically executable environment program).
- Implementing the method on top of the Symbolic PathFinder, an established symbolic executor for Java programs (JVM programs) [36]
- Evaluating SymPar empirically against other offline and online partitioning approaches, and against deep reinforcement learning methods. The experiments show that symbolic partitioning can allow the agent to learn better policies than with the baselines.

To the best of our knowledge, this is the first time that symbolic execution has been used to breath semantic knowledge into an otherwise statistical reinforcement learning process. We see it as an interesting case of a transfer of concepts from software engineering and formal methods to machine learning. It does break with the tradition of reinforcement learning to treat environments as black boxes. It is however consistent with common practice of using reinforcement learning for software defined problems and with pre-training robotic agents in simulators, as software problems and simulators are amenable to symbolic execution.

The paper proceeds as follows. Section 2 reviews the relevant state of the art. Section 3 recalls the required preliminaries and definitions. Our state space abstraction method is detailed in Sect. 4. In Sect. 5 we present the evaluation design, and then discuss the experiment results (Sect. 6). We discuss the limitations of our method in Sect. 7. Finally, Sect. 8 concludes the paper and presents future work.

2   Related Work

We study partitioning, or a discrete abstraction, of the state space in reinforcement learning by mapping from a continuous state space to a discrete one or by aggregating discrete states. To the best of our knowledge, the earliest use of partitioning, was the BOXES system [29]. The Parti-game algorithm [33] automatically partitions state spaces but applies only to tasks with known goal regions and requires a greedy local controller. While tile coding is a classic method for partitioning [4], it often demands extensive engineering effort to avoid misleading the agent with suboptimal partitions. Lanzi et al. [25] extended learning classifier systems to use tile coding. Techniques such as vector quantization [26, 28, 35, 52] and decision trees [41, 46, 53] lack adaptability to the properties of the state space and may overlook non-linear dependencies among state components. Techniques that gradually refine a coarse partition during the learning process [3, 9, 17, 28, 52] are time-intensive, and require generating numerous parts to achieve better approximations near the boundaries of nonlinear functions. Unlike other methods, SymPar incurs no direct learning cost (it is offline), requires no engineering effort (it is automated), and is not problem specific in contrast to some of the existing techniques (it is general). It produces a partition that effectively captures non-linear dependencies as well as narrow parts, without incurring additional costs or increasing the number of parts at the boundaries.

The concept of bisimulation metrics [11, 12] defines two states as being behaviorally similar if they (1) yield comparable immediate rewards and (2) transition to states that are behaviorally aligned. Bisimulation metrics have been employed to reduce the dimensionality of state spaces through the aggregation of states. However, they have not been extensively explored due to their high computational costs. Moreover, note that bisimulation-minimization-based state-space-abstraction is too fine-grained for the problem at hand. It requires that any states lumped together exhibit the same behavior. This is an unnecessary constraint from the reinforcement learning perspective, which takes no preference over behaviors provided that they lead to the same long-term reward. As long as
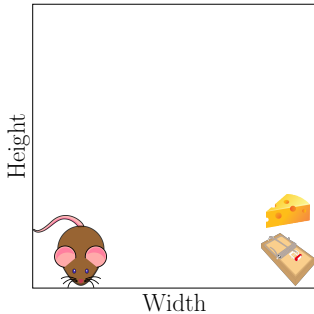
Fig. 1. Navigation environment. A mouse agent in a continuous rectangular board needs to find the cheese, while not stepping on the trap.
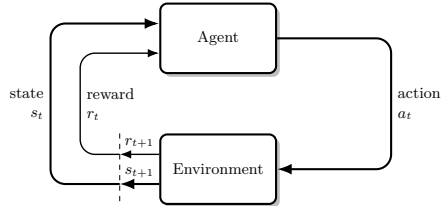
Fig. 2. Reinforcement learning schematic.

the same long-term reward estimate is expected for the same (best) local action in two states, it is theoretically sufficient for the two states to be lumped together. For this reason it is worth exploring weaker principles than bisimulation metrics for reducing dimensionality.

3   Background

Reinforcement Learning (see Fig. 2). A Partially Observable Markov Decision Process is a tuple $\mathcal{M} = (\overline{\mathcal{S}}, \overline{\mathcal{S}}_0, \mathcal{A}, \mathcal{S}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, 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, $\mathcal{S}$ is a finite set of observable states, $\mathcal{O} \in \overline{\mathcal{S}} \to \mathcal{S}$ is a total observation function, $\mathcal{T} \in \overline{\mathcal{S}} \times \mathcal{A} \to$ pdf $\overline{\mathcal{S}}$ is the transition probability function, $\mathcal{R} \in \overline{\mathcal{S}} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\mathcal{F} \in \mathcal{S} \to \{0, 1\}$ is a predicate defining final states. The task is to find a policy $\pi : \mathcal{S} \to \text{Dist}(\mathcal{A})$ that maximizes the expected accumulated reward [43], where Dist is the probability mass function over $\mathcal{A}$.

Example 1 *A mouse sits in the bottom-left corner of a room with dimensions $W \times H$. A mousetrap is placed in the bottom-right corner, and a piece of cheese next to it (Fig. 1). The mouse moves with a fixed velocity in four directions: up, down, left, right. Its goal is to find the cheese but avoid the trap. The states $\overline{\mathcal{S}}$ are ordered pairs representing the mouse's position in the room. The set of initial states $\overline{\mathcal{S}}_0$ is fixed to $(1, 1)$, a Dirac distribution. We define the actions as the set of all possible movements for the mouse: $\mathcal{A} = \{(d, v) : d \in \mathcal{D}, v \in \mathcal{V}\}$, where $\mathcal{D} = \{U, D, R, L\}$ and $\mathcal{V} = \{r_1, r_2, \ldots, r_n \mid r_i \in \mathbb{R}^+\}$. $\mathcal{S}$ can be any partitioning of the room space and $\mathcal{O}$ is the map from the real position of the mouse to the part containing it. Our goal is to find the partition, i.e., $\mathcal{S}$ and $\mathcal{O}$. The reward function $\mathcal{R}$ is zero when mouse finds the cheese, -1000 when the mouse moves into the trap, and -1 otherwise. For simplicity, we let the environment be deterministic, so $\mathcal{T}$ is a deterministic movement of the mouse from a position by a given action to a new position. The final state predicate $\mathcal{F}$ holds for the cheese and trap positions and not otherwise.*

S-ASSIGN	$(x = e, \sigma, k, \phi)$	$\rightarrow$	$(\text{skip}, \sigma[x \mapsto \sigma e], k, \phi)$
S-IF-T	$(\text{if } b: \ s_1 \text{ else}: \ s_2, \sigma, k, \phi)$	$\rightarrow$	$(s_1, \sigma, k, \phi \wedge \ \sigma b)$
S-IF-F	$(\text{if } b: \ s_1 \text{ else}: \ s_2, \sigma, k, \phi)$	$\rightarrow$	$(s_2, \sigma, k, \phi \wedge \ \sigma \neg b)$
S-WHILE-T	$(\text{while } b: \ s, \sigma, k, \phi)$	$\rightarrow$	$(s \ ; \ \text{while } b: \ s, \sigma, k, \phi \wedge \ \sigma b)$
S-WHILE-F	$(\text{while } b: \ s, \sigma, k, \phi)$	$\rightarrow$	$(\text{skip} \ ; \ \sigma, k, \phi \wedge \ \sigma \neg b)$
S-SMLP	$(x \sim \text{rnd}, \sigma, k, \phi)$	$\rightarrow$	$(\text{skip}, \sigma[x \mapsto y_k], k + 1, \phi)$

Fig. 3. Symbolic execution rules for an idealized probabilistic language. Each judgement is a quadruple: the program, the symbolic store (σ), the sample index (k), the current path condition (ϕ).

Partitioning. Partitioning is "the process of mapping a representation of a problem onto a new representation" [16]. A *partition* over a set $\overline{\mathcal{S}}$ of states is a family of sets $p_1, \ldots, p_n \subseteq \overline{\mathcal{S}}$ such that $p_1 \cup \ldots \cup p_n = \overline{\mathcal{S}}$ and $p_i \cap p_j = \emptyset$ for $1 \leq i < j \leq n$. The sets in a partition are called parts. The set of all partitionings is partially ordered: we talk about coarseness (granularity) of partitions. A partition $\mathcal{P}'$ is *coarser* than $\mathcal{P}$ (and $\mathcal{P}$ is *finer* than $\mathcal{P}'$) if $\forall p \in \mathcal{P}. \ \exists p' \in \mathcal{P}'. \ p \subseteq p'$. Recall that the space of partitions is isomorphic to the space of equivalence relations over a set.

Symbolic Execution is a program analysis technique that systematically explores program behaviors by solving symbolic constraints obtained from conjoining the program's branch conditions [21]. Symbolic execution extends normal execution by running the basic operators of a language using symbolic inputs (variables) and producing symbolic formulas as output. A symbolic execution of a program produces a set of *path conditions*—logical expressions that encode conditions on the input symbols to follow a particular path in the program.

For a program over input arguments $I = \{v_1, v_2, \ldots, v_k\}$, a path condition $\phi \in PC(I')$ is a quantifier-free logical formula defined on $I' = \{\vartheta_1, \vartheta_2, \ldots, \vartheta_k\}$, where each symbolic variable ϑ_i represents an initial value for v_i.

We briefly outline a definition of symbolic execution for a minimal language (for more details, see, e.g., [5]). Let V be a set of program variables, Ops a set of arithmetic operations, $x \in V$, $n \in \mathbb{R}$, and $op \in \text{Ops}$. We consider programs generated by the following grammar:

$e ::= x \mid n \mid op(e_1, \ldots, e_n)$
$b ::= \text{True} \mid \text{False} \mid b_1 \text{ AND } b_2 \mid b_1 \text{ OR } b_2 \mid \neg b \mid b_1 \leq b_2 \mid e_1 < e_2 \mid e_1 == e_2$
$s ::= x = e \mid x \sim \text{rnd} \mid s_1; s_2 \mid \text{if } b: \ s_1 \text{ else}: \ s_2 \mid \text{while } b: \ s \mid \text{skip}$

A symbolic store, denoted by σ maps input program variables $I \subseteq V$ to expressions, generated by productions e above. An update to a symbolic store is denoted $\sigma[x = e]$. It replaces the entry for variable x with the expression e. An expression can be interpreted in a symbolic store by applying (substituting) its mapping to the expression syntax (written $e\sigma$).

Figure 3 gives the symbolic execution rules for the above language, in terms of traces (it computes a path condition ϕ for a terminating trace). In the reduction rules, ϕ represents the path condition and k denotes the sampling index. The first rule defines the symbolic assignment. An assignment does not change the path conditions, but updates the symbolic store σ. When encountering conditional statements, the symbolic executor splits into two branches. For the true case (rule S-IF-T) the path condition is extended with the head condition of the branch, for the false case (S-IF-F), the path condition is extended with the negation of the branch condition. Similarly, for a *while* loop two branches are generated, with an analogous effect on path conditions. The last rule executes the randomized sampling statement. It simply allocates a new symbolic variable y_k for the unknown result of sampling, and advances the sampling index [50]. Figure 5 shows the path conditions obtained by applying similar rules to above for the code to the left (Fig. 4). The first path condition $PC^{(U,1)}$ corresponds to the branch where condition `d==1` is true in the program.

The above rules can be used to prove basic properties of symbolic execution. For example, since branch conditions are always introduced in dual rules, the path conditions of a program are mutually exclusive [5].

Practical symbolic executors have been realized for full scale programming languages. Although we defined symbolic execution at the level of syntax, the two most popular symbolic executors operate on compiled bytecode [6,36]. In presence of loops and recursion, symbolic execution does not terminate. To halt symbolic execution, we can set a predefined timeout in terms of an iteration limit or a program statement limit. This produces an approximation of the set of path conditions.

4   Partitioning Using Symbolic Execution

We present the idea of symbolic partitioning using a single agent with the environment modeled as a computer program. The program (Env) is implementing a single step-transition ($\mathcal{T}$) in the environment with the corresponding reward ($\mathcal{R}$). We use symbolic execution to analyze the environment program Env, then partition the state space using the obtained path conditions. The partition serves as the observation function $\mathcal{O}$. The entire process is automatic and generic—we can follow the same procedure for all problems.

Example 2 *Figure 4 shows the environment program for the 10×10 navigation problem (Example 1). For simplicity, we assume the agent can move one unit in each direction, so $\mathcal{V} = \{1\}$ and $\mathcal{A} = \{U, D, R, L\} \times \mathcal{V}$. The path conditions in Fig. 5 are obtained by symbolically executing the step and reward functions using symbolic inputs x and y and a concrete input from $\mathcal{A}$. Using path conditions in partitioning requires a translation from the symbolic executor syntax into the programming language used to implement the partitioning process, as the executor will generate abstract value names.*

A good partition maintains the Markov property, so that the same action is optimal for all unobservable states abstracted by the same part. Unfortunately,

```
1   W = 10 # Width
2   H = 10 # Height
3   def step(x, y, d, v):
4     if d == 1: # UP
5       if y < H:
6         return x, y+v
7     if d == 2: # DOWN
8       if y > 1:
9         return x, y-v
10    if d == 3: # LEFT
11      if x > 1:
12        return x-v, y
13    if d == 4: # RIGHT
14      if x < W:
15        return x+v, y
16    return x, y
17
18  def reward(x, y, d, v):
19    if x == W:
20      if y == 2:
21        return 0.0 # Cheese
22      if y == 1:
23        return -1000.0 # Trap
24    return -1.0
25
```

$PC^{(U,1)}$

5, 19, 20	$y < 10 \wedge x = 10 \wedge y+1 = 2$
5, 19, !20	$y < 10 \wedge x = 10 \wedge y+1 \neq 2$
5, !19	$y < 10 \wedge x \neq 10$
!5, 19	$y \geq 10 \wedge x = 10$
!5, !19	$y \geq 10 \wedge x \neq 10$

$PC^{(D,1)}$

8, 19, 20	$y > 1 \wedge x = 10 \wedge y-1 = 2$
8, 19, !20, 22	$y > 1 \wedge x = 10 \wedge y-1 \neq 2 \wedge y-1 = 1$
8, 19, !20, !22	$y > 1 \wedge x = 10 \wedge y-1 \neq 2 \wedge y-1 \neq 1$
8, !19	$y > 1 \wedge x \neq 10$
!8, 19, !20, 22	$y \leq 1 \wedge x = 10 \wedge y = 1$
!8, !19	$y \leq 1 \wedge x \neq 10$

$PC^{(L,1)}$

11, !19	$x > 1 \wedge x-1 \neq 10$
!11, !19	$x \leq 1 \wedge x \neq 10$

$PC^{(R,1)}$

14, 19, 20	$x < 10 \wedge x+1 = 10 \wedge y = 2$
14, 19, !20, 22	$x < 10 \wedge x+1 = 10 \wedge y \neq 2 \wedge y = 1$
14, 19, !20, !22	$x < 10 \wedge x+1 = 10 \wedge y \neq 2 \wedge y \neq 1$
14, !19	$x < 10 \wedge x+1 \neq 10$
!14, 19, 20	$x \geq 10 \wedge x = 10 \wedge y = 2$
!14, 19, !20, 22	$x \geq 10 \wedge x = 10 \wedge y \neq 2 \wedge y = 1$
!14, 19, !20, !22	$x \geq 10 \wedge x = 10 \wedge y \neq 1 \wedge y \neq 1$

Fig. 4. The environment program $(\mathcal{T}, \mathcal{R})$ for the navigation problem (Fig. 1).

Fig. 5. Path conditions collected by symbolic execution. The numbers (to the left) refer to line numbers in the program of Fig. 4.

this means that a good partition can be selected only once we know a good policy—after learning. To overcome this, SymPar heuristically bundles states into the same part if they induce the same execution path in the environment program. We use an off-the-shelf symbolic executor to extract all possible path conditions from Env, by using $\overline{\mathcal{S}}$ as symbolic input and actions from $\mathcal{A}$ as concrete input. The result is a set PC of path conditions for each concrete action: $PC = \{PC^{a_0}, PC^{a_1}, \ldots, PC^{a_m}\}$, where $PC^a = \{PC_0^a, PC_1^a, \ldots, PC_{k_a}^a\}$. The set PC^a contains the path conditions computed for action a, and k_a is the number of all path conditions obtained by running Env symbolically, for a concrete action a.

Running the environment program for any concrete state satisfying a condition PC_i^a with action a will execute the same program path. However, the partitioning for reinforcement learning needs to be action independent (the same for all actions). So the obtained path conditions cannot be used directly for the partitioning. Consider $PC_i^{a_1} \in PC^{a_1}$ and $PC_j^{a_2} \in PC^{a_2}$, arbitrary path conditions for some actions a_1, a_2. To make sure that the same program path will be taken from a concrete state for both actions, we need to create a part that corresponds to the intersection of both path conditions: $PC_i^{a_1} \wedge PC_j^{a_2}$. In general, each set in PC
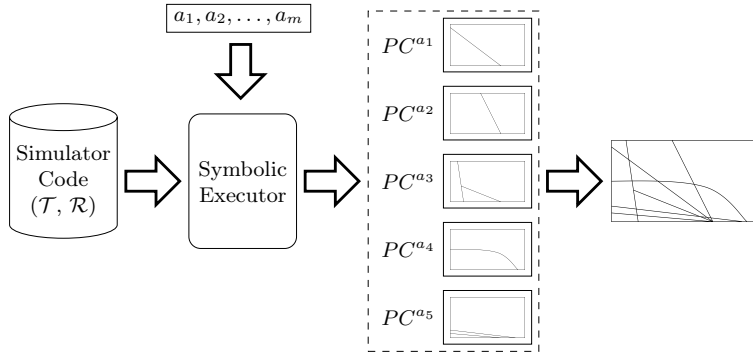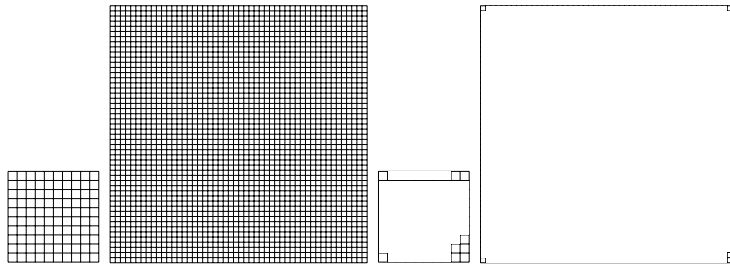
Fig. 6. Overview of SymPar.

Fig. 7. Using tile coding (left) and SymPar (right) for 10×10 and 50×50 navigation

defines partitions of the state space for different actions. To make them compatible, we need to compute the unique coarsest partition finer than those induced by the path conditions for any action, which is a standard operation in order theory [10]. In this case, this amounts to computing all intersections of partitions for all actions, and removing the empty intersections using an SMT check.

The process of symbolic state space partitioning is summarized in Fig. 6 and Alg. 1. SymPar executes the environment program symbolically. For each action, a set of path conditions is collected. In the figure, $|\mathcal{A}| = 5$ and, accordingly, five sets of path conditions are collected (shown as rectangles). Each rectangle is divided into a group of regions, each of which maps to a path condition. Thus, the rectangles illustrate the state space that is discretized by the path conditions. Note that the border of each region can be a unique path condition (an expression with equality relation) or a part of neighbour regions (an expression with inequality relation). The final partition is shown as another rectangle that contains the overlap between the regions from the previous step.

Example 3 *Figure 7 (left) shows the partitioning of the Navigation problem using tile coding [4] for two room sizes. Numerous cells share the same policy, prompting the question of why they should be divided. SymPar achieves a much*

Algorithm 1 Partitioning with Symbolic Execution (SymPar)

Input: Env, $\mathcal{A}$
Output: $\mathcal{P}$ (a partitioning of $\overline{\mathcal{S}}$)

1: $PC \leftarrow \emptyset$
2: **for** $a \in \mathcal{A}$ **do**
3:     $PC^a, \Psi \leftarrow$ SymExec (Env, symbolic $\overline{\mathcal{S}}$, concrete a)   // Ψ is the set of sampling variables
4:     Add distribution support constraints for all variables $\overline{\mathcal{S}} \in \Psi$ to PC^a
5:     Existentially quantify all sampling variables in PC^a // may introduce overlaps of conditions
6:     $PC'^a \leftarrow \emptyset$
7:     **for** $p, q \in PC^a$ **do if SAT** $(p \wedge q)$ $PC'^a \leftarrow PC'^a \cup \{p \wedge q\}$
8:     $PC^a \leftarrow PC'^a$
9: $\mathcal{P} \leftarrow PC^{\mathcal{A}[0]}$
10: **for** $a \in \mathcal{A} - \{\mathcal{A}[0]\}$ **do**
11:     $\mathcal{P}' \leftarrow \emptyset$
12:     **for** $p \in \mathcal{P}, q \in PC^a$ **do if SAT** $(p \wedge q)$ **then** $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{p \wedge q\}$
13:     $\mathcal{P} \leftarrow \mathcal{P}'$
14: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\sim \bigvee_{p \in \mathcal{P}} p\}$
15: **return** $\mathcal{P}$

---

coarser partition than the initial tiling, by discovering that for many tiles the dynamics is the same (right).

We handle stochasticity of the environment by allowing environment programs to be probabilistic and then following rule S-SMLP in symbolic execution (Fig. 3). We introduce a new symbolic variable whenever a random variable is sampled in the program [23, 50]. Consequently, our path conditions also contain these sampling variables. To make the process more reliable, one can generate constraints, limiting them to the support of the distribution. For example, for sampling from a uniform distribution $U[\alpha, \beta]$, the sampling variable n_v is subject to two constraints: $n_v \geq \alpha$ and $n_v \leq \beta$. In order to be able to compute the partition over state variables only, as above, we existentially quantify the sampling variables out. This may introduce overlaps between the conditions, so we compute their intersection at this stage before proceeding (see lines 6-9 in Alg. 1).

Since the entire setup uses logical representations and an SMT solver, we exploit it further to generate witnesses for all parts, even the smallest ones. We use them to seed reinforcement learning episodes, ensuring that each part has been visited at least once. Consequently the agent is guaranteed to learn from all the paths in the environment program. This can be further improved by constraining with a reachability predicate (not used in our examples).

Properties of SymPar. SymPar on the specifics of the environment implementation. Distinct implementations of the simulated environment may result in different partitioning outcomes for a given problem. On the other hand, the

outcome is independent of the size of state space. Recall that in Fig. 7 (right) the number of parts is the same for the small and the large room.

A partition is by definition total: every state in the input space is included in a part, ensuring the entire state space is fully covered. As symbolic execution does not terminate for many interesting programs (programs with loops have infinitely many symbolic paths), one typically stops symbolic execution after a designated timeout. This can leave a part of the state space unexplored. Hence, a partitioning obtained from path conditions generated by symbolic execution may not cover all the state space. SymPar makes the obtained partition total by adding the complement of the union of the computed partitions, to cover for the unexplored paths (l. 14 in Alg. 1). Thus, the following property holds:

Theorem 1. *The set $\mathcal{P}$ obtained in Alg. 1 is a partition (i.e., it is total): $\forall \overline{s} \in \overline{\mathcal{S}} \; \exists! \, \mathcal{P}_0 \in \mathcal{P} \cdot \overline{s} \in \mathcal{P}_0$.*

The cost of SymPar amounts to exploring all paths in the program symbolically and then computing the coarsest partition. The symbolic execution involves generating a number of paths exponential in the number of branch points in the program (and at each branch one needs to solve an SMT problem—which is in principle undecidable, but works well for many practical problems). A practical approach is to bound the depth of exploration of paths by symbolic executor for more complex programs. Computing the coarsest partition requires solving $|\mathcal{P}|^{|\mathcal{A}|}$ number of SMT problems where $|\mathcal{P}|$ is the upper bound on the number of parts (symbolic paths) and $|\mathcal{A}|$ is the number of actions. The other operations involved in this process such as computing and storing the path conditions in the required syntax are polynomial and efficient in practice.

Theorem 2. *Let PC^a be the set of path conditions produced by SymPar for each of the actions $a \in \mathcal{A}$. The size of the final partition $\mathcal{P}$ returned by SymPar is bounded from below by each $|PC^a|$ and from above by $\prod_{a \in \mathcal{A}} |PC^a|$.*

The theorem follows from the fact that $\mathcal{P}$ is finer than any of the PC^as and the algorithm for computing the coarsest partition finer than a set of partitions can in the worst case intersect each part in each set PC^a with all the parts in the partitions of the other actions.

Note that SymPar is a heuristic and approximate method. To appreciate this, define the optimal partition to be the unique partition in which each part contains all states with the same action in the optimal policy (the optimal partition is an inverse image of the optimal policy for all actions). The partitions produced by SymPar are neither always coarser or always finer than the optimal one. This can be shown with simple counterexamples. For an environment with only one action, the optimal partition has only one part as the optimal policy maps the same action for all states. But Sympar will generate more than one part (a finer partition) if the simulation program contains branching. For problems without branching in the simulator such as cart pole problem, Sympar produces only one part. However, the optimal partition contains more than one part as optimal actions for all states in the state space are not the same. To understand the significance of this approximation in practice, we evaluate SymPar empirically against the existing methods.

5   Evaluation Setup

The partitioning of the state space faces a trade-off: on one hand, the granularity of the partition should be fine enough to distinguish crucial differences between states in the state space. On the other hand, this granularity should be chosen to avoid a combinatorial explosion, where the number of possible parts becomes unmanageably large. Achieving this balance is essential for efficient and effective learning. In this section, we explore this trade-off and evaluate the performance of our implementation in SymPar empirically by addressing the following research questions:

RQ1 *How much smaller are the SymPar partitions compared to other methods, and how do these smaller partitions impact learning performance?*
RQ2 *How does the granularity of the partition affect the learning performance?*
RQ3 *How does SymPar scale with increasing state space sizes?*
RQ4 *How well does SymPar group together behaviorally similar states?*

We compare SymPar to CAT-RL [9] (online partitioning) and with tile coding techniques (offline partitioning) for different examples [43]. Tile coding is a classic technique for partitioning. It splits each feature of the state into rectangular tiles of the same size. Although there are new problem specific versions, we opt for the classic version due to its generality.

To answer **RQ1**, we measure (a) the *size of partition*, (b) the *failure and success rates* and (c) the *accumulated reward* during learning. Being offline, our approach is hard to compare with online methods, since the different parameters may affect the results. Therefore, we separate the comparison with offline and online algorithms. For offline algorithms, we first find the number of abstract states using SymPar and partition the state space using tile coding accordingly (i.e., the number of tiles is set to the smallest square number greater than the number of parts in SymPar's partition). Then, we use standard Q-learning for these partitions, and compare their performance. For online algorithms, we compute the running time for SymPar and its learning process, run CAT-RL for the same amount of time, and compare their performance. Obviously, if the agent observes a failing state, the episode stops. This decreases the running time. Finally, we compare the accumulated reward for SymPar with well-known algorithms DQN [31], A2C [30], PPO [40], using the Stable-Baselines3 implementations[3] [38]. These comparisons are done for two complementary cases: (1) randomly selected states and (2) states that are less likely to be chosen by random selection. The latter are identified by SymPar's partition. We sample states from different parts obtained by SymPar and evaluate the learning process by measuring the accumulated reward.

To answer **RQ2**, we create different learning problems with various partitioning granularities by changing the search depth for the symbolic execution. We then compare the maximum accumulated reward of the learned policy to gain an understanding of the learning performance for the given abstraction.

To answer **RQ3**, we compare the number of parts when increasing the state space of problems.

---

[3] https://github.com/DLR-RM/stable-baselines3

	SymPar					Tile Coding					CAT-RL			
	$|\mathcal{S}|$	Succ	Fail	T_{out}	Opt	$|\mathcal{S}|$	Succ	Fail	T_{out}	Opt	$|\mathcal{S}|$	Succ	Fail	T_{out}
	(#)	(%)	(%)	(%)	(%)	(#)	(%)	(%)	(%)	(%)	(#)	(%)	(%)	(%)
SM	33	74.9	<0.1	25.0	5.0	10^4	6.0	7.1	86.9	0.0	154	63.0	5.0	32.0
MAN	130	5.8	82.6	11.6	0.0	10^4	0.0	99.6	0.4	0.0	620	0.0	74.7	25.3
WW 1	73	18.4	0.0	81.6	2.1	8^4	9.6	0.0	90.4	0.0	157	2.7	0.0	97.3
WW 2	52	37.3	22.9	39.8	4.2	64	19.1	33.2	47.7	0.0	22	14.5	30.2	55.3
Nav	51	13.2	4.8	82.0	<0.1	64	0.0	0.0	100.0	0.0	100	1.7	1.5	96.8
BC	81	89.1	10.9	0.0	29.8	81	82.0	18.0	0.0	14.9	127	34.0	66.0	0.0
MC	70	82.2	0.0	17.8	61.3	81	59.4	0.0	40.6	14.7	16	78.7	0.0	21.3
RW	184	61.2	11.1	27.7	44.0	196	6.5	5.1	88.4	<0.1	52	41.8	31.8	26.4

Table 1. Partitions size and learning performance. Discrete cases above bar, continuous below. **SM**, **MAN**, **WW 1**, **WW 2**, **Nav**, **BC**, **MC**, **RW**, respectively, stand for Simple Maze, Multi-Agent Navigation, Wumpus World 1, Wumpus World 2, Navigation, Braking Car, Mountain Car, Random Walk.

To answer **RQ4**, we select five random parts from the partition obtained by SymPar, and five random concrete states from each part. Then, we feed the concrete states as initial states to RL, and compute the accumulated reward using the policy obtained from a trained model, assuming the training converged to the optimal policy. This way we can check how different the concrete states are with regard to performance.

Test Problems. The **Navigation** problem with a room (continuous) size of 10×10. The **Simple Maze** is a discrete environment (100×100) including blocks, goal and trap, in which a robot tries to find the shortest and safest route to the goal state [43]. **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 [47]. The **Multi-Agent Navigation** environment (10×10 grid) contains two agents attempting to find safe routes to a goal location. They must arrive to the goal position at the same time [42]. The **Mountain Car** aims to learn how to obtain enough momentum to move up a steep slope [32]. The **Random Walk** in continuous space is an agent with noisy actions on an infinite line [43]. The agent aims to avoid a hole and reach the goal region. **Wumpus World** [39] is a grid world (1: 64×64, 2: 16×16) in which the agent should avoid holes and find the gold.

6   Results

6.1   RQ1: Partition Size

Table 1 shows that SymPar consistently outperforms both tile coding (offline) and CAT-RL (online) on discrete state space cases in terms of success and failure

rates, and reduces number of timeouts ($\mathbf{T_{out}}$) during learning in majority of cases. Also, the agents using SymPar partitions show better performance in terms of the percentage of episodes during the learning in which they achieve the maximum accumulated reward in comparison to tile coding partitions ($\mathbf{Opt}$), cf. Tbl. 1. Note that in Tbl. 1, the size of partitions is substantially biased in favour of tiling. Nevertheless, SymPar enables better learning. In Tbl. 1, CAT-RL obtains smaller partitions for $\mathbf{WW2}$, $\mathbf{MC}$, and $\mathbf{RW}$ in the same amount of time as SymPar. However, the results for CAT-RL show worse learning performance in comparison to SymPar for these cases as demonstrated by failure and success rates (reporting $\mathbf{Opt}$ is not supported by the available CAT-RL implementations, and would require a modification of that method). The small partition size in CAT-RL can be explained by its operational mechanism, which involves initialising the agent from a small set. This approach prevents divergence and ensures the number of parts remains constant. Subsequently, CAT-RL implements a policy, aiming to identify the goal state and partitions based on the observations it gathers. Hence, in scenarios where the initial states are not limited and the policies are not goal-oriented, the number of parts will increase. For instance, we have evaluated CAT-RL for mountain car in scenarios where exploration is unrestricted, and the number of parts for a given number of episodes has increased to 302. For the other test problems, SymPar achieves better results than CAT-RL in both the partition size and learning performance.

For randomly selected states, the three left plots in Fig. 8, show that the agents trained by SymPar obtain a better normalized cumulative reward and subsequently converge faster to a better policy than the best competing approaches. The three right plots in the figure show the accumulated reward when starting from unlikely states (small parts) for the best competing approaches. Here, we expect to observe a good policy from algorithms that capture the dynamics of environment. Interestingly, the online technique CAT-RL struggles when dealing with large sets of initial states. This can be seen in, e.g., the training for Braking Car, where each episode introduces new positions and velocities.

6.2   RQ2: Granularity vs Learning

The plots in Fig. 9 shows that a higher granularity of partitions yields a higher accumulated reward achieved with the optimal policy. To be more specific, increasing the depth of search for symbolic execution would result in additional constraints on each PC, consequently a finer partition. Then, given sufficient repetition of RL algorithms, finer partitions can yield a better policy for each part, due to a reduction in the variance of optimal policies across states in the part. This results in a higher accumulated reward when both partitions are evaluated for the same states.

The plots in Fig. 10 show the shapes of partitions obtained by SymPar for Braking Car and Simple Maze. The first plot represents different parts with different colors. Notably, the green and purple parts depict partition expressions that contain a non-linear relation between the components of the state space (position and velocity). Besides, close to the x-axis, narrow parts are discernible,
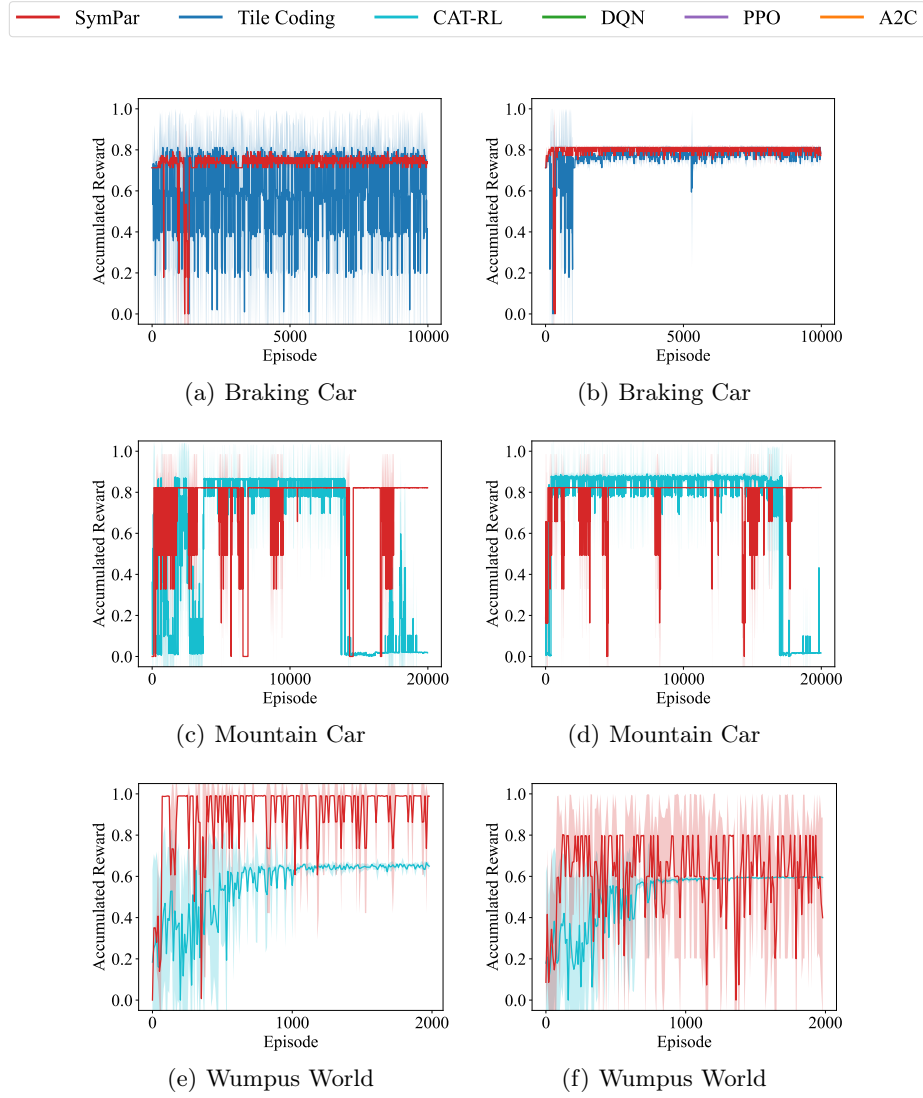
(a) Braking Car

(b) Braking Car

(c) Mountain Car

(d) Mountain Car

(e) Wumpus World

(f) Wumpus World

Fig. 8. Normalized cumulative reward per episode while evaluating ten random states (Left), and less likely states (Right). The best approach for each case is shown.

depicted in yellow and pink. To illustrate the partitions obtained for Simple Maze, the expressions are translated into a 10×10 grid. The maze used for Fig. 10(b) differs from the one before, by including additional obstacles in the environment. These two visualizations shed light on the intricacies of state space partitioning and hint at the logical explainability of the partitions obtained by SymPar.
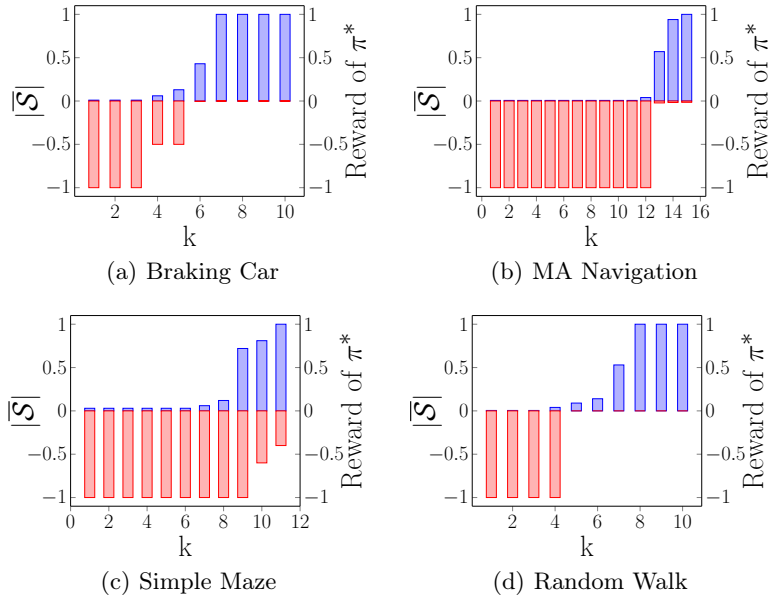
(a) Braking Car

(b) MA Navigation

(c) Simple Maze

(d) Random Walk

Fig. 9. Normalized granularity of states and its performance for symbolic execution with search depth k.
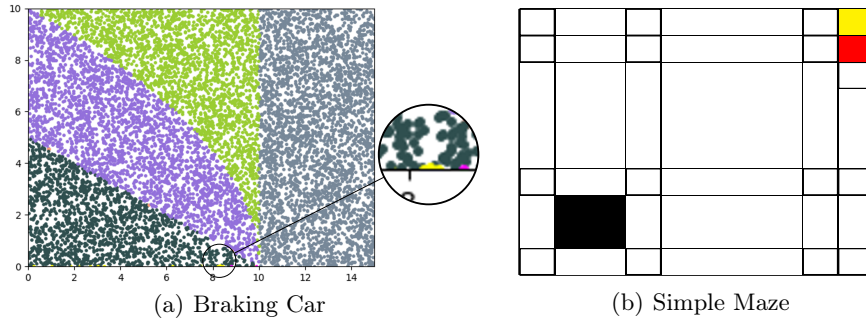

(a) Braking Car

(b) Simple Maze

Fig. 10. Partitions with SymPar for the Braking Car and Simple Maze.

6.3   RQ3: Scalability

Table 2 shows that the number of parts in SymPar partitions is independent of the size of the state space. However, this does not imply the universal applicability of the same partition across different sizes. The conditions specified within the partitions are size-dependent. Consequently, when analyzing environments with different sizes for a given problem, running SymPar is necessary to ensure the appropriate partition, even though the total number of parts remains the same.

	$|\overline{\mathcal{S}}|$	$|\mathcal{S}|$		$|\overline{\mathcal{S}}|$	$|\mathcal{S}|$		$|\overline{\mathcal{S}}|$	$|\mathcal{S}|$
Simple Maze	10×10 $10^2 \times 10^2$ $10^3 \times 10^3$	33 33 33	**Wumpus World**	64×64 $10^2 \times 10^2$ $10^3 \times 10^3$	73 73 73	**Navigation**	10×10 $10^2 \times 10^2$ $10^3 \times 10^3$	51 51 51

Table 2. Size of state space and partition for test problems.

	$\mathcal{P_1}$	$\mathcal{P_2}$	$\mathcal{P_3}$	$\mathcal{P_4}$	$\mathcal{P_5}$
BC	$-0.05 \pm 0.0\%$	$-0.01 \pm 0.0\%$	$-0.5 \pm 0.0\%$	$-10.0 \pm 0.0\%$	$-10.01 \pm 0.0\%$
MC	$996.1 \pm 0.1\%$	$975.5 \pm 0.2\%$	$979.04 \pm 0.2\%$	$986.7 \pm 0.2\%$	$981.6 \pm 0.2\%$
WW 1	$486.8 \pm 0.3\%$	$490.0 \pm 0.2\%$	$477.6 \pm 0.2\%$	$475.0 \pm 0.0\%$	$495.8 \pm 0.1\%$

Table 3. Assessment of similarity of concrete states within parts. **BC**, **MC**, **WW 1**, respectively, stand for Braking Car, Mountain Car, Wumpus World 1.

	Off	Auto	Dyn	NonL	NarrP	SInd
SymPar	✓	✓	✓	✓	✓	✓
CAT-RL	✗	✓	✓	✗	✗	✗
Tiling	✓	✗	✗	✗	✗	✗

Table 4. Capabilities and properties.

6.4   RQ4: Partitioning as an Abstraction

Table 3 presents the variance in accumulated rewards for concrete states across various parts. The findings demonstrate a notable consistency in accumulated rewards among states within the same part, indicating minimal divergence. This is particularly evident when the mean and normalized standard deviation are compared, which demonstrates that the standard deviation is considerably smaller in relation to the mean accumulated reward.

Summary. Our experiments show distinct advantages of SymPar over the other approaches, cf. Tbl. 4. It is an offline (**Off**) automated (**Auto**) approach, which captures the dynamics of the environment (**Dyn**), and maps the nonlinear relation between components of the state into their representation (**NonL**). SymPar can detect narrow parts (**NarrP**) without excessive sampling and generates a logical partition that is independent of the specific size of the state space (**SInd**). This comprehensive comparison underscores the robust capabilities of SymPar across various dimensions, positioning it as a versatile and powerful approach compared to CAT-RL and Tile Coding.

Threats to Internal Validity. The data produced in response to RQ1 and RQ2 may be incorrectly interpreted as suggesting existence of a correlation between the size of partitions and the effectiveness of learning. No such obvious correlation exists: too small, too large, and incorrectly selected partitions hamper learning.

We cannot claim any such correlation. We merely report the size of the partitions and the performance of learning for the selected cases.

While we study the impact of the state space size on SymPar (RQ3), one should remember that there is no strong relationship between the size and the complexity of the state space. In general, the complexity (the branching of the environment model) has a dominant effect on the performance of symbolic execution.

We assumed that two states are similar if they have yield similar accumulated reward in the obtained policy (RQ4). First, note that we have no guarantee that the used policy is optimal, although the plots suggest convergence. Second, a more precise, but also more expensive, alternative would be to compute the optimal policy for each of the two states separately (taking them out of partitions). This could lead to higher and different reward values. However, even this would not guarantee the reliability of the estimated state values, as the hypothetical optimal accumulated reward requires representing the optimal policy precisely for all reachable states, which is infeasible in a continuous state space with a probabilistic environment.

Threats to External Validity. The results of experiments are inherently not generalizable, given that we use a finite set of cases. However, the selected cases do cover a range of situations: discrete and continuous state spaces, deterministic and non-deterministic environments, as well as single- and multi-agent environments.

Technical Details. The implementation of SymPar (will be publicly available upon the acceptance) uses Symbolic PathFinder[4] [36] as its symbolic executor, Z3[5] [34] as its main SMT-Solver and the SMT-solver DReal[6] [14] to handle non-linear functions such as trigonometric functions.

7   Discussion and Limitations

SymPar is not limited to reinforcement learning. Theoretically, it could be applied with traditional solving techniques for MDPs. However, this would require efficient methods for extracting MDP models from simulator code.

SymPar uses environments that are implemented as programs, so they are formally specified. This may suggest that one can obtain the policies analytically, not through (statistical) RL. However, many problems exist which we can formulate as programs, but those semantics are too complex to handle with precise analytical methods by solving the derived MDPs. For example, consider an autonomous drone delivery system in an urban setting that needs to transport packages efficiently, while avoiding static (buildings) and dynamic (other drones) obstacles. The urban environment can be modeled with precise geometry and established laws of physics that govern drone flight dynamics. Weather predictions and obstacle patterns can frequently be pre-simulated. Despite the availability of an exact

---

[4] https://github.com/SymbolicPathFinder
[5] https://github.com/Z3Prover/z3
[6] https://github.com/dreal/dreal4

environment model, reinforcement learning remains the preferred solution due to its ability to scale to this complexity, which analytical methods cannot [7, 19, 24].

Simulations often rely on simplifying assumptions to make them computationally feasible. If these assumptions abstract from critical details, the simulations might not be fully transparent or interpretable. Even in these cases, the efficacy of SymPar in achieving effective partitions, and the capacity of RL to identify optimal policies, remains valid. While we did not undertake a direct experiment for this scenario, the experiments for RQ2 can serve as a surrogate evidence. Limiting the depth of search for symbolic execution may generate a coarser partition than from a fully analyzed the program, which while not exactly the same, is similar to using a more abstract program for partitioning. These experiments indicate that if a simplified model of the environment is used, SymPar could still generate a partition that can be used for more realistic environment models.

In concurrency theory, lumping or bisimulation minimization is sometimes used as a partitioning technique. Note that bisimulation minimization is presently not possible for environment models expressed as computer programs. We would need symbolic bisimulation-minimization methods. Also, note that bisimulation induces a finer partitioning than we need: it puts in a single equivalence class all states that are externally indistinguishable, while we only need to unify states that share the same optimal action in one step. In contrast, symbolic execution performs a mixed syntactic-semantic decomposition of the input state space by means of path conditions. This process is mainly driven by the syntax of the program, yet it is semantically informed via the branch conditions. The obtained partition might be unsound from the bisimulation perspective, but it tends to produce coarser partitions.

SymPar analyzes single step executions of the environment. There are however problems where the interesting behaviors are observed only over a sequence of decisions. For example, the dynamics of Cart Pole [43] is described by a continuous formula over its position and velocity along with the angle and angular velocity of the pole. There is, in fact, no interesting explicit branching—the path conditions found by symbolic execution are trivial. SymPar is better suited for problems with explicit branching in the environment dynamics. At the same time, excessive branching can hamper its efficiency. In these cases, choosing a reasonable depth may achieve a partition that is sufficiently good while controlling its size (Fig. 9).

8   Conclusion

SymPar is a new generic and automatic offline method for partitioning state spaces in reinforcement learning based on a symbolic analysis of the environment's dynamics. In contrast to related work, SymPar's partitions effectively capture the semantics of the environment. SymPar accommodates non-linear environmental behaviors by using adaptive partition shapes, instead of rectangular tiles. Our experiments demonstrate that SymPar improves state space coverage with respect to environmental behavior and allows reinforcement learning to better handle with sparse rewards. However, since SymPar analyzes the simulator of the environment, it is sensitive to the implementation of the environment model.

The performance of the underlying tools, including the symbolic executor and SMT solvers, also affect the effectiveness of SymPar for complex simulators with long execution paths. In the future, we would like to address these limitations and consider using symbolic execution also for online partitioning.

Data Availability Statement. The source code of SymPar, the benchmark items, the evaluation results and instructions for reproduction are available online via DOI 10.5281/zenodo.14620119.

References

1. Adelt, J., Herber, P., Niehage, M., Remke, A.: Towards safe and resilient hybrid systems in the presence of learning and uncertainty. In: Proc. 11th Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (ISoLA 2022). Lecture Notes in Computer Science, vol. 13701, pp. 299–319. Springer (2022). https://doi.org/10.1007/978-3-031-19849-6_18
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Akrour, R., Veiga, F., Peters, J., Neumann, G.: Regularizing reinforcement learning with state abstraction. In: Proc. Intl. Conf. on Intelligent Robots and Systems (IROS). pp. 534–539. IEEE (2018)
4. Albus, J.S.: Brains, behavior, and robotics. BYTE Books (1981)
5. de Boer, F.S., Bonsangue, M.M.: Symbolic execution formally explained. Formal Aspects Comput. **33**(4-5), 617–636 (2021). https://doi.org/10.1007/S00165-020-00527-Y
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI 2008). pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
7. Chen, X., Wang, H., Li, Z., Ding, W., Dang, F., Wu, C., Chen, X.: Deliversense: Efficient delivery drone scheduling for crowdsensing with deep reinforcement learning. In: Adjunct Proceedings of the 2022 ACM International Joint Conference on Pervasive and Ubiquitous Computing and the 2022 ACM International Symposium on Wearable Computers. pp. 403–408 (2022)
8. Clarke, L.A.: A program testing system. In: Proc. 1976 Annual Conf. pp. 488–491. ACM (1976). https://doi.org/10.1145/800191.805647
9. Dadvar, M., Nayyar, R.K., Srivastava, S.: Conditional abstraction trees for sample-efficient reinforcement learning. In: Proc. 39th Conf. on Uncertainty in Artificial Intelligence. Proc. Machine Learning Research, vol. 216, pp. 485–495. PMLR (2023)
10. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press, Cambridge (1990), http://www.worldcat.org/search?qt=worldcat_org_all&q=0521367662

11. Ferns, N., Panangaden, P., Precup, D.: Metrics for finite Markov decision processes. In: UAI. vol. 4, pp. 162–169 (2004)
12. Ferns, N., Panangaden, P., Precup, D.: Bisimulation metrics for continuous Markov decision processes. SIAM Journal on Computing **40**(6), 1662–1714 (2011)
13. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In: Proc. 32nd Conf. on Artificial Intelligence (AAAI-18). pp. 6485–6492. AAAI Press (2018). https://doi.org/10.1609/AAAI.V32I1.12107
14. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. 24th Intl. Conf. on Automated Deduction (CADE-24). Lecture Notes in Computer Science, vol. 7898, pp. 208–214. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_14
15. Ghaffari, M., Afsharchi, M.: Learning to shift load under uncertain production in the smart grid. Intl. Transactions on Electrical Energy Systems **31**(2), e12748 (2021)
16. Giunchiglia, F., Walsh, T.: A theory of abstraction. Artificial intelligence **57**(2-3), 323–389 (1992)
17. Jaeger, M., Jensen, P.G., Larsen, K.G., Legay, A., Sedwards, S., Taankvist, J.H.: Teaching Stratego to play ball: Optimal synthesis for continuous space MDPs. In: Proc. 17th Intl. Symposium on Automated Technology for Verification and Analysis (ATVA 2019). Lecture Notes in Computer Science, vol. 11781, pp. 81–97. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_5
18. Jansson, A.D.: Discretization and representation of a complex environment for on-policy reinforcement learning for obstacle avoidance for simulated autonomous mobile agents. In: Proc. 7th Intl. Congress on Information and Communication Technology. Lecture Notes in Networks and Systems, vol. 464, pp. 461–476. Springer (2023)
19. Jevtić, Đ., Miljković, Z., Petrović, M., Jokić, A.: Reinforcement learning-based collision avoidance for uav. In: 2023 10th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN). pp. 1–6. IEEE (2023)
20. Jin, P., Tian, J., Zhi, D., Wen, X., Zhang, M.: Trainify: A CEGAR-driven training and verification framework for safe deep reinforcement learning. In: Proc. 34th Intl. Conf. on Computer Aided Verification (CAV 2022). Lecture Notes in Computer Science, vol. 13371, pp. 193–218. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_10
21. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
22. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: A survey. The Intl. Journal of Robotics Research **32**(11), 1238–1274 (2013)
23. Kozen, D.: Semantics of probabilistic programs. In: Proc. 20th Annual Symposium on Foundations of Computer Science (SFCS 1979). pp. 101–114. IEEE Computer Society (1979). https://doi.org/10.1109/SFCS.1979.38
24. Kretchmara, R.M., Young, P.M., Anderson, C.W., Hittle, D.C., Anderson, M.L., Delnero, C.C.: Robust reinforcement learning control. In: Proceedings of the 2001 American Control Conference.(Cat. No. 01CH37148). vol. 2, pp. 902–907. IEEE (2001)
25. Lanzi, P.L., Loiacono, D., Wilson, S.W., Goldberg, D.E.: Classifier prediction based on tile coding. In: Proc. Genetic and Evolutionary Computation Conf. (GECCO 2006). pp. 1497–1504. ACM (2006). https://doi.org/10.1145/1143997.1144242
26. Lee, I.S., Lau, H.Y.: Adaptive state space partitioning for reinforcement learning. Engineering applications of artificial intelligence **17**(6), 577–588 (2004)

27. Madumal, P., Miller, T., Sonenberg, L., Vetere, F.: Explainable reinforcement learning through a causal lens. In: Proc. 34th Conf. on Artificial Intelligence (AAAI 2020). pp. 2493–2500. AAAI Press (2020). https://doi.org/10.1609/AAAI.V34I03.5631
28. Mavridis, C.N., Baras, J.S.: Vector quantization for adaptive state aggregation in reinforcement learning. In: 2021 American Control Conf. (ACC). pp. 2187–2192. IEEE (2021)
29. Michie, D., Chambers, R.A.: Boxes: An experiment in adaptive control. Machine intelligence **2**(2), 137–152 (1968)
30. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: Proc. 33nd Intl. Conf. on Machine Learning (ICML 2016). JMLR Workshop and Conf. Proceedings, vol. 48, pp. 1928–1937. JMLR.org (2016), http://proceedings.mlr.press/v48/mniha16.html
31. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
32. Moore, A.W.: Efficient memory-based learning for robot control. Ph.D. thesis, University of Cambridge, UK (1990). https://doi.org/10.1.1.17.2654
33. Moore, A.W.: Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In: Machine Learning Proceedings 1991, pp. 333–337. Elsevier (1991)
34. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Proc. 14th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
35. Nicol, S., Chadès, I.: Which states matter? an application of an intelligent discretization method to solve a continuous POMDP in conservation biology. PloS one **7**(2), e28993 (2012)
36. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013). https://doi.org/10.1007/S10515-013-0122-2
37. Puiutta, E., Veith, E.M.S.P.: Explainable reinforcement learning: A survey. In: Proc. 4th Intl. Cross-Domain Conf. (CD-MAKE 2020). Lecture Notes in Computer Science, vol. 12279, pp. 77–95. Springer (2020). https://doi.org/10.1007/978-3-030-57321-8_5
38. Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., Dormann, N.: Stable baselines3 (2019), https://stable-baselines3.readthedocs.io/
39. Russell, S.J., Norvig, P.: Artificial intelligence a modern approach. London (2010)
40. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
41. Seipp, J., Helmert, M.: Counterexample-guided cartesian abstraction refinement for classical planning. Journal of Artificial Intelligence Research **62**, 535–577 (2018)
42. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40–66 (2015)
43. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, 2nd edn. (2018)
44. Szita, I.: Reinforcement learning in games. In: Reinforcement Learning, Adaptation, Learning, and Optimization, vol. 12, pp. 539–577. Springer (2012). https://doi.org/10.1007/978-3-642-27645-3_17

45. Tran, H.D., Cai, F., Diego, M.L., Musau, P., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. ACM Transactions on Embedded Computing Systems (TECS) **18**(5s), 1–22 (2019)

46. Uther, W.T.B., Veloso, M.M.: Tree based discretization for continuous state space reinforcement learning. In: Proc. 15th National Conf. on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conf. (AAAI 98, IAAI 98). pp. 769–774. AAAI Press / The MIT Press (1998), http://www.aaai.org/Library/AAAI/1998/aaai98-109.php

47. Varshosaz, M., Ghaffari, M., Johnsen, E.B., Wąsowski, A.: Formal specification and testing for reinforcement learning. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023). https://doi.org/10.1145/3607835

48. Verdier, C.F., Babuška, R., Shyrokau, B., Mazo, M.: Near optimal control with reachability and safety guarantees. IFAC-PapersOnLine **52**(11), 230–235 (2019). https://doi.org/10.1016/j.ifacol.2019.09.146

49. Visser, W., Pasareanu, C.S., Pelánek, R.: Test input generation for java containers using state matching. In: Pollock, L.L., Pezzè, M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006. pp. 37–48. ACM (2006). https://doi.org/10.1145/1146238.1146243

50. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A.: Symbolic semantics for probabilistic programs. In: Proc. 20th Intl. Conf. on Quantitative Evaluation of Systems (QEST 2023). Lecture Notes in Computer Science, vol. 14287, pp. 329–345. Springer (2023). https://doi.org/10.1007/978-3-031-43835-6_23

51. Vyetrenko, S., Xu, S.: Risk-sensitive compact decision trees for autonomous execution in presence of simulated market response. arXiv preprint arXiv:1906.02312 (2019). https://doi.org/10.48550/ARXIV.1906.02312

52. Wei, H., Corder, K., Decker, K.: Q-learning acceleration via state-space partitioning. In: Proc. 17th Intl. Conf. on Machine Learning and Applications (ICMLA 2018). pp. 293–298. IEEE (2018)

53. Whiteson, S.: Adaptive Representations for Reinforcement Learning, Studies in Computational Intelligence, vol. 291. Springer (2010). https://doi.org/10.1007/978-3-642-13932-1

54. Yu, C., Liu, J., Nemati, S., Yin, G.: Reinforcement learning in healthcare: A survey. ACM Computing Surveys (CSUR) **55**(1), 1–36 (2021)

55. Zelvelder, A.E., Westberg, M., Främling, K.: Assessing explainability in reinforcement learning. In: Proc. Third Intl. Workshop on Explainable and Transparent AI and Multi-Agent Systems (EXTRAAMAS 2021). Lecture Notes in Computer Science, vol. 12688, pp. 223–240. Springer (2021). https://doi.org/10.1007/978-3-030-82017-6_14