Programmatic Reinforcement Learning: Navigating Gridworlds

Guruprerana Shabadi[1,3], **Nathanaël Fijalkow**[2,3], **Théo Matricon**[2]

[1]University of Pennsylvania, United States,
[2]CNRS, LaBRI, Université of Bordeaux, France,
[3]University of Warsaw, Poland

Abstract

The field of reinforcement learning (RL) is concerned with algorithms for learning optimal policies in unknown stochastic environments. Programmatic RL studies representations of policies as programs, meaning involving higher order constructs such as control loops. Despite attracting a lot of attention at the intersection of the machine learning and formal methods communities, very little is known on the theoretical front about programmatic RL: what are good classes of programmatic policies? How large are optimal programmatic policies? How can we learn them? The goal of this paper is to give first answers to these questions, initiating a theoretical study of programmatic RL. Considering a class of gridworld environments, we define a class of programmatic policies. Our main contributions are to place upper bounds on the size of optimal programmatic policies, and to construct an algorithm for synthesizing them. These theoretical findings are complemented by a prototype implementation of the algorithm.

1 Introduction

Reinforcement Learning (RL) is a very popular and successful field of machine learning where the agent learns a policy in an unknown environment through numerical rewards, modelled as a Markov decision process (MDP). In the tabular setting, the environment is given explicitly, which implies that typically policies are also represented explicitly, meaning as functions mapping each state to an action (or distribution of actions). Such a representation becomes quickly intractable when the environment is large and makes it hard to compose policies or reason about them. In the general setting, the typical assumption is that the environment can be simulated as a black-box. Deep reinforcement learning algorithms which learn policies in the form of large neural networks have been scaled to achieve expert-level performance in complex board and video games (Silver et al. 2018; Vinyals et al. 2019). However, they suffer from the same drawbacks as neural networks which means that the learned policies are vulnerable to adversarial attacks (Qu et al. 2021) and do not generalize to novel scenarios (Sünderhauf et al. 2018). Moreover, big neural networks are very hard to interpret and their verification is computationally infeasible.

To alleviate these pitfalls, a growing body of work has emerged which aims to learn policies in the form of programs (Verma et al. 2018; Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2019; Zhu et al. 2019; Inala et al. 2020; Landajuela et al. 2021; Trivedi et al. 2021; Qiu and Zhu 2022; Andriushchenko et al. 2022; Liang et al. 2023), under the name "*programmatic reinforcement learning*". Programmatic policies can provide concise representations of policies which are easier to read, interpret, and verify. Furthermore, their short size compared to neural networks would mean that they can also generalize well to out-of-training situations while also smoothing out erratic behaviors. The goal of the line of work we initiate here is to lay the theoretical foundations for programmatic reinforcement learning.

A programming language defined for a specific set of tasks is commonly called a Domain Specific Language (DSL). All the works cited above use very simple DSLs, combining finite state machines, decision trees, and Partial Integral Derivate (PID) controllers (originating from control theory). We believe – and show evidence in this work – that more expressive DSLs can help describing policies succinctly and naturally. The fundamental question we ask is:

> Given a class $\mathscr{P}_{\text{Env}}$ of environments, how to define a DSL $\mathscr{P}_{\text{Pol}}$ such that for each environment in $\mathscr{P}_{\text{Env}}$, there exists an optimal[1] programmatic policy $\sigma \in \mathscr{P}_{\text{Pol}}$.

Designing a DSL means choosing appropriate programming paradigm, control operators, and primitives. The design of $\mathscr{P}_{\text{Pol}}$ is a compromise: the DSL should be rich enough to express optimal policies, but simple enough to meet the objectives stated above: readable, interpretable, verifiable, generalizable, as well as learnable. We call such statements "*expressivity results*". We find many instances of expressivity results originating from different fields:

- In the study of games, positionality results for MDPs (Novotny 2023; Fijalkow et al. 2023) state the existence of optimal pure memoryless policies.
- In program verification, more specifically in the analysis of pushdown systems (Bouajjani, Esparza, and Maler

[1]Different notions of optimality can be considered here.

1997; Carayol and Serre 2023), there are several results proving the existence of optimal pushdown policies.

- In automatic control, PID controllers (Åström and Hägglund 1995) form a very classical and versatile class of programmatic controllers widely used in practice for continuous systems.
- In machine learning, the fact that neural networks are universal approximators implies that neural networks can be used as (approximate) programmatic policies for general reinforcement learning tasks.

Once existence is understood, one can wonder about sizes: how large are optimal programmatic policies? We refer to results placing upper and lower bounds on sizes of optimal programmatic policies as "*succinctness results*".

Contributions. In this work we focus on (two dimensional) gridworlds, which is a classical example in RL and inspired from theoretical robotics. We construct a very simple and elegant class of programmatic policies in the form of sequences of subgoals, inspired by Shannon's early experiments on mechanical mice. Our main contribution is a theoretical result: we prove the existence of optimal programmatic policies (an *expressivity* result), and place upper bounds on their sizes (a *succinctness* result). To the best of our knowledge, this is the first example of a non-trivial DSL, employing at its heart control loops, which can express optimal policies in a succinct and natural way on a large class of environments.

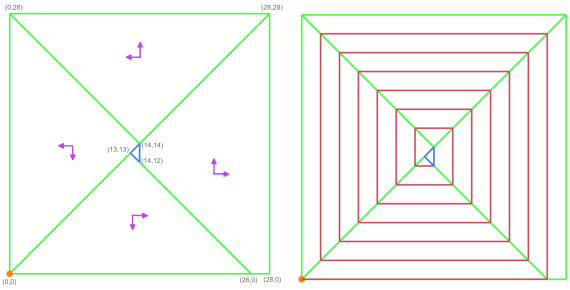
Together with this article we release a small Python package including modules for generating instances of the environments we study as well as implementation of all the algorithms defined here. See the appendix for more information.

Outline. We define gridworlds in Section 2. We introduce our DSL, called the "subgoal DSL", in Section 3, which defines the class of programmatic policies. Our algorithm for constructing programmatic policies follows two steps: first in Section 4 we define an algorithm for constructing the tree of shortest paths, and second in Section 5 we define a second algorithm for extracting from the tree a programmatic policy. The most technical proofs can be found in the appendix, together with implementation details and experiments.

2 Gridworlds

Gridworlds form a very classical example of environments in reinforcement learning and beyond. The particular class of gridworlds we consider here is inspired by theoretical robotics, and closely resembles for instance (Degener et al. 2011). The same model (with minor variations) was studied to model hybrid systems (Asarin, Maler, and Pnueli 1995).

We consider gridworlds in two dimensions: the state space is $[0, \ell]^2 \subset \mathbb{R}^2$. It is divided into closed convex polygonal regions, which we refer to later as the *regions*. In each region we specify a convex cone of available actions. A move inside a region consists in picking an available action and moving along it within the region. We also fix an initial state and a target region: the goal is from the initial state to reach any state in the target region.



(a) Gridworld (b) The optimal policy

Figure 1: Spiral

Example 1 (Spiral)**.** *In Figure 1a we have an example of a gridworld. There are* 5 *regions with the convex cone of available actions indicated by arrows within the region. The initial state (in orange) is* $(0, 0)$ *and the target region is the triangular region in the middle. Figure 1b visualizes the path from the initial state to the target region, spiralling around it.*

Formally[2], states are pairs $(x, y) \in [0, \ell]^2$. For any $v_1, v_2 \in [0, \ell]^2$, we define the segment $[v_1, v_2]$ to be the set of points connecting v_1 and v_2. We let $\texttt{Regions}$ denote the set of regions, which are closed convex polygons: we assume that $\bigcup_{R \in \texttt{Regions}} R = [0, \ell]^2$ and any two different regions only intersect on edges.

We let $\texttt{Actions}(R)$ denote the set of actions available in region R, it is a convex cone in $\mathbb{R}^2$ generated by a subset of the four cardinal directions $\texttt{LEFT}, \texttt{RIGHT}, \texttt{UP}$, and $\texttt{DOWN}$. We say that there exists a move between v_1 and v_2 if they belong to the same region and $v_2 - v_1 \in \texttt{Actions}(R)$. By extension, there exists a path from v_1 to v_k if there exists a sequence of consecutive moves starting in v_1 and ending in v_k. The length of a path is its number of moves[3].

For a region $R \in \texttt{Regions}$, we use $\texttt{Edges}(R)$ to refer to the set of edges of R. The convexity assumptions imply that we can restrict our attention to moves between edges:

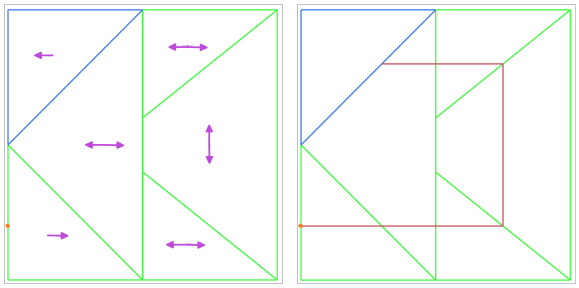
Lemma 1. *If* v_1 *and* v_2 *belong to the same region and there exists a path from* v_1 *to* v_2 *inside this region, then there exists a move between* v_1 *and* v_2*.*

Note that edges are segments, but when introducing an edge we implicitly mean the edge of some region. Since each edge is shared by at most two regions, we can define $\texttt{Adj}(R, [v, v']) \in \texttt{Regions}$ to be the region adjacent to R which both share the edge $[v, v']$ on their boundaries. However, since some edges lie on the boundaries of the grid $[0, \ell]^2$, $\texttt{Adj}(R, [v, v'])$ might not exist.

Note that gridworlds are deterministic environments, which implies that our goal is to find a minimal *path* from the initial state to the target region, where minimal means of minimal length. Thus, we will be using path and policy interchangeably moving forward.

[2]We remark that since we have a continuous state space, the value of ℓ can always be scaled and it does not play a role in the presented results.

[3]Note that this notion of length is different from the total length of the segments describing the path.



(a) Gridworld (b) Policy path

Figure 2: Double pass triangle

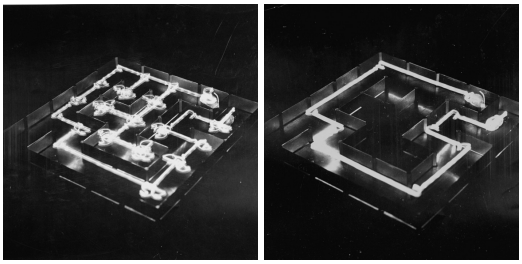


Figure 3: Images published in Life magazine in 1952 which show the path taken by Claude Shannon's mechanical mouse first while learning to navigate the maze and the direct path taken in its second attempt.

The first direction we explore while searching for concise representations of policies is region based policies where the policy picks a single action per region. For example, with the spiral gridworld from Example 1, it is sufficient to pick one action per region to navigate the agent from the initial state to the target region. However, this is not the case in general, as shown in the following example.

Example 2 (Double pass triangle)**.** *The gridworld is represented in Figure 2a. Figure 2b visualizes the (shortest) path from the initial state to the target region. The important remark here is that the triangular region in the middle is crossed twice, with different actions: first time right, second time left.*

3 The subgoal DSL

3.1 Inspiration

In 1950, Claude Shannon built, as a small project at home, one of the first instances of machine learning that the world had witnessed: a mechanical mouse capable of learning to solve a configurable maze in which the maze walls could be positioned as desired[4]. To do this, he repurposed telephone relay circuits and placed them underneath the maze board to navigate the mouse towards the exit. In a first pass, the mouse would systematically explore the whole maze looking for the exit and *learn* the path, so that in its subsequent attempts, it could swiftly reach the target. The magic was hidden in the relay circuits which would remember the path and were able to tell the mouse to turn left or right based on whether a switch was on or off. The first attempt is reminiscent of reinforcement learning with a trial and error approach to learn a policy to solve the game. However, our focus in this work will be on the subsequent attempts where we observe a programmatic abstraction to obtain a concise representation of the policy: instead of specifying the direction to follow at each point of the maze, the relay switches only indicated the points at which to change direction.

3.2 Programmatic subgoal policies

Before defining the DSL, let us dive into an example.

[4]https://www.technologyreview.com/2018/12/19/138508/mighty-mouse/

Example 3. *The following code block contains a policy program for the spiral example given in Example 1. The segments are identified by their coordinates.*

```
Do:
    From [(0, 0), (13, 13)] ->
        Target [(13, 13), (14, 12)],
    Preference: (13,13)
        Else Target [(14, 12), (26, 0)],
    Preference: (14,12)
    From [(14, 12), (26, 0)] ->
        Target [(14, 14), (28, 28)],
    Preference: (14, 14)
    From [(14, 14), (28, 28)] ->
        Target [(14, 14), (0, 28)],
    Preference: (0,28)
    From [(14, 14), (0, 28)] ->
        Target [(0, 0), (13, 13)],
    Preference: (0,0)
Until([(13, 13), (14, 12)])
```

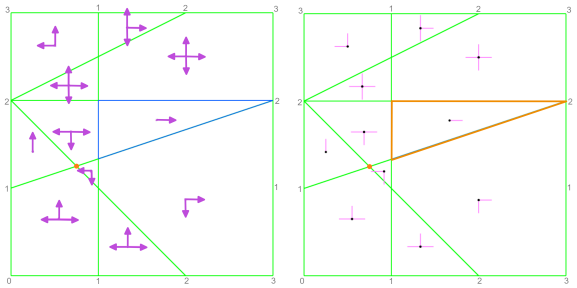
This program ensures to reach the segment $[(13, 13), (14, 12)]$*, which is an edge of the target region. To do this, it specifies moves starting from four segments:* $[(0, 0), (13, 13)]$*,* $[(14, 12), (26, 0)]$*,* $[(14, 14), (28, 28)]$*, and* $[(14, 14), (0, 28)]$*. Figure 1a can help visualize these segments: this is the four diagonals. From* $[(0, 0), (13, 13)]$ *we have two target segments: if we can reach* $[(13, 13), (14, 12)]$*, then we do (indeed this is part of the target region so we are done), otherwise we aim at* $[(14, 12), (26, 0)]$*. Importantly, when aiming at* $[(14, 12), (26, 0)]$ *we want to go as close as possible to* $(14, 12)$ *as specified by the preference. Here this means choosing only action* `RIGHT`*, and not* `DOWN`*. In the other three segments there is a single target segment.*

In general, a policy program is a sequence of `Do Until` loops:

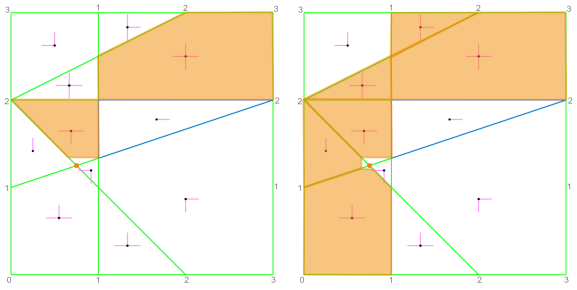
```
Do:
    P
Until(e)
```

This is interpreted as: run the local program P until reaching the edge e, which we call the local goal. A local program is a set of instructions of the form

```
From s ->
  Target s1, Preference: v1
```



(a) A gridworld (b) Depth 1



(c) Depth 2 (d) Depth 4

Figure 4: Backward construction of the tree of the winning region

```
  Else Target s2, Preference: v2
  ...
  Else Target sk, Preference: vk
```

where $s, s_1, \ldots, s_k$ are segments and $v_1, \ldots, v_k$ states, with v_i an extremal point of s_i. It is interpreted as: from any state v in the segment s, let i be the *least* index such that there is a move from v to s_i; move to the reachable state of s_i closest to v_i.

4 The tree of shortest paths

4.1 The backward algorithm

We introduce an algorithm computing the set of winning states, meaning for which there exists a path to the target region. At a high-level, the algorithm is a generic backward breadth-first search algorithm: starting from the target region, it builds and expands a tree where the nodes represent states that can reach the target.

The backward algorithm builds a tree as follows. The root is a special node, whose children are all the edges of the target region. Nodes are pairs consisting of a segment $[v_1, v_2]$ and a region R such that $[v_1, v_2] \subseteq R$. To expand a node $([v_1, v_2], R)$, we identify the region $R' = \texttt{Adj}(R, [v_1, v_2])$ sharing $[v_1, v_2]$ with R, if it exists. We then consider the set of states of R' for which there exists a move to a state in $[v_1, v_2]$: it is the convex combination of segments included in the edges of R'. For each such segment $[v_1', v_2']$, we remove from it all segments already appearing in a node of the tree, and if the segment $[v_1'', v_2'']$ it yields is non-empty, then we add a node $([v_1'', v_2''], R')$ as a child of the node $([v_1, v_2], R)$.

Example 4. *In Figure 4, we can visualize how the backward algorithm works. The tree itself can be seen in the appendix (Figure 7).*

By construction, from each state in a segment of a node of the tree we can construct a path to the target region. This is a shortest path, where shortest means that it is minimal in number of moves. Conversely, every state on an edge for which there exists a path to the target region belongs to some node of the tree. Since there might be more than one state in its parent segment that is reachable, the tree represents a family of paths to the target region. Importantly, these paths never visit the same point twice: they are *non self-intersecting*. This is due to how we have constructed the tree, filtering out parts of segments which we have already visited.

The algorithm does not terminate in general: in this example, the tree is infinite, it contains states closer and closer to the initial state but never that state. One of the main results of (Asarin, Maler, and Pnueli 1995) is a termination argument, giving bounds on how large the tree can be to include a fixed initial state. This implies the decidability of our problem, but does not address the question of representation of paths. Interestingly, this decidability result is complemented by an undecidability result for the same problem in dimension 3.

4.2 Properties of the tree

We now prove properties of paths in the tree. These are preliminary steps before constructing an algorithm deriving a programmatic policy from the tree. A branch of the tree is a sequence of segments

$$([a_1, b_1], [a_2, b_2], \ldots, [a_p, b_p]).$$

It induces a sequence of pairs of regions and edges $((R_1, e_1), \ldots, (R_{p-1}, e_{p-1}))$ satisfying the following properties, for each $i \in [1, p-1]$: (i) there exists a move between $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ contained in R_i, and (ii) $R_i \neq R_{i+1}$, and (iii) $[a_i, b_i] \subseteq e_i$, and (iv) $e_{i+1} \in \texttt{Edges}(R_i) \cap \texttt{Edges}(R_{i+1})$.

Lemma 2. *There exist at most two indices $i < j \in [1, p-1]$ such that $R_i = R_j$ and $e_i = e_{i+1} = e_j = e_{j+1}$.*

Proof. Arguing by contradiction, suppose there exist 3 indices $k < i < j$ such that $R_k = R_i = R_j$ and $e_k = e_{k+1} = e_i = e_{i+1} = e_j = e_{j+1}$. Let us denote the common edge by the segment $[a, b]$ and without loss of generality, assume for each of the segments $[a_i, b_i]$, $[a_{i+1}, b_{i+1}]$, $[a_j, b_j]$, $[a_{j+1}, b_{j+1}]$, $[a_k, b_k]$, and $[a_{k+1}, b_{k+1}]$, the first vertex of the segment is closer to a and the second vertex is closer to b. This orientation makes the following arguments easier.

In the rest of the proof, we base our arguments on the algorithm used to construct the tree of the winning region. First, let us place ourselves in the situation when the leaf associated to the segment $[a_{j+1}, b_{j+1}]$ was being extended. $[a_j, b_j]$ is a segment from which there exists a path to $[a_{j+1}, b_{j+1}]$. As we filter out parts of edges which have already been explored, either $[a_j, b_j] \subseteq [a, a_{j+1}]$ or $[a_j, b_j] \subseteq [b_{j+1}, b]$. We consider the first case $[a_j, b_j] \subseteq$

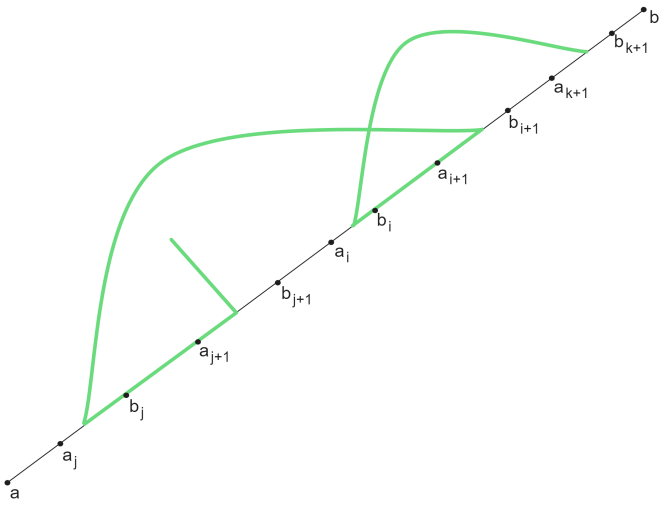


Figure 5: Any path which visits $[a_{k+1}, b_{k+1}] \to [a_i, b_i] \to [a_{i+1}, b_{i+1}] \to [a_j, b_j] \to [a_{j+1}, b_{j+1}]$ is necessarily self-intersecting.

$[a, a_{j+1}]$. Then, in fact we have a path from each point in $[a, a_{j+1}]$ to $[a_{j+1}, b_{j+1}]$. This is due to the fact that there exists $x \in [a_{j+1}, b_{j+1}]$ such that $x - a_j$ is included in the cone of available actions in R_j. As for each $y \in [a, a_{j+1}]$, $x - y$ is along the same direction as $x - a_j$, it is also in the cone of available actions. Thus, the whole segment $[a, b_{j+1}]$ has been explored until now.

The next time we visit this edge, we have that $[a_{i+1}, b_{i+1}] \subseteq [b_{j+1}, b]$. Similar to before, since $[a, b_{j+1}]$ has been explored we have either $[a_i, b_i] \subseteq [b_{j+1}, a_{i+1}]$ or $[a_i, b_i] \subseteq [b_{i+1}, b]$. We consider the former case. So $[a, b_{i+1}]$ has been explored and thus $[a_{k+1}, b_{k+1}] \subseteq [b_{i+1}, b]$. Now, one can see in Figure 5 that any path represented by the segments which visits $[a_{k+1}, b_{k+1}] \to [a_i, b_i] \to [a_{i+1}, b_{i+1}] \to [a_j, b_j] \to [a_{j+1}, b_{j+1}]$ is necessarily self-intersecting. We have completed the case $[a_j, b_j] \subseteq [a, a_{j+1}]$ and $[a_i, b_i] \subseteq [b_{j+1}, a_{i+1}]$. Each of the three other cases can be verified similarly, they all give us self-intersecting paths. □

Lemma 3. *Assume there exist indices $i < j \in [1, p-1]$ such that $(R_i, e_i) = (R_j, e_j)$, $e_i \neq e_{i+1}$ and $\texttt{Actions}(R_i)$ contains at least two orthogonal directions. Then,*

1. *if the sequence of segments forms an **inner loop** at (R_j, e_j), then all the edges of regions **inside the loop** are not visited by the subsequence $(e_1, \ldots, e_j)$.*
2. *if the sequence of segments forms an **outer loop** at (R_j, e_j), then all the edges of regions **outside the loop** are not visited by the subsequence $(e_1, \ldots, e_j)$.*
3. *if j is the least index such that $i < j$ and $(R_j, e_j) = (R_i, e_i)$, we can construct $[a'_{j+1}, b'_{j+1}]$ from $[a_j, b_j]$, $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ such that $[a'_{j+1}, b'_{j+1}] \subseteq [a_{j+1}, b_{j+1}]$ and each point in $[a'_{j+1}, b'_{j+1}]$ is reachable from a point in $[a_j, b_j]$.*

Proof. Let us begin by understanding what we mean by a loop and edges being inside and outside loops. Looking at Figure 6a, the path (in green) starting at (R_i, e_i) forms a loop at this edge. As soon as this loop is completed, the edges of regions not involved in the loop are partitioned into disjoint two sets: the edges inside the loop (in blue) and those outside (in orange). It is important that $e_{i+1} \neq e_i$ else the loop of edges would not be formed.

We will now prove item 1, the proof of item 2 follows a similar pattern. To this end, we again proceed by contradiction and assume that we have a sequence of segments which visits an inner edge before forming a loop at $(R_j, e_j) = (R_i, e_i)$. Let $p_i \in [a_i, b_i], p_j \in [a_j, b_j]$. There exists a path going from p_i to p_j. Now take a look at Figure 6b in which we can see a loop being formed by the path from p_i to p_j. If this sequence of segments visits an inner edge before forming the loop, it has to pass through the space in the edge between p_i and p_j because if not, we would have a self-intersecting path. Thus, there exist indices $k < l < i$ such that $e_k = e_l = e_i$ which are the indices where the sequence enters and exits the edge e_i while visiting an inner edge. Let $p_k \in [a_k, b_k], p_l \in [a_l, b_l]$ and consider a path visiting $p_k \to p_l \to p_i \to p_j$ which can also be seen in Figure 6b.

Since $\texttt{Actions}(R_i) \neq \emptyset$, without loss of generality, we assume that $\texttt{LEFT} \in \texttt{Actions}(R_i)$ as seen in the figure. Using the assumption that we have at least two orthogonal directions available in R_i, we have two cases: $\texttt{UP} \in \texttt{Actions}(R_i)$ or $\texttt{DOWN} \in \texttt{Actions}(R_i)$. In the first case, p_{i+1} would be reachable from p_k and $[a_k, b_k]$ (see Figure 6c) and so this segment would have been explored while the node $[a_{i+1}, b_{i+1}]$ was being extended which means that $[a_k, b_k]$ cannot exist in the space between p_i and p_j. Similarly in the second case, if $\texttt{DOWN} \in \texttt{Actions}(R_i)$, p_{j+1} would be reachable from p_i and $[a_i, b_i]$ (see Figure 6d) and therefore for the same reasons, $[a_i, b_i]$ cannot exist above p_j. This concludes the proof of item 1. □

5 Constructing programmatic policies

What remains to be done is derive from the tree a programmatic policy; one could say "compress" paths, or find regularity. This is the purpose of Algorithm 1. Lemma 3 provides the main idea for the construction of the policy synthesis procedure. As we have shown, whenever a loop is formed by the path, a new edge is discovered as soon as the loop is exited due to the non self-intersecting property. This motivates us to consider programmatic policies with a sequence of `Do Until` blocks corresponding to the sequence of edges in the order in which they are discovered by a path.

The algorithm works as follows: it takes in a sequence of segments from the tree $([a_1, b_1], \ldots, [a_p, b_p])$ and the corresponding sequence of edges $(e_1, \ldots, e_p)$. It goes through both these sequences and each time a new edge is encountered, it begins a new `Do Until` block. At each iteration of the loop, if it sees that a segment of the next edge is already a target, i.e., if it visits the same pair of consecutive edges twice, it merges the two segments. This merging procedure ensures that when we encounter a loop in our path, segments belonging to the same edge are merged thus resulting in a compact representation of the sequence of segments.

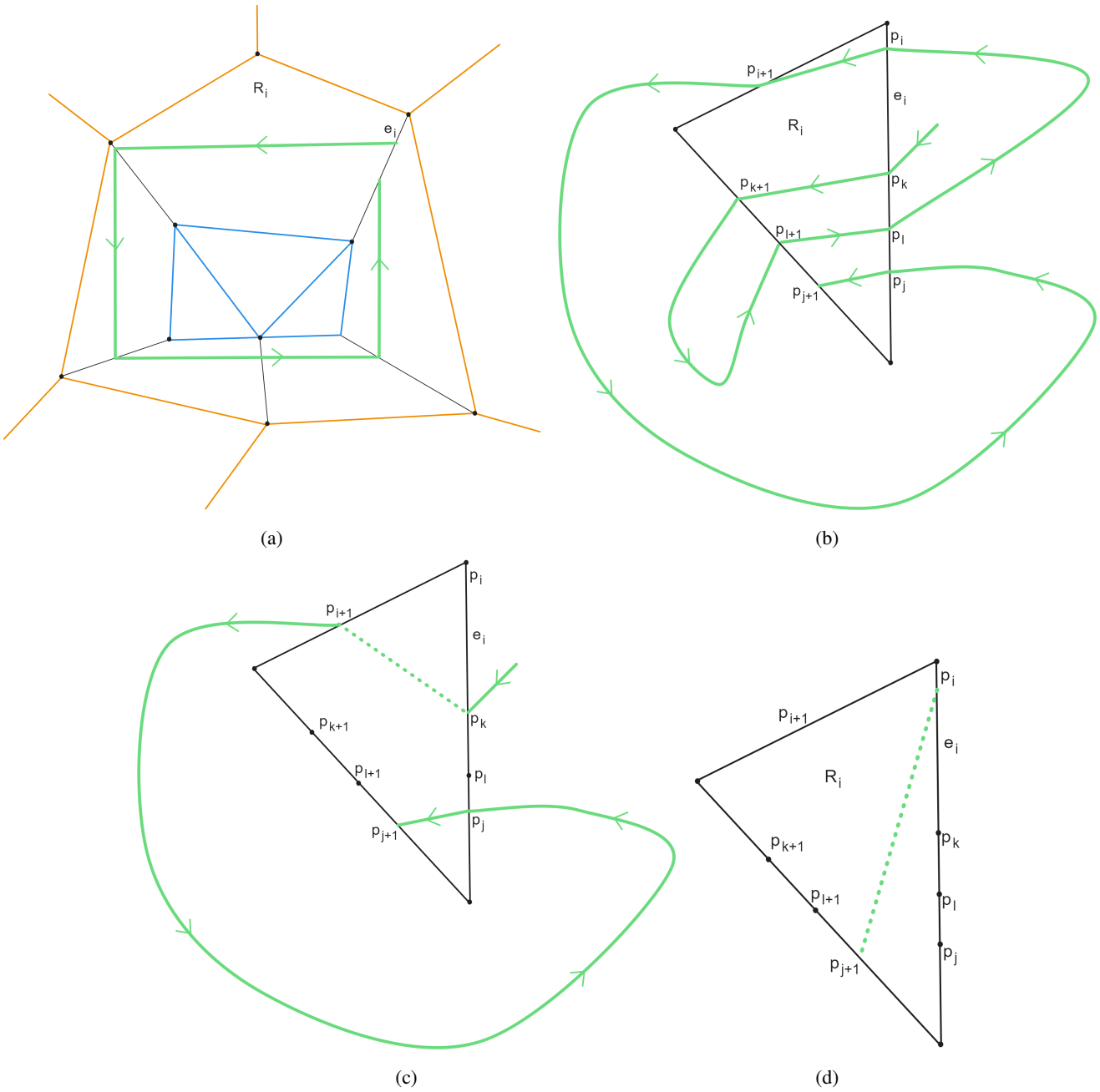


Figure 6: (a) Inner loop at (R_i, e_i) partitions all the edges not involved in the loop into inner (blue) and outer (orange) edges (b) Loop at (R_i, e_i) with the path previously visiting an inner edge at index $k+1$ (c) Counterexample path in the case where DOWN and LEFT are available in R_i (d) Counterexample path in the case where UP and LEFT are available in R_i.

When we merge, we also set the preference depending on which side the next segment is with respect to the previous segment on the same edge. Intuitively, this allows to distinguish between *inner* and *outer* loops where the preference would force the policy to navigate towards a certain extreme of a segment thereby allowing the agent to progress closer towards the target region. The regions in which the allowed actions are a subset of $\{\texttt{LEFT}, \texttt{RIGHT}\}$ or $\{\texttt{UP}, \texttt{DOWN}\}$ are handled differently by the algorithm. Since diagonal directions are not allowed in such regions, it suffices to specify the direction in which to navigate.

We now shift our attention to proving the correctness of Algorithm 1, which means proving an *expressivity result*, given by Theorem 1 and a *succinctness result* in the form of an upper bound on the size of the synthesized programmatic policies, in Theorem 2.

Algorithm 1: Synthesizing a programmatic policy

Input: A branch in the tree $([a_1, b_1], \ldots, [a_p, b_p])$ and the corresponding sequence of edges $(e_1, \ldots, e_p)$ and regions $(R_1, \ldots, R_{p-1})$
VisitedEdges = $\emptyset$
for $i = 1$ to $p - 1$ **do**
  if $e_i \notin$ VisitedEdges **then**
    add e_i to VisitedEdges
    start new `Do Until` block with local goal e_i
  if there is no `From` e_i in current `Do Until` block **then**
    add a `From` e_i instruction to the current block
  if ($\texttt{Actions}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$ or $\texttt{Actions}(R_i) \subseteq \{\texttt{UP}, \texttt{DOWN}\}$) and $e_{i+1} \neq e_i$ **then**
    let `dir` $\in \{\texttt{LEFT}, \texttt{RIGHT}, \texttt{UP}, \texttt{DOWN}\}$ such that `dir` is the direction of e_{i+1} with respect to e_i
    add the instruction `GO dir` inside the `From` e_i instruction if it is not already present
  else if some segment $[a, b]$ included in e_{i+1} is currently the preferred target segment of e_i and $e_{i+1} \neq e_i$ **then**
    merge $[a, b]$ and $[a_{i+1}, b_{i+1}]$ into one segment
    if merged segment is $[a, b_{i+1}]$ **then**
      set preference to a
    if merged segment is $[a_{i+1}, b]$ **then**
      set preference to b
  else
    add to the top of the instruction `From` e_i a new target $[a_{i+1}, b_{i+1}]$ with preference a_{i+1}
end

Theorem 1. *Given the shortest sequence of segments in the tree $([a_1, b_1], \ldots, [a_p, b_p])$ going from the initial state to the target region, Algorithm 1 synthesizes an optimal programmatic policy that can navigate an agent through these segments.*

Theorem 2. *Given the shortest sequence of segments in the tree $([a_1, b_1], \ldots, [a_p, b_p])$ going from the initial state to the target region, Algorithm 1 synthesizes an optimal programmatic policy of size at most $O(|\texttt{Regions}|^4)$.*

In Theorem 2, we made the assumption that each segment in the sequence can be stored in constant space. This would imply that we can store rationals of arbitrary precision representing the endpoints of the segments in constant space. Obviously, this is not a valid assumption in practice and in the case where all the edges of the regions are described by rationals, we prove the following upper bound on the space required to store the segments.

Lemma 4. *Suppose there exists $D \in \mathbb{N}$ such that each of the endpoints of each of the edges of the regions are of the form $\left(\frac{a}{D}, \frac{b}{D}\right)$ for some $a, b \in [0, D]$, then each segment of a path of the tree $([a_1, b_1], \ldots, [a_p, b_p])$ can be stored in space at most $O(pD \log(D))$ when both a_p and b_p can be written in the same form.*

6 Conclusions and future work

This work is a first step towards theoretical foundations of programmatic reinforcement learning, and more specifically the question of designing domain-specific languages for policies. The take away message is that for a large class of environments we were able to construct programmatic policies using in a non-trivial way control loops. We proved expressivity results, meaning existence of optimal programmatic policies, as well as succinctness results, proving that there exist optimal programmatic policies of size polynomial in the number of regions. We hope that this paper will open a fruitful line of research on the theoretical front. We outline here promising directions.

A burning question is studying the trade-offs between sizes of programmatic policies and their performances. In this paper, we focused on optimal policies, meaning shortest paths to the target region. Are there smaller programmatic policies ensuring near optimal number of moves?

The motivations of this work is to construct programmatic policies because they are readable, interpretable, and verifiable. Hence alongside with expressivity and succinctness results, we should also investigate how we can reason with programmatic policies, and in particular verify them. Developing verification algorithms for programmatic policies is a natural next step for this work. Another desirable property is generalizability: programmatic policies are expected to generalize better, as it was argued in the original papers (Bastani, Pu, and Solar-Lezama 2018; Inala et al. 2020; Trivedi et al. 2021). Further theoretical and empirical studies will help us understand this argument better.

Last but not least, once we understand which classes of programmatic policies are expressive and succinct, remains the main question: how do we learn programmatic policies? Many approaches have been developed for decision trees, PIDs, and related classes. Learning more structured programmatic policies involving control loops is a very exciting challenge for the future, which has been tackled very recently (Moraes et al. 2023; Aleixo and Lelis 2023; Batz et al. 2024)!

Acknowledgements

This work was supported by the SAIF project, funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-23-PEIA-0006.

References

Aleixo, D. S.; and Lelis, L. H. S. 2023. Show Me the Way! Bilevel Search for Synthesizing Programmatic Strategies. In *Conference on Artificial Intelligence, AAAI*, 4991–4998. AAAI Press.

Andriushchenko, R.; Ceska, M.; Junges, S.; and Katoen, J. 2022. Inductive synthesis of finite-state controllers for POMDPs. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence, UAI*, volume 180 of *Proceedings of Machine Learning Research*, 85–95. PMLR.

Asarin, E.; Maler, O.; and Pnueli, A. 1995. Reachability Analysis of Dynamical Systems Having Piecewise-Constant Derivatives. *Theoretical Computer Science*, 138(1): 35–65.

Åström, K.; and Hägglund, T. 1995. *PID Controllers: Theory, Design, and Tuning*. ISA - The Instrumentation, Systems and Automation Society. ISBN 1-55617-516-7.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Annual Conference on Neural Information Processing Systems, NeurIPS*, 2499–2509.

Batz, K.; Biskup, T. J.; Katoen, J.-P.; and Winkler, T. 2024. Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs. *Proceedings of the ACM on Programming Languages*, 8(POPL): 2792–2820.

Bouajjani, A.; Esparza, J.; and Maler, O. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory, CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, 135–150. Springer.

Carayol, A.; and Serre, O. 2023. Pushdown games. In Fijalkow, N., ed., *Games on Graphs*. Arxiv.

Degener, B.; Kempkes, B.; Langner, T.; auf der Heide, F. M.; Pietrzyk, P.; and Wattenhofer, R. 2011. A tight runtime bound for synchronous gathering of autonomous robots with limited visibility. In *ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, 139–148. ACM.

Fijalkow, N.; Bertrand, N.; Bouyer-Decitre, P.; Brenguier, R.; Carayol, A.; Fearnley, J.; Gimbert, H.; Horn, F.; Ibsen-Jensen, R.; Markey, N.; Monmege, B.; Novotný, P.; Randour, M.; Sankur, O.; Schmitz, S.; Serre, O.; and Skomra, M. 2023. *Games on Graphs*. Online.

Inala, J. P.; Bastani, O.; Tavares, Z.; and Solar-Lezama, A. 2020. Synthesizing Programmatic Policies that Inductively Generalize. In *International Conference on Learning Representations, ICLR*. OpenReview.net.

Landajuela, M.; Petersen, B. K.; Kim, S.; Santiago, C. P.; Glatt, R.; Mundhenk, T. N.; Pettit, J. F.; and Faissol, D. M. 2021. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning, ICML*, volume 139 of *Proceedings of Machine Learning Research*, 5979–5989. PMLR.

Liang, J.; Huang, W.; Xia, F.; Xu, P.; Hausman, K.; Ichter, B.; Florence, P.; and Zeng, A. 2023. Code as Policies: Language Model Programs for Embodied Control. In *IEEE International Conference on Robotics and Automation, ICRA*, 9493–9500. IEEE.

Moraes, R. O.; Aleixo, D. S.; Ferreira, L. N.; and Lelis, L. H. S. 2023. Choosing Well Your Opponents: How to Guide the Synthesis of Programmatic Strategies. In *International Joint Conference on Artificial Intelligence, IJCAI*, 4847–4854. ijcai.org.

Novotny, P. 2023. Markov decision processes. In Fijalkow, N., ed., *Games on Graphs*. Arxiv.

Qiu, W.; and Zhu, H. 2022. Programmatic Reinforcement Learning without Oracles. In *International Conference on Learning Representations, ICLR*. OpenReview.net.

Qu, X.; Sun, Z.; Ong, Y.; Gupta, A.; and Wei, P. 2021. Minimalistic Attacks: How Little It Takes to Fool Deep Reinforcement Learning Policies. *IEEE Transactions on Cognitive and Developmental Systems*, 13(4): 806–817.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Sünderhauf, N.; Brock, O.; Scheirer, W. J.; Hadsell, R.; Fox, D.; Leitner, J.; Upcroft, B.; Abbeel, P.; Burgard, W.; Milford, M.; and Corke, P. 2018. The limits and potentials of deep learning for robotics. *International Journal on Robotics Research*, 37(4-5): 405–420.

Trivedi, D.; Zhang, J.; Sun, S.; and Lim, J. J. 2021. Learning to Synthesize Programs as Interpretable and Generalizable Policies. In *Annual Conference on Neural Information Processing Systems, NeurIPS*, 25146–25163.

Verma, A.; Le, H. M.; Yue, Y.; and Chaudhuri, S. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Annual Conference on Neural Information Processing Systems, NeurIPS*, 15726–15737.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically Interpretable Reinforcement Learning. In *International Conference on Machine Learning, ICML*, volume 80 of *Proceedings of Machine Learning Research*, 5052–5061. PMLR.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnevets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gülçehre, Ç.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T. P.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354.

Zhu, H.; Xiong, Z.; Magill, S.; and Jagannathan, S. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 686–701. ACM.

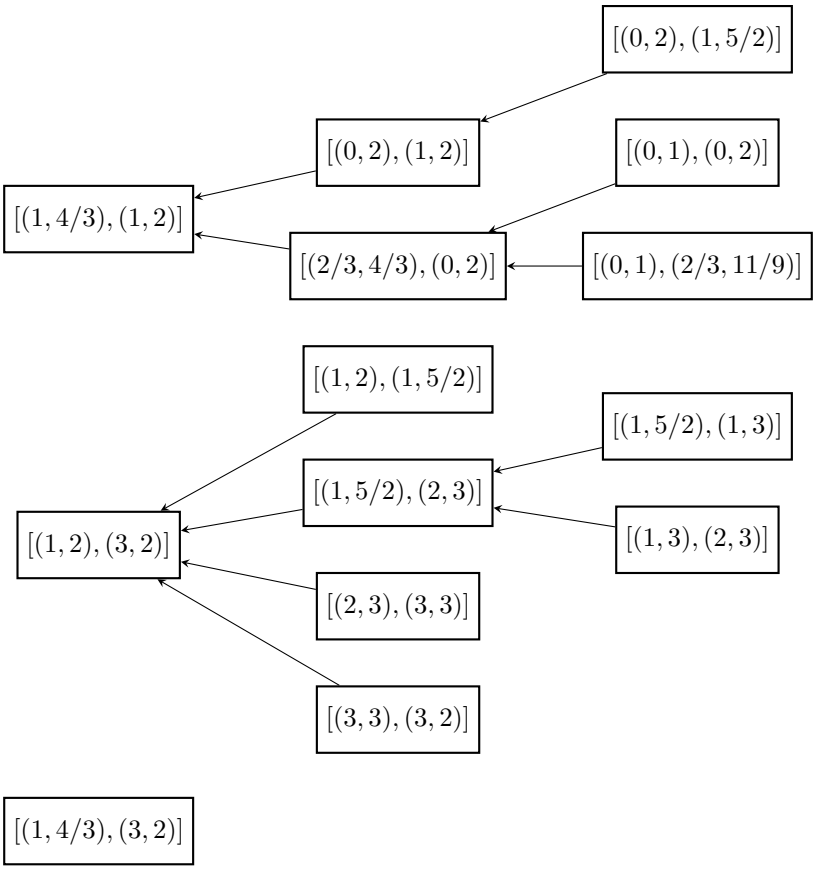


Figure 7: Tree of the winning region corresponding to Figure 4 until depth 3. The leftmost nodes are the three children of the root. In this tree we only indicate segments, not the corresponding regions.

A Illustration of the tree

See Figure 7. We represent 3 subtrees, each corresponding to an edge of the target region. Starting from these 3 edges, we add segments of edges of adjacent regions as nodes to the trees in a breadth-first manner. There exists a path from each state in a node to a state in its parent node.

B Proof of the third item of Lemma 3

We prove item 3. Let us denote $[a_z, b_z]$ by $[(x_z, y_z), (x'_z, y'_z)]$ for $z \in \{i, i+1, j, j+1\}$. Same as before, let us assume without loss of generality that $\texttt{LEFT} \in \texttt{Actions}(R_i)$ and $x_{i+1} \leq x_i, x_{j+1} \leq x_j$. This can be visualized through Figure 8. As at least two orthogonal directions are available in R_i, let us again split into two cases with the first one being $\texttt{UP} \in \texttt{Actions}(R_i)$. Firstly, $[a_j, b_j]$ is below $[a_i, b_i]$ as seen in the figure because if not, $[a_{j+1}, b_{j+1}]$ (which would also have to be above $[a_{i+1}, b_{i+1}]$ to avoid self-intersecting paths) would be reachable from $[a_i, b_i]$ so it would have already been explored at index j and cannot exist there at index i. By assumption, as j is the least such index satisfying the property, using arguments similar to the previous part of the proof, we have that there is no index $k > j$ such that $[a_k, b_k] \subseteq [a_i, b_j]$. This means that when the tree node associated to the segment $[a_{j+1}, b_{j+1}]$ was being extended, $[a_i, b_j]$ was unexplored. Also, we have that $\texttt{DOWN} \notin \texttt{Actions}(R_i)$, otherwise $[a_{j+1}, b_{j+1}]$ would be reachable from $[a_i, b_i]$. Thus necessarily, $y_{j+1} = y_j$, i.e., a_{j+1} lies on the same y-coordinate as a_j. As a result, we can determine a_{j+1} simply by intersecting the line $y = y_j$ with the region R_i. Lastly, $y'_j \leq y'_{j+1}$, i.e., b_j lies below b_{j+1} and therefore we can set $a'_{j+1} \coloneqq a_{j+1}$ (which can be computed) and b'_{j+1} to be other point on the edge of R_i on the same y-coordinate as b_j. We remark that there is a path from each point in $[a_j, b_j]$ to $[a'_{j+1}, b'_{j+1}]$: just go left! Symmetric arguments can be used to deal with the case in which $\texttt{DOWN} \in \texttt{Actions}(R_i)$.

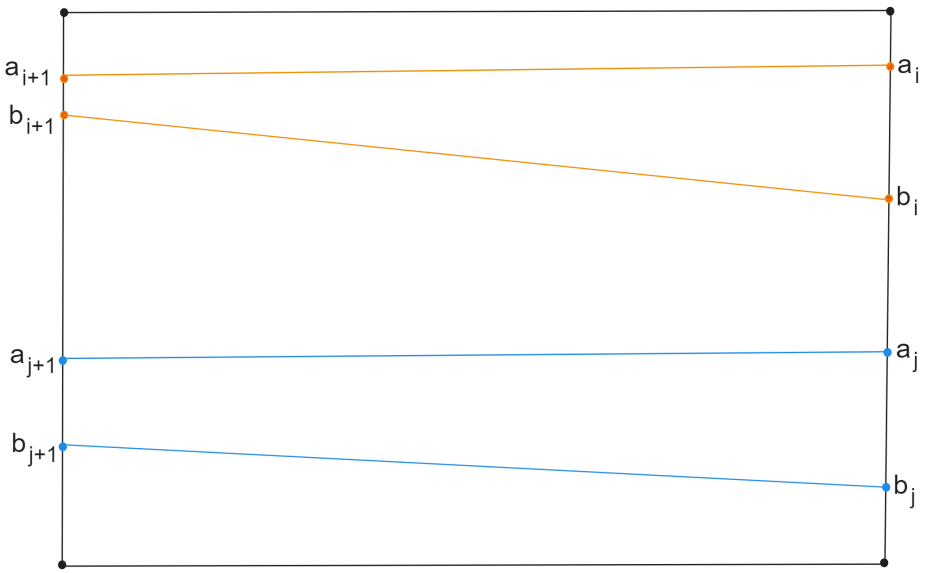


Figure 8: Constructing $[a'_{j+1}, b'_{j+1}]$ from $[a_j, b_j]$, $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ in the proof of Lemma 3.

C Correctness of the compression algorithm: Proof of Theorem 1

Proof. We will show that the synthesized policy navigates an agent through the sequence of segments in the same order. Arguing by induction, it suffices to show that if the agent is currently at a point $p_i \in [a_i, b_i]$ for some $i \in [1, p-1]$, the policy would guide the agent towards a point $p_{i+1} \in [a_{i+1}, b_{i+1}]$. When the agent is at p_i, the execution of the program would be in a certain `Do Until` block. Firstly, we argue that the edge e_i that contains $[a_i, b_i]$ contains at least one target in its `From` e_i instruction in the current block. This is true by construction because the synthesis algorithm processes each segment sequentially and as a result each edge that appears in the sequence would have an associated `From` instruction. Note that the only case when this would be untrue is when the goal of the `Do Until` block is reached in which case the program execution would switch to the next block.

Next, within a `From` instruction, there may be several target segments separated by `Else` statements. Again, by construction, at least one of them is reachable from p_i. Furthermore, the first reachable target segment contains $([a_{i+1}, b_{i+1}])$ because if not, this target segment (which would appear at index greater than $i+1$) would be reachable from $[(a_i, b_i)]$. Consequently, we would have a shorter sequence of segments leading to a contradiction.

Lastly, it remains to prove that the policy would indeed navigate to a point in $([a_{i+1}, b_{i+1}])$. Here, we need to distinguish two cases. In the first case, suppose that the target segment in the `From` e_i instruction was formed without any merging of segments. Then necessarily the target segment coincides with $([a_{i+1}, b_{i+1}])$ or it is a region in which the allowed actions are a subset of $\{\texttt{LEFT}, \texttt{RIGHT}\}$ or $\{\texttt{UP}, \texttt{DOWN}\}$. Both the scenarios are easily handled. The interesting case is when the target segment was formed by the merging of segments. It means that the edge e_i was visited twice and we have a loop at e_i. This is where the `Preference` plays a role in navigating the agent in the correct direction. Note that at least two orthogonal actions allowed in $\mathcal{R}_i$. As we noted in the proof of Lemma 3, when we enter a loop in such a region, the segment $[a_{i+1}, b_{i+1}]$ has the same x or y coordinates as $[a_i, b_i]$ depending on the actions allowed. This means that there is a point p_{i+1} with the same x or y coordinate as p_i in the target segment. By taking another look at Figure 8, we can further see that p_{i+1} coincides with the point reachable in the target segment that is extremal with respect to the `Preference`. Here, the synthesized merged segment would be $[a_{i+1}, b_{j+1}]$ with `Preference:` b_{j+1}. If the agent is at a point in $[a_j, b_j]$, and the allowed actions are `UP` and `LEFT`, the policy would navigate the agent towards a point in $[a_{j+1}, b_{j+1}]$ with the same y-coordinate (i.e., go `LEFT`) because that would be the point that is extremal with respect to the specified `Preference`. □

D Analysis of the size of programs: Proof of Theorem 2

Proof. Firstly, we remark that as we have at most $|\texttt{Regions}|^2$ edges in the gridworld, we have at most $|\texttt{Regions}|^2$ blocks `Do Until` representing the subgoals. Suppose that the sequence of segments visits q unique edges and let

$\bar{e}_1 \to \bar{e}_2 \to \cdots \to \bar{e}_q$ denote the sequence of these edges in the order that they are first visited. Each of these correspond to a `Do Until` block in the policy.

Let us now analyse the size of each `Do Until` block which corresponds to a programmatic representation of a part of the sequence of segments going from $\bar{e}_m$ to $\bar{e}_{m+1}$ for a certain $m \in [1, q-1]$. Let $([c_1, d_1], \ldots, [c_l, d_l])$ denote this sequence of segments which forms a part of the sequence $([a_1, b_1], \ldots, [a_p, b_p])$. Note that $[c_1, d_1]$ and $[c_l, d_l]$ are segments within the edges $\bar{e}_m$ and $\bar{e}_{m+1}$ respectively. Also, keep in mind the sequence of pairs of regions and edges traversed by the sequence $((R_1, e_1), \ldots, (R_l, e_l))$ which will be useful in the rest of the proof.

Observe that each unique edge in $\{e_1, \ldots, e_l\}$ is associated with a `From` instruction within the `Do Until` block. As a first step, let us treat the indices $i \in [1, l-1]$ such that $e_{i+1} = e_i$. By Lemma 2, this happens at most twice with e_i so this contributes at most two targets, and thereby two lines to the `From` instruction of e_i. On the other hand, suppose $e_{i+1} \neq e_i$. As a first subcase, suppose $\texttt{Actions}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$ or $\texttt{Actions}(R_i) \subseteq \{\texttt{UP}, \texttt{DOWN}\}$. From Algorithm 1 it is clear that such regions would add at most two targets (directions) to the `From` e_i instruction. Next assume at least two orthogonal directions are allowed in R_i. If $e_{i+1} \neq e_i$ at most once with e_i, then it contributes only one target to the `From` e_i instruction. However, if there is another index $j \in [1, l-1]$ such that $e_i = e_j$ and $e_{j+1} \neq e_j$, by Lemma 3, it means that we have an inner or an outer loop. From this case, we would again have at most two targets: either $e_{j+1} = e_{i+1}$ and so the target segments would be merged (and the loop continues) or e_{j+1} is a new edge never visited before (and the loop is exited). In other words, a loop contributes at most two target segments to each edge and there is at most one loop in a `Do Until` block.

In total, we have at most six targets associated with each edge-region pair in the `Do Until` block. As each edge can be shared between two regions, at most twelve targets are associated with each edge. Further, since only m edges are explored by the m-th block, each block has at most $12m$ instructions. Thus, we obtain the following bound on the total length of the policy

$$\sum_{m=1}^{|\texttt{Regions}|^2} 12m = O(|\texttt{Regions}|^4). \tag{1}$$

□

E Analysis of the bitsize of programs: Proof of Lemma 4

Proof. Let

$$\left[\left(\frac{x_1}{D}l, \frac{y_1}{D}l\right), \left(\frac{x_2}{D}l, \frac{y_2}{D}l\right)\right]$$

be an edge of a region in `Regions` for some $x_1, y_1, x_2, y_2 \in [\![0, D]\!]$.

Associated to this edge, we can write the two following equations for the line on which it lies on:

$$y = \frac{y_2 - y_1}{x_2 - x_1}x + \frac{(x_2 - x_1)y_1 - (y_2 - y_1)x_1}{(x_2 - x_1)D}l \tag{2}$$

$$x = \frac{x_2 - x_1}{y_2 - y_1}y + \frac{(y_2 - y_1)x_1 - (x_2 - x_1)y_1}{(y_2 - y_1)D}l \tag{3}$$

Note that when $y_2 - y_1 = 0$ or $x_2 - x_1 = 0$, one of the two equations does not exist. Observing that $(x_2 - x_1), (y_2 - y_1) \in [\![-D, D]\!]$, we have that $H \coloneqq D(D!)$ is divisible by $(x_2 - x_1)D$ and $(y_2 - y_1)D$. So we can write

$$y = \frac{s}{H}x + \frac{d}{H}l \tag{4}$$

$$x = \frac{s'}{H}y + \frac{d'}{H}l \tag{5}$$

where

$$s \coloneqq H\frac{y_2 - y_1}{x_2 - x_1} \tag{6}$$

$$d \coloneqq H\frac{(x_2 - x_1)y_1 - (y_2 - y_1)x_1}{(x_2 - x_1)D} \tag{7}$$

and similarly for s' and d'. In particular, these numerators satisfy the following bounds

$$|s| = |H\frac{y_2 - y_1}{x_2 - x_1}| \leq H|y_2 - y_1| \leq HD \tag{8}$$

$$|d| = |H\frac{(x_2 - x_1)y_1 - (y_2 - y_1)x_1}{(x_2 - x_1)D}| \leq \frac{H}{D}(|(x_2 - x_1)y_1 + (y_2 - y_1)x_1|) \leq \frac{H}{D}2D^2 = HD \tag{9}$$

With this, we are now able to write the equation for the line containing each of the edges as shown in 4. These two forms of the equation are relevant to us because each time we are extending a node of a segment $[a_i, b_i]$ in a tree of the winning region, we are computing $[a_{i-1}, b_{i-1}]$ by intersecting an edge with the half-planes reachable by the allowed actions in R_{i-1} which amounts to finding the intersection points of the line containing e_{i-1} with a certain horizontal or vertical line. So we can substitute the value of the x or y coordinate in 4 to obtain the endpoints of $[a_{i-1}, b_{i-1}]$. Notice that filtering out explored parts of edges only uses precomputed intersection points.

For example, if while extending the segment $[a_p, b_p]$, to compute $[a_{p-1}, b_{p-1}]$, we have to intersect with the line $x = (a/H)l$ where $a \in [\![0, H]\!]$. Substituting this into 4, gives us

$$y_{p-1} = \frac{s}{H}\frac{a}{H}l + \frac{d}{H}l = \frac{sa + dH}{H^2}l \tag{10}$$

with $|sa + dH| \leq H^2D$. Now suppose while extending the segment $[a_{p-1}, b_{p-1}]$ to $[a_{p-2}, b_{p-2}]$, we have to intersect with the line $y = \frac{u}{H^2}l$ where $u \in [\![0, H^2D]\!]$. Then, in the same way,

$$x_{p-2} = \frac{v}{H^3} \tag{11}$$

for some $v \in [\![0, H^3D^2]\!]$.

Continuing this argument inductively, we get that the endpoints of the first segment $[a_1, b_1]$ can be written with coordinates of the form

$$\frac{t}{H^p}$$

where $t \in [\![0, H^pD^{p-1}]\!]$. Furthermore, since

$$H^pD^{p-1} = D^{2p-1}(D!)^p = O(D^{p(D+2)-1}) \tag{12}$$

So, in order to store $[a_1, b_1]$ which potentially requires more space to store than any of the other segments, we need to store a few integers in $[\![0, H^pD^{p-1}]\!]$ which requires

$$\log(H^pD^{p-1}) = O((p(D + 2) - 1)\log(D)) = O(pD\log(D)) \tag{13}$$

bytes. □

F Implementation and evaluation

Together with this article we release a small Python package[5] including modules for generating gridworld instances as well as implementation of the algorithms for the construction of the tree of the winning region and synthesis of policies.

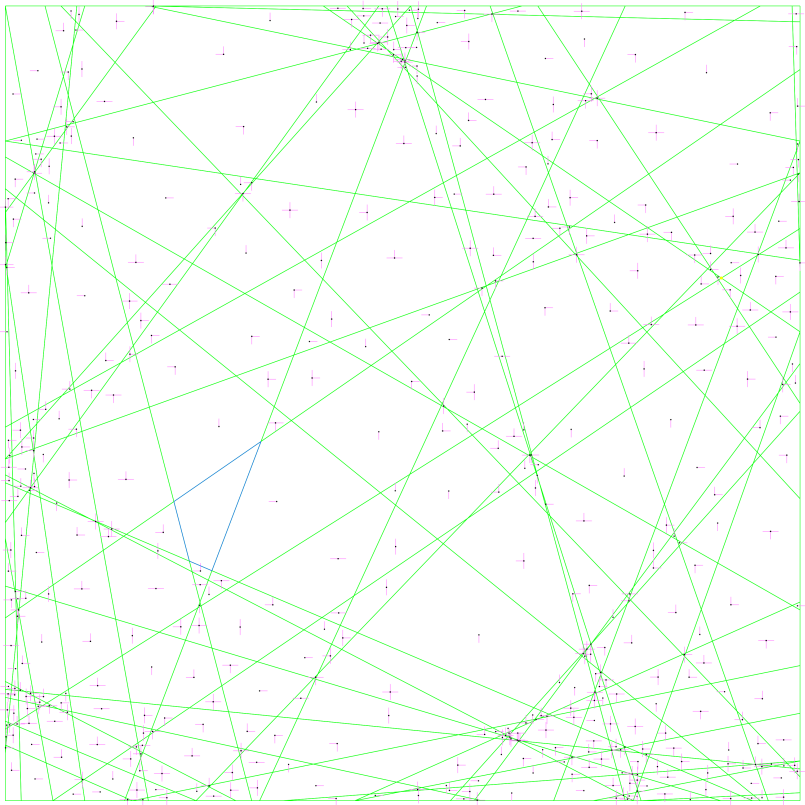
Gridworlds are continuous environments. In practice, we use a discretised version: the state space is a $n \times n$. The regions are defined by linear predicates.

The `linpreds` module contains classes to generate random gridworlds. This is done by choosing at random linear predicates on a $n \times n$ grid where the endpoints of the linear predicates are in $[0, n-1]$. We then assign random actions to each of the regions that are created by the intersections of these linear predicates. It also includes functions to generate a PRISM program from the gridworld.

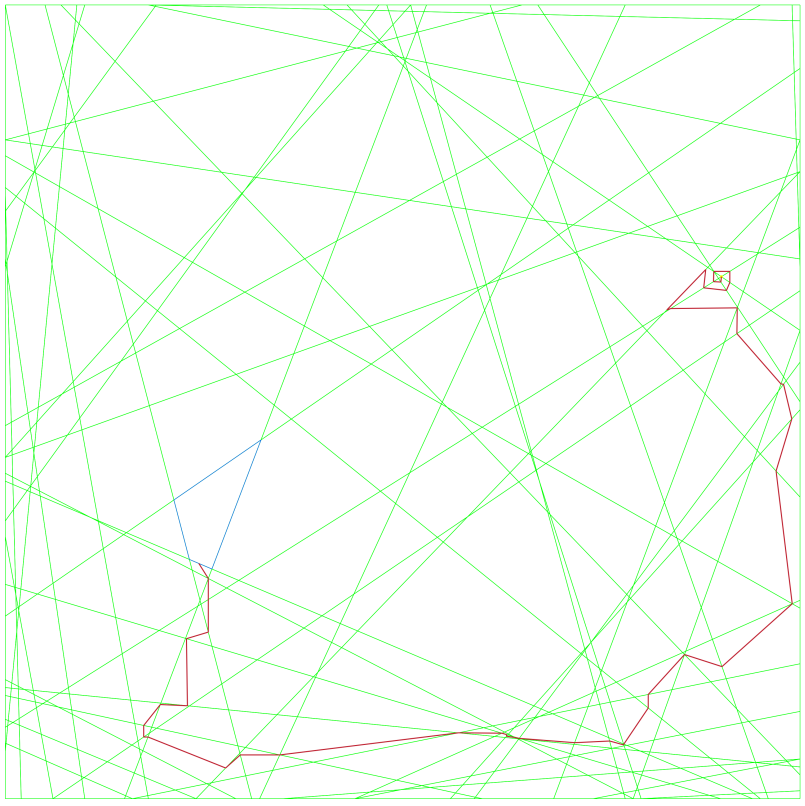
The `polygons` module contains the infrastructure to translate the linear predicates generated into a data structure which makes the backward winning region construction efficient. We use the half-edge data structure (popular in computational geometry) by looking at the gridworld as a planar tiling of the grid with polygons. The `backward_reachability` module constructs the tree of the winning region. The `game.continuous` module implements a reinforcement learning-like game environment which can simulate a policy for gridworld instances. Lastly, `policy.subgoals` implements Algorithm 1 and can synthesize programmatic policies from a path of segments.

The `benchmarks` folder contains a set of 17 benchmarks including the spiral and double-pass triangle examples. The others were generated by our code and go up to instances with 50 linear predicates and 600+ regions. The synthesized policies can be seen and the policy path visualized in images in the respective folders of the benchmarks. The benchmark data can be found in Table 1. We measure size in bytes to take into account the size of numerical coefficients involved. We observe that the size of the policy is polynomial (almost linear) in the size of the gridworld. Note that the size of the gridworld is the space required to store all the edges of all the regions of the gridworld.

[5] https://github.com/guruprerana/smol-strats



(a) A gridworld



(b) Synthesized policy path

Figure 9: The `size100preds50-4` benchmark is a gridworld with 542 regions.

Benchmark	Gridworld size	Policy size	Regions
spiral	10833	20847	14
size3preds5loopy	8786	15744	11
size50preds10-1	30669	57926	32
size50preds20-1	105252	126263	113
size100preds20-1	119253	136257	126
size100preds20-2	108256	130591	115
size100preds30-1	228676	256031	233
size100preds30-2	220557	248063	230
size100preds30-3	221308	244846	227
size100preds30-4	266882	303592	271
size100preds50-1	612940	655357	616
size100preds50-2	668670	706836	668
size100preds50-3	635978	663439	628
size100preds50-4	538503	576528	542
size100preds50-5	603314	641681	616

Table 1: Size of synthesized policies (in bytes) for a set of generated benchmarks.