Towards programmatic reinforcement learning: the case of deterministic gridworlds

GURUPRERANA SHABADI*, École Polytechnique, Institut Polytechnique de Paris, France

Category: Graduate
Research advisor: Nathanäel Fijalkow, LaBRI, Bordeaux, France

Starting from a programmatic representation of a Markov decision process (MDP) in the PRISM syntax [4], we examine the task of synthesizing a policy in the form of a program for the MDP. The PRISM syntax allows us to specify MDPs concisely by partitioning the state space into regions with similar actions and transitions. In this work, we restrict ourselves to a subclass of two dimensional deterministic gridworlds partitioned into regions along linear predicates. We present an algorithm to synthesize programmatic policies to achieve reachability objectives which exploit the symmetries present in the specification of the MDP. Our programs use memory to track subgoals and navigate between the edges of regions to provide a concise representation of a policy. Our main result is a proof of an upper bound on the size of the synthesized programs. We also provide an implementation of our synthesis algorithm which is evaluated on randomly generated instances of gridworlds.

CCS Concepts: • **Computing methodologies** → **Planning for deterministic actions**; *Discrete space search*; *Planning with abstraction and generalization*; • **Software and its engineering** → **Automatic programming**.

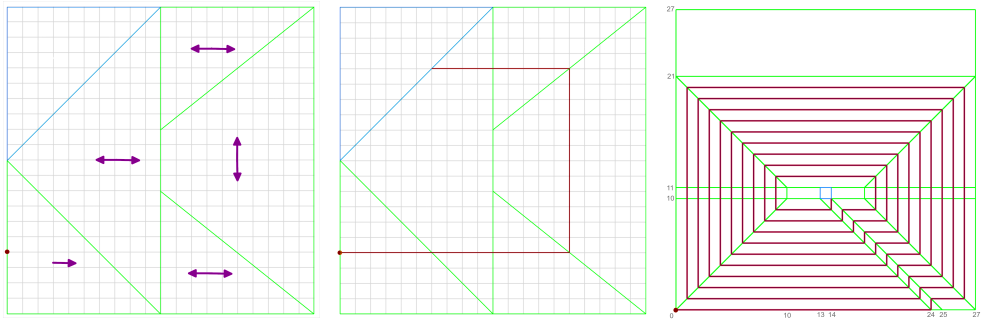
Additional Key Words and Phrases: Reinforcement learning, Program synthesis, Gridworld, PRISM

1 INTRODUCTION

Model-free reinforcement learning has established itself as the paradigm of choice to learn policies in environments that can be described by Markov decision processes (MDPs). 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 [3, 9, 12] and have found applications in areas such as robotics [6], autonomous driving [8], and epidemic control [5]. However, they suffer from the same drawbacks as neural networks which means that the learned policies are vulnerable to adversarial attacks [7] and do not generalize to novel scenarios [10, 13]. To alleviate these pitfalls, a growing body of work has emerged which aims to learn policies in the form of programs. *Programmatic* policies can provide concise representations of policies which are easier to interpret and verify. Furthermore, their short size compared to neural networks means that they can also generalize well to out-of-training situations while also smoothing out erratic behaviors. Most existing work on programmatic reinforcement learning work within a teacher-learner framework where they search for a program in a domain-specific language (DSL) that can approximate a pre-trained neural network policy [1, 2, 11]. A drawback with this approach is that while the symmetries and domain knowledge of the MDP are represented in the DSL, they remain unused during the first step of training the neural network policy.

We take a different approach in this work: *starting* from a programmatic representation of an MDP, our goal is to synthesize a policy for the MDP in the form of another program. PRISM [4], a probabilistic model checker, provides a programmatic syntax for specifying MDPs. By allowing us to partition the state space into regions which have similar transitions and allowing updates to be an arithmetic function of the previous states, the PRISM syntax enables us to construct concise representations of MDPs. Concretely, this helps us avoid specifying the transitions of each individual state of the MDP. Similar to how the PRISM syntax allows us to specify MDPs, we want to construct a space of *programmatic* policies that can exploit the symmetries in the MDP program

*Email: guruprerana.shabadi@polytechnique.edu, ACM student member number: 0299459



(a) (b) (c)

Fig. 1. (a) An example gridworld where the linear predicates are in green, the enabled directions are indicated by the arrows in each region, the entry point in red, and the target region indicated in blue (b) A policy (in red) for this gridworld which is a sequence of `LEFT`, `RIGHT`, `UP`, or `DOWN` actions forming a path from the entry to the target (c) Another gridworld which requires a looping path to reach the target

to obtain concise policy representations. In other words, we want to synthesize a programmatic representation to avoid explicitly storing all the state-action pairs of a policy. Our approach presents a novel practical method for programmatic reinforcement learning through the design of algorithms which work directly on the program syntax of an MDP to synthesize a programmatic policy. As a consequence, we also open up a rich set of theoretical questions on obtaining upper and lower bounds on the size of the policy with respect to the size and complexity of the MDP program.

2 REACHABILITY IN A 2D DETERMINISTIC GRIDWORLD

The PRISM syntax [4] can be used to programmatically specify an MDP over a finite state space described by a set of bounded integer valued variables. Boolean combinations of standard arithmetic predicates over the variables like `x + y >= 5 && y < 25` allow to partition the state space into regions. Within each region, a set of actions are enabled each leading to an update of the state variables such as `(x := x + y) && (y := y + 1)`. In the stochastic setting, these updates can be assigned a probability. In the deterministic case, the reachability objective asks whether there is a sequence of actions (i.e., policy) that will reach a given target state from an initial state.

In this work we consider 2D gridworlds of size $n \times n$ represented by two state variables `(x, y)` partitioned into regions along linear predicates over these variables. Each resulting region is thus a convex polygon. We equip each region with a subset of simple actions: `LEFT`, `RIGHT`, `UP`, and `DOWN` corresponding to the updates `x := x - 1`, `x := x + 1`, `y := y + 1`, and `y := y - 1` respectively. In other words, at each instant, we are only allowed to move one space in each direction. Instances of these gridworlds can be visualized as a generalization of a maze (see Fig. 1). Our objective is to find the most efficient representation of a path from the entry point to a point in the target region.

3 CONTRIBUTIONS

Subgoal-based programmatic policies. We introduce a class of programmatic policies that provides succinct policy representations for these 2D gridworlds by making use of an *edge-based policy data structure* and *memory* which stores information about the current trajectory. As is evident from the policy in Fig. 1b, it is sufficient to store the points on the edges of the polygonal regions through which the path passes. The edge-based policy encodes this in a map from each edge in the gridworld to a line segment on the edge of an adjacent region. We also observe in Fig. 1b that

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

Benchmark	Gridworld size (kB)	Policy size (kB)	Regions
size100preds30-4	266.8	303.5	271
size100preds50-1	612.9	655.3	616
size100preds50-2	668.6	706.8	668
size100preds50-4	538.5	576.5	542
size100preds50-5	603.3	641.7	616

we are required to go `RIGHT` and then later `LEFT` in the same triangular region in the middle. This motivates the use of memory in a programmatic policy because if we can store in memory that this region was already visited, then we can *switch modes* and only go `LEFT` the next time it is visited. Furthermore, in the case of the policy in Fig. 1c, we observe a looping path that visits each edge several times indicating us to incorporate a looping structure in our programs.

Combining these elements, we construct an algorithm that can produce a compressed programmatic representation of a policy. Our programmatic policies are composed of a sequence of subgoal blocks where each subgoal is an edge of a region. Within each subgoal block we have an edge-based policy which tells us where to navigate to from each edge in the gridworld. Intuitively, until the edge specified by the subgoal block is reached, the edge-based policy inside the block is repeated. Upon reaching this edge, it switches to the next subgoal block.

Theoretical bound on programmatic policies. Given an $n \times n$ gridworld with k convex polygonal regions, assuming that the endpoints of each edge of a region with rational coordinates can be represented by integers in the interval $[-D, D]$, we prove the following upper bound on the size of the synthesized policies.

Theorem 3.1. *If the target region is reachable from the entry point by a continuous path, then there exists a path with a programmatic representation of size $O(k^4 pD \log D)$ where p is the number of times the path changes directions.*

Importantly, note that the size of the policy given by the theorem is independent of the size of the grid n which means that we are more efficient than explicitly storing the set of state-action pairs. The proof of the theorem is presented in Appendix B.3. It requires a careful analysis of the path and decomposing it into a sequence of looping and non-looping segments giving us a sequence of subgoals.

Implementation and experimental results. Our implementation[1] of the synthesis algorithm can take a gridworld specification and construct a programmatic policy for it. We are able to generate images and videos of the policies along with the corresponding PRISM programs of the gridworlds. We tested our implementation on a set of benchmarks composed of randomly generated gridworlds with upto 50 linear predicates and 600+ regions. A few benchmarks are included in Table 1. We remark that the sizes of the synthesized policies are compact and comparable to those of the gridworlds themselves. The full set of results along with policy images are included in Appendix C.

Future work. This work presents a first step towards programmatic reinforcement learning. While we only address a class of deterministic 2D gridworlds, we hope that the ideas presented here of using memory and domain-specific data structures will help us progressively build programmatic policies for complex environments with more tricky predicates and stochasticity.

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

REFERENCES

[1] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html

[2] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 2499–2509. https://proceedings.neurips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR* abs/1912.06680 (2019). arXiv:1912.06680 http://arxiv.org/abs/1912.06680

[4] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[5] Pieter J. K. Libin, Arno Moonens, Timothy Verstraeten, Fabian Perez-Sanjines, Niel Hens, Philippe Lemey, and Ann Nowé. 2020. Deep Reinforcement Learning for Large-Scale Epidemic Control. In *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track - European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 12461)*, Yuxiao Dong, Georgiana Ifrim, Dunja Mladenic, Craig Saunders, and Sofie Van Hoecke (Eds.). Springer, 155–170. https://doi.org/10.1007/978-3-030-67670-4_10

[6] Johannes Pitz, Lennart Röstel, Leon Sievers, and Berthold Bäuml. 2023. Dextrous Tactile In-Hand Manipulation Using a Modular Reinforcement Learning Architecture. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*. IEEE, 1852–1858. https://doi.org/10.1109/ICRA48891.2023.10160756

[7] Xinghua Qu, Zhu Sun, Yew-Soon Ong, Abhishek Gupta, and Pengfei Wei. 2021. Minimalistic Attacks: How Little It Takes to Fool Deep Reinforcement Learning Policies. *IEEE Trans. Cogn. Dev. Syst.* 13, 4 (2021), 806–817. https://doi.org/10.1109/TCDS.2020.2974509

[8] Dhruv Mauria Saxena, Sangjae Bae, Alireza Nakhaei, Kikuo Fujimura, and Maxim Likhachev. 2020. Driving in Dense Traffic with Model-Free Reinforcement Learning. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*. IEEE, 5385–5392. https://doi.org/10.1109/ICRA40945.2020.9197132

[9] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://www.science.org/doi/pdf/10.1126/science.aar6404

[10] Niko Sünderhauf, Oliver Brock, Walter J. Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, and Peter Corke. 2018. The limits and potentials of deep learning for robotics. *Int. J. Robotics Res.* 37, 4-5 (2018), 405–420. https://doi.org/10.1177/0278364918770733

[11] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15726–15737. https://proceedings.neurips.cc/paper/2019/hash/5a44a53b7d26bb1e54c05222f186dcfb-Abstract.html

[12] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nat.* 575, 7782 (2019), 350–354. https://doi.org/10.1038/S41586-019-1724-Z

[13] Chiyuan Zhang, Oriol Vinyals, Rémi Munos, and Samy Bengio. 2018. A Study on Overfitting in Deep Reinforcement Learning. *CoRR* abs/1804.06893 (2018). arXiv:1804.06893 http://arxiv.org/abs/1804.06893

A REACHABILITY ON A DETERMINISTIC 2D GRIDWORLD WITH LINEAR PREDICATES

Here we will formally define the gridworlds that we work with. Our subclass of PRISM programs consist of only 2 state variables x, y representing a grid and guards composed of linear predicates over these variables. We also restrict ourselves to deterministic actions along the cardinal directions of LEFT, RIGHT, UP, and DOWN corresponding to the updates $x' = x - 1, x' = x + 1, y' = y + 1$, and $y' = y - 1$ respectively. The linear predicates partition the state space (2D grid) into regions where a certain subset of these actions are enabled. We also specify a target region as the reachability objective from an entry cell (x_0, y_0). We call this resulting class of games LINGRID which can be defined as follows.

*Definition A.1 (*LINGRID*).* An instance of LINGRID $\mathcal{L}$ is a deterministic reachability game defined on a 2D grid represented as a tuple $\mathcal{L} = (n, \mathcal{R}, \mathcal{A}, e, T)$ composed of the following components:

(1) $n \in \mathbb{N}$ is the size of the grid which forms the state space $\mathbb{S} = [\![0, n]\!]^2$. The state variables x, y range over this state space.
(2) A set $\mathcal{R} \subset \mathcal{P}(\mathbb{S})$ of pairwise disjoint subsets of $\mathbb{S}$ which forms a partition of the state space such that each $R \in \mathcal{R}$ is convex and polygonal, i.e., represents a subset of the grid bound by linear predicates over x, y.
(3) A set of actions allowed in each region given by a function
$$\mathcal{A} : \mathcal{R} \to \mathcal{P}(\{\texttt{LEFT}, \texttt{RIGHT}, \texttt{UP}, \texttt{DOWN}\}).$$
(4) An entry cell $e = (x_0, y_0) \in \mathbb{S}$ (usually $(0, 0)$) and a target region $T \in \mathcal{R}$.

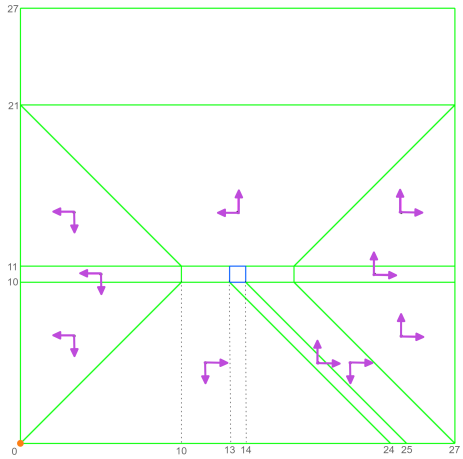
Each cell in $[\![0, n]\!]^2$ belongs to a unique region from which the player can move one cell along the four directions based on the actions that are allowed. The goal of LINGRID is to construct a policy that can guide an agent to navigate from the entry cell to a target state. Note that it is straightforward to encode LINGRID in the PRISM syntax.

Given an instance $\mathcal{L} = (n, \mathcal{R}, \mathcal{A}, e, T)$ of LINGRID, a *valid path* in $\mathcal{L}$ is a sequence of cells and actions $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots \xrightarrow{a_{p-1}} s_p$ such that each $s_i \in [\![0, n]\!]^2$ and each $a_i \in \mathcal{A}(R_i)$ where $R_i \in \mathcal{R}$ is the region containing s_i. Lastly, a path completes the game when $s_0 = e$ and $s_p \in T$.

We would like to build a concise *programmatic* representation of a *policy* which can guide an agent from the entry point to the target region. However, since we have a deterministic environment, this objective reduces to building a concise representation of a *path* from the entry to the target. Thus, we will be using path and policy interchangeably moving forward.

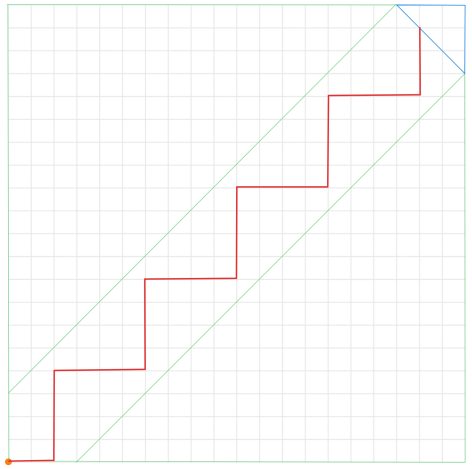
Example A.2 (Double pass triangle). In Figure 1a, we have an example of an instance of LINGRID. The state space is a 21×21 grid and there are 6 regions with the allowed actions in the regions indicated by arrows within the region. The entry point (in orange) is $(0, 4)$ and the target region is the triangular region in the top-left corner. Figure 1b visualizes the path from the entry point to the target region. Explicitly, the path of length 36 is given by the following sequence of states and actions.
$$(0, 4) \xrightarrow{\texttt{RIGHT}} (1, 4) \xrightarrow{\texttt{RIGHT}} (2, 4) \dots (15, 4) \xrightarrow{\texttt{UP}} (15, 5) \dots$$
Clearly we can come up with a more concise representation of this path since within these 36 steps, we only change direction twice.

Example A.3 (Spiral). In Figure 2 we have another instance of LINGRID where the policy has to loop around until it can reach the target region. Here too, we observe that within the loop, we repeat the same set of actions several times within the path which indicates that we should be able to synthesize a more compact representation.



(a) Gridworld

Fig. 2. Spiral



(a) Policy

Fig. 3. Diagonal

Example A.4 (Diagonal). In Figure 3a we have an instance of `LINGRID` in which the policy requires the agent to proceed in a diagonal direction by alternating between going `UP` and `RIGHT`.

B EDGE-BASED POLICIES

B.1 Relaxation to a continuous grid

Consider a natural continuous relaxation of the class `LINGRID` which permits us to abstract away the grid and consider policies that only navigate between edges of the regions.

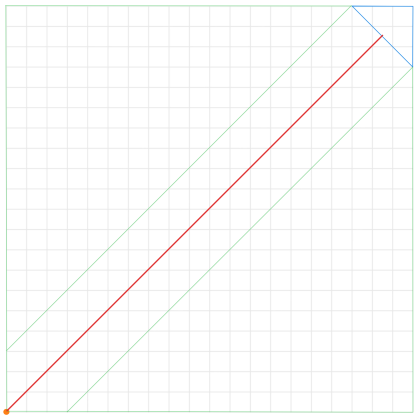


Fig. 4. Policy for the Example A.4 (diagonal) in the continuous case

*Definition B.1 (*CLINGRID*).* An instance of CLINGRID C is a deterministic reachability game defined on a bounded 2D space represented as a tuple $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ composed of the following components:

(1) $l > 0$ is the size of the state space $\mathbb{S} = [0, l]^2$.
(2) $\mathcal{R} \subset \mathcal{P}(\mathbb{S})$ is a finite set of convex polygonal regions which forms a polygonal tiling of the space $[0, l]^2$, i.e., each $R \in \mathcal{R}$ is the closed and convex region bound by a set of finite linear predicates. We further require that $\mathcal{R}$ covers $\mathbb{S}$ and that for all $R_1, R_2 \in \mathcal{R}$, $R_1 \cap R_2$ is either empty, a single vertex, or a single line segment. Also, no two edges of a polygonal region are allowed to be on the same line (colinear).
(3) $\mathcal{A}$ is again the function which associates each region with a subset of allowed actions in {LEFT, RIGHT, UP, DOWN}.
(4) $e \in \mathbb{S}$ is the entry point and $T \in \mathcal{R}$ is the target region.

A valid path in an instance of CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ is a sequence of states $s_1 \to s_2 \to \ldots \to s_p$ in $[0, l]^2$ such that each consecutive pair of states s_i, s_{i+1} is within the same region and the vector $s_{i+1} - s_i$ lives within the cone generated by the actions allowed within the region. Note that we associate the actions {LEFT, RIGHT, UP, DOWN} with the vectors $(-1, 0), (1, 0), (0, 1), (0, -1)$ respectively. Since some line segments and vertices are shared between regions, a path can traverse regions.

The key difference with LINGRID is that in the continuous version, the player is allowed to move to any point within the same region as long as it lies in a direction which can be *generated* by the set of allowed actions in the region. As a result, a path in an instance of CLINGRID can move directly from an edge of a region to another edge of the region in order to move between regions. In other words, in a path $s_1 \to s_2 \to \cdots \to s_p$, each s_i lies on an edge of a region. The length of the path p is then the number of regions traversed by the path.

Example B.2. In the continuous counterpart of the diagonal Example A.4, the policy just involves going in a diagonal path from the entry point to the target region as seen in Figure 4 because this diagonal direction is included in the cone generated by the allowed actions in the region (UP and RIGHT).

Example B.3. With Example A.2, considering the continuous variant, the path would just reduce to $(0, 4) \to (6, 4) \to (10, 4) \to (15, 4) \to 15, 16) \to (10, 16) \to (6, 16)$.

B.2 Backward winning region construction

Given an instance of CLINGRID C, we would like to construct the winning region which is the set of points in the state space from which there exists a valid path from the point to the target region.

Before describing the algorithm, we will define a few terminologies and notation that will be useful for us. Firstly, for any $v_1, v_2 \in \mathbb{R}^2$, we define $[v_1, v_2]$ to refer to the set of points on the line segment connecting v_1 and v_2. Next, let $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ be an instance of CLINGRID. For a region $R \in \mathcal{R}$, we use $\texttt{Edges}(R) = [[v_1, v_2], [v_2, v_3], \ldots, [v_r, v_1]]$ to refer to the list of edges of the polygon associated to the region where $v_1, \ldots, v_r \in [0, l]^2$. Assuming that each edge is shared uniquely between two adjacent regions, we can define $\texttt{Adj}(R, [v_i, v_j]) \in \mathcal{R}$ to be the region adjacent to R which both share the edge $[v_i, v_j]$ on their boundaries. However, since some edges lie on the boundaries of the grid $[0, l]^2$, the Adj might not always exist.

We will construct the winning region by iteratively building a set of trees, each of which is associated to an edge of the target region $\texttt{Edges}(T)$. The nodes of these trees are line segments in $[0, l]^2$. A *link* in a tree between two line segments $[v_1, v_2]$ and $[v_1', v_2']$ means that they are both parts of the edges of the same polygonal region $R \in \mathcal{R}$ and that there exists a valid path from each $v \in [v_1, v_2]$ to a point in $[v_1', v_2']$. Since each region is convex, we further have that there exists a path of length 1, i.e., for each $v \in [v_1, v_2]$, there exists $v' \in [v_1', v_2']$ such that $v' - v$ is in the cone of allowed actions in R. Observe that this link also implies the existence of a path from each point in the quadrilateral (v_1, v_2, v_1', v_2') to the segment $[v_1', v_2']$. This again results from the convexity of R which tells us that the quadrilateral is contained within R. The union of these quadrilaterals over the tree naturally gives us the winning region when the roots of the trees are the edges of the target region.

Thus, to construct these trees, we start from the edges of the target region, $\texttt{Edges}(T)$ which become the roots of our trees. Then we look at the edges of the regions adjacent to the target region and filter the part of the edges from which we can draw a link in the tree to a target edge. If such a segment exists, i.e., there exists a path to the target edge from each point in the segment, we add it as a node in our tree. We repeat this process by extending the leaves of the trees until the entry point e is in the winning region, or when none of the leaves can be extended anymore which happens when all the segments of the leaves lie on the boundary of the grid. However, to avoid re-exploring parts of the state space, before adding a segment to a tree, we filter out parts of the segment that are already present in a tree.

The pseudocode for this procedure is described in Algorithm 1 and it can be seen in action in Example B.4. We make use of the procedure "$\texttt{filter}([v_1', v_2'])$ `to reach` $[v_1, v_2]$ `in` R" which returns a segment $[v_1^*, v_2^*] \subseteq [v_1', v_2']$ which is the part of the segment from which there exists a valid path in R to $[v_1, v_2]$. Also, $\texttt{unexplored}([v_1, v_2])$ returns a list of segments within a segment that have not been explored so far. This can be facilitated by additionally storing a list of segments that have been explored within each edge. Lastly, $\texttt{add\_node}([v_1, v_2], [v_1', v_2'])$ adds a node $[v_1, v_2]$ in the tree with $[v_1', v_2']$ as the parent.

Example B.4. In Figure 5, we can visualize how Algorithm 1 works to construct the trees of the winning region. The trees itself can be seen in Figure 6. We have 3 trees, 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 valid path from each point in a node to a point in its parent node.

B.3 Subgoal-based strategy

Using the trees which we have constructed to describe the winning region in an instance of CLINGRID, it is straightforward to construct a path from the entry point to the target region,

Algorithm 1: Constructing winning regions

Input: An instance of CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$
trees = [new_tree($[v_1, v_2]$) for $[v_1, v_2]$ in Edges(T)]
extendable_leaves = queue([($[v_1, v_2]$, T) for $[v_1, v_2]$ in Edges(T)])
found_entry = false
while extendable_leaves is not empty and not found_entry **do**
  $[v_1, v_2], R$ = pop(extendable_leaves)
  R' = Adj($[v_1, v_2], R$)
  if R' exists **then**
    for $[v'_1, v'_2]$ in Edges(R') **do**
      $[v^*_1, v^*_2]$ = filter($[v'_1, v'_2]$) to reach $[v'_1, v'_2]$ in R
      for $[v''_1, v''_2]$ in unexplored($[v^*_1, v^*_2]$) **do**
        add_node($[v''_1, v''_2]$, $[v_1, v_2]$)
        push(($[v''_1, v''_2], R'$), extendable_leaves)
        if $[v''_1, v''_2]$ contains e **then**
          found_entry = true
      end
    end
end
ret trees

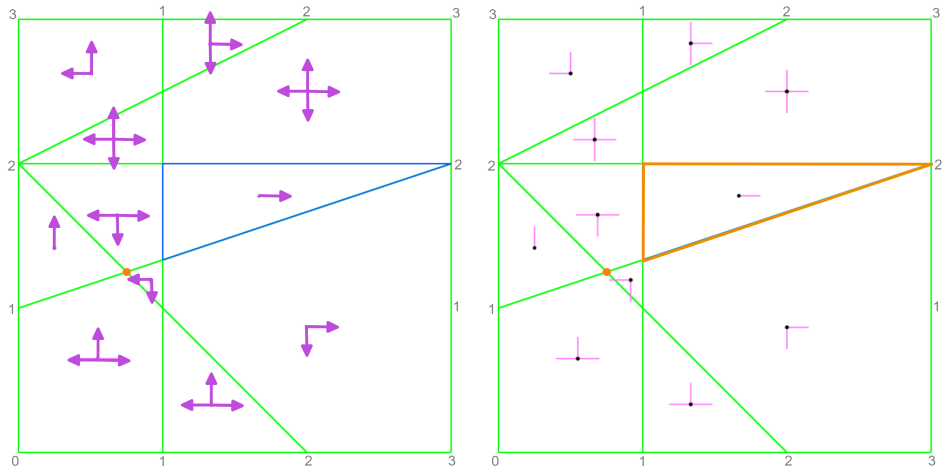
assuming that the entry point exists within the winning region. If the entry point belongs to a segment of a node in one of the trees, we can construct this path by simply following the segments in the tree from the node to the root, which is an edge of the target region. In fact, since from each point in a segment of a tree there might be more than one point in its parent segment that is reachable, every path in the tree from a node to one of its ancestors represents a family of valid paths in the CLINGRID instance.

Furthermore, all the paths in the trees combined represent a family of valid paths which all satisfy the property that they never visit the same point twice. In other words, all these paths are *non-self intersecting*. This is simply due to how we have constructed the trees in which we filter out parts of segments which we have already visited using the unexplored function while extending the leaves.

In the next step, observe that based on our primary objective of building a concise *programmatic* representation of a path from the entry point to the target region, it is now sufficient to obtain a programmatic representation of a path in a tree of the winning region from a leaf containing the entry point to the root. Thus, we consider a path in a tree from a node to one of its ancestors which is a sequence of segments the form $([a_1, b_1], [a_2, b_2], \ldots, [a_p, b_p])$, where each $a_i, b_i \in [0, l]^2$. Note that contrary to the ordering of the indices which we use here, during the *backward* construction, the last segment representing the ancestor is added to the tree before the first segment. But the sequence of segments from a node to one of its ancestors represents a family of *forward* valid paths. For example, in Figure 6, a path in a tree is $[(0, 2), (1, 5/2)] \rightarrow [(0, 2), (1, 2)] \rightarrow [(1, 4/3), (1, 2)]$.

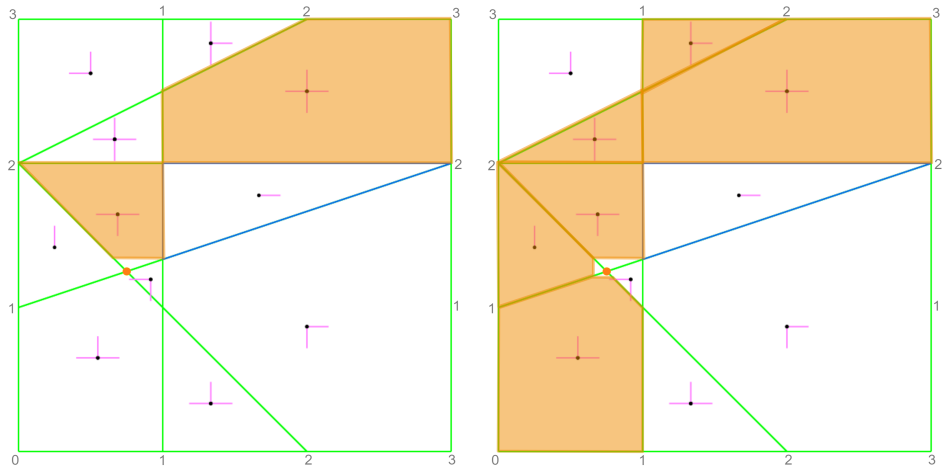
Before building a *programmatic* representation of such a path, we begin by defining the notions of equivalent and reduced sequences of segments. We then prove a few intermediary results which will be crucial in the later proof to show that we have a concise programmatic representation of an *equivalent reduced* sequence.

Consider an instance of CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ and a path in a tree of its winning region $([a_1, b_1], \ldots, [a_p, b_p])$.



(a) Instance of CLINGRID

(b) Trees at depth 0



(c) Trees at depth 1

(d) Trees at depth 3

Fig. 5. Backward construction of the trees of the winning region

Definition B.5 (Equivalent paths). Two sequences of segments $([a_1, b_1], \ldots, [a_p, b_p])$ and $([a'_1, b'_1], \ldots, [a'_{p'}, b'_{p'}])$ in C are equivalent if $[a_1, b_1] = [a'_1, b'_1]$ and $[a_p, b_p] = [a'_{p'}, b'_{p'}]$.

The intuition behind this definition is that if the first and the last segments are the same, then there exists a path in both the sequences from each point in the first segment to a point in the last segment.

Definition B.6 (Reduced path). A path $([a_1, b_1], \ldots, [a_p, b_p])$ is said to be reduced if for each $2 \leq i \leq p-1$, for all $y \in [a_i, b_i]$, there exists $x \in [a_{i-1}, b_{i-1}]$ such that there exists a valid path from x to y in C

In particular, in a reduced path we require that each point in a segment is reachable from a point in the previous segment.

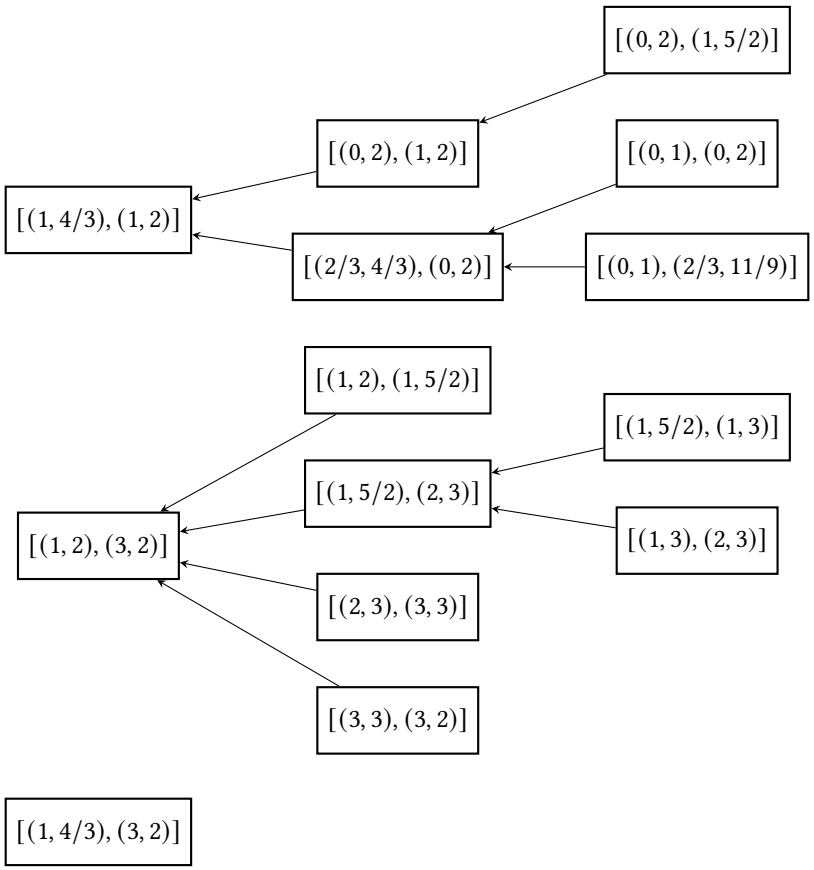


Fig. 6. Trees of the winning region corresponding to Figure 5 until depth 2

Lemma B.7. *Every path of segments in a tree of the winning region* $([a_1, b_1], \ldots, [a_p, b_p])$ *of an instance of* CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ *satisfies the following properties:*

(1) *The path induces a sequence of regions* $(R_1, \ldots, R_{p-1})$ *in* $\mathcal{R}$ *such that for each* $1 \leq i \leq p-1$, *every path between* $[a_i, b_i]$ *and* $[a_{i+1}, b_{i+1}]$ *is contained in the region* R_i, $\mathcal{A}(R_i) \neq \phi$, *and* $R_i \neq R_{i+1}$.
(2) *The path induces a sequence of edges of the polygonal regions* $(e_1, \ldots, e_p)$ *such that for each* $1 \leq i \leq p-1$, $[a_i, b_i] \subseteq e_i$ *and* $e_{i+1} \in \mathtt{Edges}(R_i) \cap \mathtt{Edges}(R_{i+1})$.
(3) *The path induces an order on the set of edges visited by the path* $\{\bar{e}_1, \ldots, \bar{e}_q\}$ *based on the first time an edge is visited by the path.*
(4) *We can construct a* ***reduced sequence*** *of segments which is* ***equivalent*** *to* $([a_1, b_1], \ldots, [a_p, b_p])$.

Proof. 1 and 2 can be deduced directly from the construction of the tree of the winning region as each node containing a segment is extended by traversing a region to another edge shared with an adjacent region.

The order on the set of edges visited can be computed using the sequence of edges from 2.

It is also easy to construct the reduced sequence: for each $1 \leq i \leq p-2$, starting from $i = 1$, filter out parts of $[a_{i+1}, b_{i+1}]$ which are not reachable from $[a_i, b_i]$. After filtering, the remaining part

remains a non-empty segment. There is necessarily at least one point in $[a_{i+1}, b_{i+1}]$ reachable from $[a_i, b_i]$ because there exists a path from each point in $[a_i, b_i]$ to $[a_{i+1}, b_{i+1}]$. It remains a segment because the cone reachable from $[a_i, b_i]$ is convex and we intersect it with the convex segment. □

For the next few lemmas, suppose we are given a path in a tree of the winning region $([a_1, b_1], \ldots, [a_p, b_p])$ which is assumed to be in reduced form as given by Lemma B.7.4 and consider the induced the sequence of pairs of regions and edges in C traversed by the path $((R_1, e_1), \ldots, (R_{p-1}, e_{p-1}))$.

Lemma B.8. *For each* $1 \leq i \leq p-1$, *suppose* $[a_i, b_i] = [(x_1, y_1), (x_2, y_2)]$ *and* $[a_{i+1}, b_{i+1}] = [(x'_1, y'_1), (x'_2, y'_2)]$. *Then,*

(1) *if* $\mathcal{A}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$, *then* $\min(y'_1, y'_2) = \min(y_1, y_2)$ *and* $\max(y_1, y_2) = \max(y'_1, y'_2)$.
(2) *if* $\mathcal{A}(R_i) \subseteq \{\texttt{UP}, \texttt{DOWN}\}$, *then* $\min(x'_1, x'_2) = \min(x_1, x_2)$ *and* $\max(x_1, x_2) = \max(x'_1, x'_2)$.

Proof. Consider the case when $\mathcal{A}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$. Since the region does not allow for any displacement in the y-direction, by construction, since we require that from each $x \in [a_i, b_i]$ there exists a valid path to $x' \in [a_{i+1}, b_{i+1}]$, we have that $\min(y'_1, y'_2) \leq \min(y_1, y_2)$ and $\max(y'_1, y'_2) \geq \max(y_1, y_2)$. But after considering the reduced sequence, since we further require that for each $x' \in [a_{i+1}, b_{i+1}]$, there exists a valid path from some $x \in [a_i, b_i]$, we obtain $\min(y'_1, y'_2) \geq \min(y_1, y_2)$ and $\max(y'_1, y'_2) \leq \max(y_1, y_2)$ and the result follows. Symmetrical arguments can be applied in the second case as well. □

Lemma B.9. *There exist at most two indices* $1 \leq i < j \leq p-1$ *such that* $(R_i, e_i) = (R_j, e_j)$ *and* $e_{i+1} = e_i, e_{j+1} = e_j$.

Proof. Arguing by contradiction, suppose there exist 3 indices $k < i < j$ such that $(R_i, e_i) = (R_j, e_j) = (R_k, e_k)$ and $e_{i+1} = e_i, e_{j+1} = e_j$ and $e_{k+1} = e_k$. Let us denote the common edge e_i by the segment $[a, b]$ and without loss of generality, assume for each of the segments $[a_i, b_i]$, $[a_{i+1}, b_{i+1}]$, $[a_j, b_j]$, $[a_{j+1}, b_{j+1}]$, $[a_k, b_k]$, $[a_{k+1}, b_{k+1}]$, 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 valid 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 $[a_j, b_j] \subseteq [b_{j+1}, b]$. So assume $[a_j, b_j] \subseteq [a, a_{j+1}]$. 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 $x - a_j$ in included in the cone of actions allowed 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 allowed 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 $[a_i, b_i] \subseteq [b_{j+1}, a_{i+1}]$ or $[a_i, b_i] \subseteq [b_{i+1}, b]$. Suppose we are in 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 7 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. Note that here we assumed $[a_j, b_j] \subseteq [a, a_{j+1}]$ and $[a_i, b_i] \subseteq [b_{j+1}, a_{i+1}]$, but each of the three other cases can be verified similarly that they all give us self intersecting paths. □

Lemma B.10. *Assume there exist indices* $1 \leq i < j \leq p-1$ *such that* $(R_i, e_i) = (R_j, e_j)$, $e_i \neq e_{i+1}$ *and* $\mathcal{A}(R_i)$ *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* $(e_1, \ldots, e_j)$.

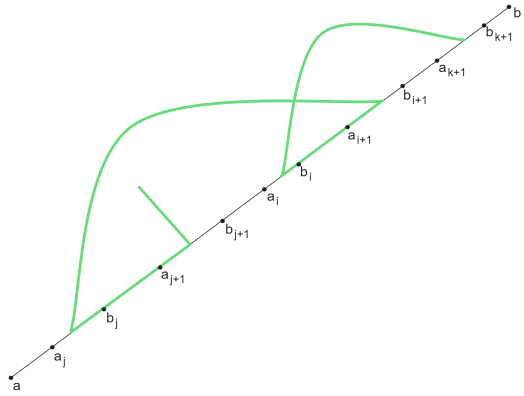


Fig. 7. Any path which visits $[a_{k+1}, b_{k+1}] \rightarrow [a_i, b_i] \rightarrow [a_{i+1}, b_{i+1}] \rightarrow [a_j, b_j] \rightarrow [a_{j+1}, b_{j+1}]$ is necessarily self intersecting.

(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* $j > i$ *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* $([a_1, b_1], \ldots, [a_j, b_j], [a'_{j+1}, b'_{j+1}], \ldots, [a_p, b_p])$ *is an* ***equivalent*** *sequence.*

Proof. Let us begin by understanding what we mean by a loop and edges being inside and outside loops. Looking at Figure 8a, 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 1 and the proof of 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 valid path going from p_i to p_j. Now take a look at Figure 8b 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 $p_k \rightarrow p_l \rightarrow p_i \rightarrow p_j$ which can also be seen in Figure 8b.

From 1, as $\mathcal{A}(R_i) \neq \phi$, without loss of generality, we assume that $\texttt{LEFT} \in \mathcal{A}(R_i)$ as seen in the figure. Using the assumption that we have at least two orthogonal directions allowed in R_i, we have two cases: $\texttt{UP} \in \mathcal{A}(R_i)$ or $\texttt{DOWN} \in \mathcal{A}(R_i)$. In the first case, p_{i+1} would be reachable from p_k and $[a_k, b_k]$ (see Figure 8c) 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 $\texttt{DOWN} \in \mathcal{A}(R_i)$, p_{j+1} would be reachable from p_i and $[a_i, b_i]$ (see Figure 8d) and therefore for the same reasons, $[a_i, b_i]$ cannot exist above p_j. This concludes the proof of 1.

Moving onto proving 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 $\texttt{LEFT} \in \mathcal{A}(R_i)$ and $x_{i+1} \leq x_i, x_{j+1} \leq x_j$. This can be visualized through Figure 9. As at least two orthogonal directions are allowed in R_i, let us again split into two cases with the first one being $\texttt{UP} \in \mathcal{A}(R_i)$. Firstly, $[a_j, b_j]$ is below $[a_i, b_i]$ as

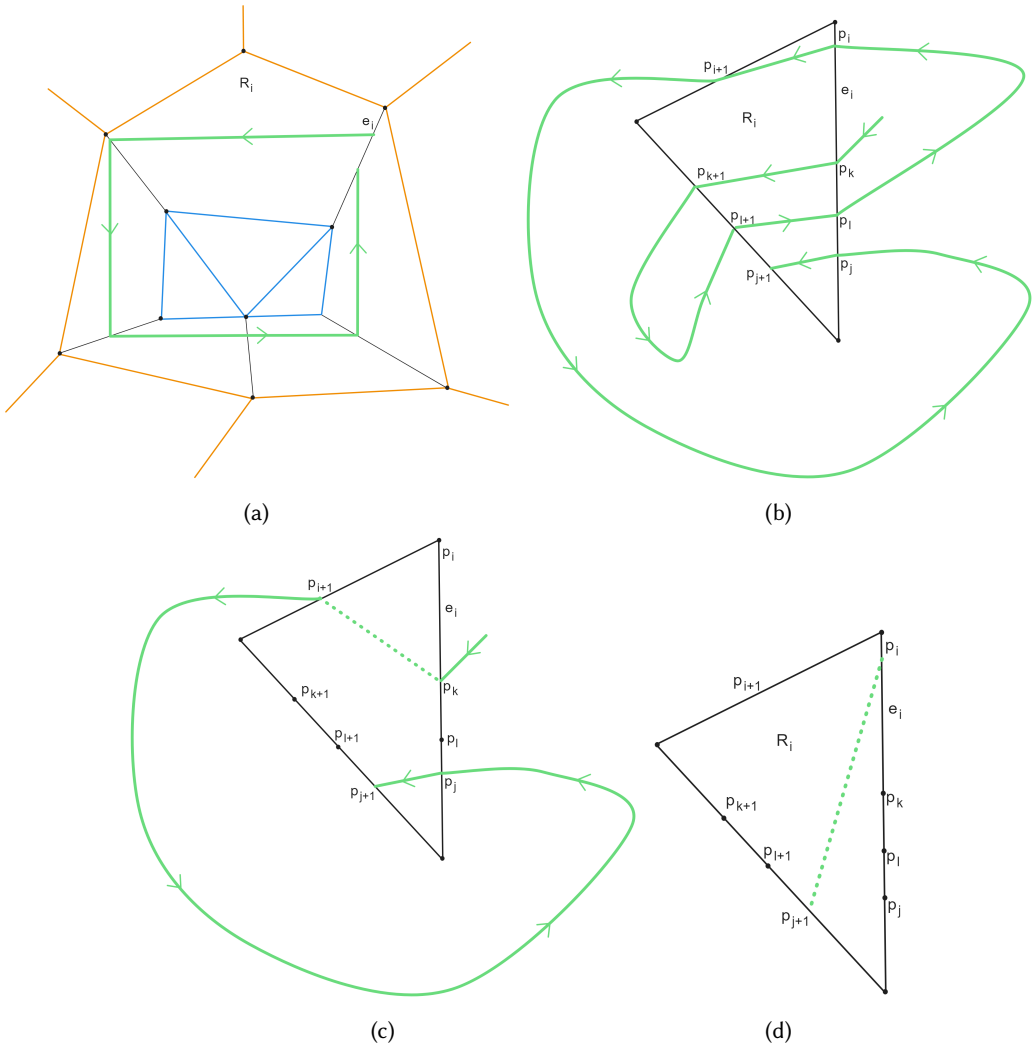


Fig. 8. (a) Inner loop at (R_i, e_i) partitions all the edges not involved in the loop into inner (blue) and outer (orange) edges (b) Loop at (R_i, e_i) with the path previously visiting an inner edge at index $k+1$ (c) Counterexample path in the case where DOWN and LEFT are allowed in R_i (d) Counterexample path in the case where UP and LEFT are allowed in R_i

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 $\texttt{DOWN} \notin \mathcal{A}(R_i)$, 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 polygonal region R_i. Lastly, $y'_j \leq y'_{j+1}$, i.e., b_j lies below b_{j+1} and therefore we can

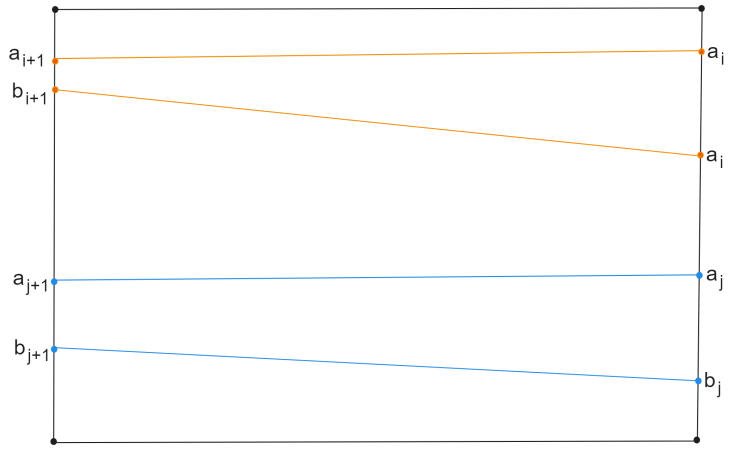


Fig. 9. Constructing $[a'_{j+1}, b'_{j+1}]$ from $[a_j, b_j]$, $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ in the proof of Lemma B.10

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. The modified sequence remains equivalent to the first one as there is still 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 $\texttt{DOWN} \in \mathcal{A}(R_i)$. □

Now suppose the path of segments in the tree visits q edges in the order $\bar{e}_1 \to \bar{e}_2 \to \cdots \to \bar{e}_q$. Then we have the following lemma.

Lemma B.11. *For all $1 \le m \le q-1$, after having visited m edges $\bar{e}_1 \to \cdots \to \bar{e}_m$, there exists a **reduced equivalent** sequence of segments going from $\bar{e}_m$ to $\bar{e}_{m+1}$ which admits a **programmatic representation** of size $O(m)$ assuming that each segment requires constant $O(1)$ space to store.*

Proof. Let $1 \le m \le q-1$. Denote the sequence of segments going from $\bar{e}_m$ to $\bar{e}_{m+1}$ by $([a_1, b_1], \ldots, [a_p, b_p])$ and the corresponding sequence of regions $(R_1, \ldots, R_{p-1})$ and that of edges $(e_1, \ldots, e_p)$ such that $e_1 = \bar{e}_m$ and $e_{p+1} = \bar{e}_{m+1}$. Note that the number of unique regions visited by $(R_1, \ldots, R_{p-1})$ is also at most m. Keep in mind the sequence of pairs

$$((R_1, e_1), \ldots, (R_{p-1}, e_{p-1})), \tag{1}$$

which will play an important role in the remaining proof. As each edge is shared uniquely among two regions, the number of unique pairs that occur in this sequence is at most $2m$.

As a first step, let us treat the indices $1 \le i \le p-1$ such that $\mathcal{A}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$ or $\mathcal{A}(R_i) \subseteq \{\texttt{UP}, \texttt{DOWN}\}$. We will handle the case where $\mathcal{A}(R_i) \subseteq \{\texttt{LEFT}, \texttt{RIGHT}\}$ and the other case can be dealt with symmetrical arguments. Firstly, we know that $\mathcal{A}(R_i) \neq \phi$ from Lemma B.7. In particular, $[a_i, b_i] \neq [a_{i+1}, b_{i+1}]$. By Lemma B.8, we know that if $[a_i, b_i] = [(x_1, y_1), (x_2, y_2)]$ and $[a_{i+1}, b_{i+1}] = [(x'_1, y'_1), (x'_2, y'_2)]$ then $\min(y'_1, y'_2) = \min(y_1, y_2)$ and $\max(y_1, y_2) = \max(y'_1, y'_2)$. First, consider the case where $y_2 \neq y_1$. We necessarily have that $e_{i+1} \neq e_i$. Furthermore, x'_1 and x'_2 can be determined directly by simply considering the other pair of x coordinates where the lines $y = y_1$ and $y = y_2$ intersects the polygonal region R_i. Note that by convexity, the two lines intersects the polygonal region in 4 unique points if $y_1 \neq y_2$. This helps us with our programmatic representation because if we know $[a_i, b_i]$, we can compute $[a_{i+1}, b_{i+1}]$ without requiring any additional storage. This computation can be visualized in Figure 10.

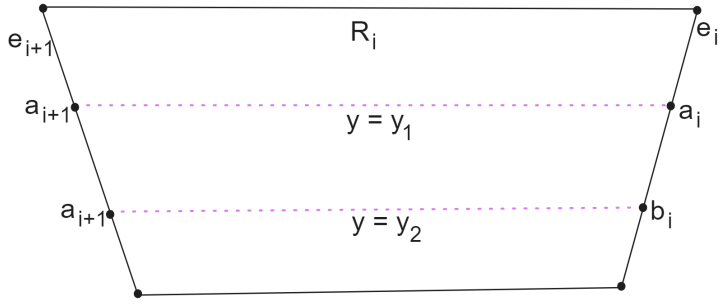


Fig. 10. The segment $[a_{i+1}, b_{i+1}]$ can be computed from $[a_i, b_i]$ by intersecting two horizontal lines when $\mathsf{LEFT} \in \mathcal{A}(R_i)$

The other case when $y_1 = y_2$, i.e., e_i is parallel to the x-axis, is more intricate. We remark that in this case $e_i = e_{i+1}$ because no two edges of a polygonal region are colinear. And so by Lemma B.9, the pair (R_i, e_i) is visited at most twice. So, we need to store at most two segments for all such regions which can again be done in space $O(m)$.

In the next big step, we need to treat the indices $1 \leq j \leq p-1$ such that $\mathcal{A}(R_j)$ contains at least two perpendicular directions. So let us restrict ourselves to the subsequence

$$((R_{j_1}, e_{j_1}), \ldots, (R_{j_{p'}}, e_{j_{p'}})) \tag{2}$$

where this condition holds. As a first case, assume that each pair in this sequence occurs at most thrice. In this case, $p' \leq 3 \times 2m$ because each of the $2m$ pairs of regions and edges is visited at most thrice and so we can explicitly store the whole sequence of segments in space $O(m)$. We now focus our attention on the other case in which we have a pair (R_t, e_t) which occurs at least four times in (2). Without loss of generality, we can assume that t is the least index such that (R_t, e_t) is visited four times. By Lemma B.9, out of the four times, at least two times, the consecutive edges will be different. So we can further assume that t and $t+d$ are the least indices such that $e_t \neq e_{t+1}$ and $e_{t+d+1} \neq e_{t+d}$. Since t is the least such index, we have $t \leq 3 \times 2m$ and $d \leq 3 \times 2m$. Define $z := \lfloor (p-t)/d \rfloor$. We first claim that for all $0 \leq u \leq z-1$ and $0 \leq v \leq d-1$, $(R_{t+ud+v}, e_{t+ud+v}) = (R_{t+v}, e_{t+v})$, i.e., starting from the index t, the sequence of pairs of regions and edges (1) becomes cyclic with period d. We also have that for the remaining path, for $0 \leq v \leq p-(t+zd)$, $(R_{t+zd+v}, e_{t+zd+v}) = (R_{t+v}, e_{t+v})$.

Before proving this, let us first apply Lemma B.10 to the pair (R_{t+d}, e_{t+d}) and assume we have an inner loop, so we obtain a set I of edges interior to the loop that do not belong to the m edges visited by the path until the index p. Next, let us begin by showing that $(R_{t+d+1}, e_{t+d+1}) = (R_{t+1}, e_{t+1})$. For the sake of contradiction, suppose $e_{t+d+1} \neq e_{t+1}$. Then necessarily since e_{t+d+1} is not an edge of the loop, $e_{t+d+1} \in I$. But this is in contradiction with the fact that none of the edges in I are visited until the index $p+1$. Repeating this argument inductively gives us the cyclicity result and it similarly follows for the case where we have an outer loop.

Lastly, by applying Lemma B.10.3, we can compute all the segments of the cycle if we know one iteration of the cycle. Thus, we can explicitly store all the segments up to the index $t \leq 6m$ in space $O(m)$ and then all the segments in one iteration of the cycle also in space $O(m)$ as $d \leq 6m$. □

Theorem B.12. *In an instance of* CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$*, for every path of segments in a tree of the winning region, there exists an equivalent reduced sequence of segments which admits a* ***programmatic representation*** *which requires storing* $O(\|\mathcal{R}\|^4)$ *segments.*

Proof. The theorem follows directly from Lemma B.11 as we have at most $\|\mathcal{R}\|^2$ edges and representing the equivalent reduced path between the m-th new edge and the $(m+1)$-th edge can be done in space at most $O(m)$ which means that we can represent the whole path in space at most

$$\sum_{m=1}^{\|\mathcal{R}\|^2} O(m) = O(\|\mathcal{R}\|^4)$$

□

In the previous theorem, we made an assumption that each segment in the sequence can be stored in constant space. This would imply that we can store reals 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 polygonal regions in the instance of CLINGRID C are described by rationals, we prove the following upper bound on the space required to store the segments.

Lemma B.13. *In an instance of* CLINGRID $C = (l, \mathcal{R}, \mathcal{A}, e, T)$*, 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*

$$\left(\frac{a}{D}l, \frac{b}{D}l\right)$$

for some $a, b \in [\![0, D]\!]$*, then each segment of a path of a tree of the winning region* $([a_1, b_1], \ldots, [a_p, b_p])$ *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.*

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 polygonal region in $\mathcal{R}$ 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:

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

$$x = \frac{x_2 - x_1}{y_2 - y_1}y + \frac{(y_2 - y_1)x_1 - (x_2 - x_1)y_1}{(y_2 - y_1)D}l \tag{4}$$

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 $H := D(D!)$ is divisible by $(x_2 - x_1)D$ and $(y_2 - y_1)D$. So we can write

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

$$x = \frac{s'}{H}y + \frac{d'}{H}l \tag{6}$$

where

$$s := H\frac{y_2 - y_1}{x_2 - x_1} \tag{7}$$

$$d := H\frac{(x_2 - x_1)y_1 - (y_2 - y_1)x_1}{(x_2 - x_1)D} \tag{8}$$

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

$$\|s\| = \|H\frac{y_2 - y_1}{x_2 - x_1}\| \leq H\|y_2 - y_1\| \leq HD \tag{9}$$

$$\|d\| = \|H\frac{(x_2 - x_1)y_1 - (y_2 - y_1)x_1}{(x_2 - x_1)D}\| \leq \frac{H}{D}(\|(x_2 - x_1)y_1 + (y_2 - y_1)x_1\|)$$
$$\leq \frac{H}{D}2D^2 = HD \quad (10)$$

With this, we are now able to write the equation for the line containing each of the edges as shown in 5. 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 5 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 5, gives us

$$y_{p-1} = \frac{s}{H}\frac{a}{H}l + \frac{d}{H}l = \frac{sa + dH}{H^2}l \quad (11)$$

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,

$$x_{p-2} = \frac{v}{H^3} \quad (12)$$

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

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

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

$$\log(H^pD^{p-1}) = O((p(D + 2) - 1)\log(D)) = O(pD\log(D)) \quad (14)$$

bytes. □

B.4 Programmatic representation of the policy

Theorem B.12 naturally provides us with an algorithm to produce a compressed representation of a path of segments and a *program* to reconstruct the segments. Since each edge visited by the path becomes a *subgoal*, we can do this by tracking in memory each time a new edge is visited and once a new edge is visited, switch to a different reconstruction strategy as given by Lemma B.11. If we can reconstruct the segments, we can also reconstruct a policy (or a path) from the entry point to the target region by picking at each step a point in the following segment that is reachable from the current position.

More concretely, to write down the *programmatic* representation of the policy, we can construct a program of the form given by Grammar 1. A programmatic policy is composed of a sequence of `UNTIL` blocks, each of which is associated with an edge of the instance of `CLINGRID` representing a subgoal. These blocks correspond to the sequence of edges the first time they are visited and the intended semantic is that *after* a certain edge is reached, we follow the policy under the `UNTIL` block. Within an `UNTIL` block, we can specify the target segments of each edge. The target segments

of an edge are a sequence of segments whose semantic with respect to the policy corresponds to navigating to a point in the *last* possible segment. Associated to each target, we can also specify a preference within the segment which can be one of `UP`, `DOWN`, or `NEUTRAL`. The preference `UP` within a target segment $[s_1, s_2]$ means that the policy should navigate towards the reachable point closest to s_2 and `DOWN` means that the policy should navigate towards the reachable point closest to s_1. `NEUTRAL` means that it does not matter which (reachable) point is chosen within the target segment.

$\mathcal{P}$	::=	$(\mathcal{U})^*$	*Programmatic policy*
$\mathcal{U}$	::=	`UNTIL edge:` $(\mathcal{S};)^*$	*Until block*
$\mathcal{S}$	::=	edge $\rightarrow$ $(\mathcal{T},)^*$	*Edge targets*
$\mathcal{T}$	::=	segment `PREFERENCE:` $\mathcal{B}$	*Target segment*
$\mathcal{B}$	::=	`UP` | `DOWN` | `NEUTRAL`	*Preference within target*
edge	::=	$e \in \cup_{R \in \mathcal{R}}$ `Edges`(R)	
segment	::=	$[s_1, s_2], s_1, s_2 \in \mathbb{Q} \cap [0, l]$	

Grammar 1. Syntax for programmatic policies for `CLINGRID`.

In Algorithm 2, we present the pseudocode we can use to synthesize a programmatic policy of the form given by Grammar 1. It takes in a path in a tree of the winning region $([a_1, b_1], \ldots, [a_p, b_p])$ 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 `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. 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 role of the preference can be clearly seen in Example ??.

On the other hand, Theorem B.12 ensures that the size of the program is at most $O(\|R\|^4 pD \log D)$. Note that we do not distinguish regions where at least two orthogonal directions are permitted like we do in the proof of Theorem B.12. However, this is unnecessary because the bound still remains the same and removing these would further reduce the size of the program but we do not do this in our current implementation.

Example B.14. The following code block contains a part of the policy synthesized by Algorithm 2 for the spiral example given in Example A.3.

```
...
Until([(0, 0), (10, 10)]):
[(10, 10), (0, 0)] ->
    [(13, 10), (22, 1)], Preference: Down

[(13, 10), (23, 0)] ->
    [(14, 10), (22, 2)], Preference: Down
    [(13, 10), (14, 10)], Preference: Neutral
...
```

Algorithm 2: Synthesizing a programmatic policy

Input: A path in a tree of the winning region $([a_1, b_1], \ldots, [a_p, b_p])$ and the corresponding sequence of edges $(e_1, \ldots, e_p)$

```
visited_edges = initialize empty set of edges
for i = 1 to p − 1 do
    if e_i ∉ visited_edges then
        add e_i to visited_edges
        start new UNTIL block
    if e_i in current UNTIL block then
        if a segment [a, b] of e_{i+1} is currently the last target segment of e_i then
            merge [a, b] and [a_{i+1}, b_{i+1}] into one segment
            if merged segment is [a, b_{i+1}] set preference to UP
            otherwise if it is [a_{i+1}, b], set preference to DOWN
        else
            add new target [a_{i+1}, b_{i+1}] to e_i with preference NEUTRAL
    else
        add e_i to current UNTIL block
        add new target [a_{i+1}, b_{i+1}] to e_i with preference NEUTRAL
end
```

This has been taken from the final UNTIL block which is the part of the policy used after visiting the edge $[(0, 0), (10, 10)]$ for the first time. We show the targets from two edges of the gridworld: $[(10, 10), (0, 0)]$ and $[(13, 10), (23, 0)]$. Figure 2a can help visualize these edges. From $[(10, 10), (0, 0)]$ we have one target segment $[13, 10), (22, 1)]$. This is a result of the merging of several segments when the same edge is visited by the path of segments multiple times within the loop. The preference of DOWN ensures that we navigate to the reachable point closest to $(13, 10)$ allowing the agent to progress closer to the target while looping. Without this preference, we might loop forever.

For the edge $[(13, 10), (23, 0)]$, we have two target segments. The first one being $[(14, 10), (22, 2)]$ and the second $[(13, 10), (14, 10)]$. Two targets means that we navigate to the first segment until the second one becomes reachable which happens in the last iteration of the loop. In fact, $[(13, 10), (14, 10)]$ is an edge of the target region.

Remark 1. Algorithm 2 exploits Theorem B.12 to create a representation that can reconstruct the sequence of segments in a forward fashion starting from the first to the last. This requires explicitly storing some segments which needs space $O(pD \log D)$ from Lemma B.13. However, if we choose to do the reconstruction in a backward fashion, we do not need to store segments but only the indices of the edges. But doing it in a backward fashion would mean that we have to reconstruct the whole path of segments before being able to obtain a forward policy.

B.5 Translating from continuous to a discrete policy

Equipped with the algorithm to synthesize programmatic policies for the continuous relaxation of our original problem, we would like to shift our attention back to the discrete case. In particular, we would like to see how we can exploit the programmatic policy for instances of CLINGRID to synthesize policies for their corresponding instances of LINGRID. However, this is not as straightforward as it may seem at first glance because as illustrated in Example B.15, there are cases where there exists a winning policy in the discrete case but none in the corresponding continuous instance.

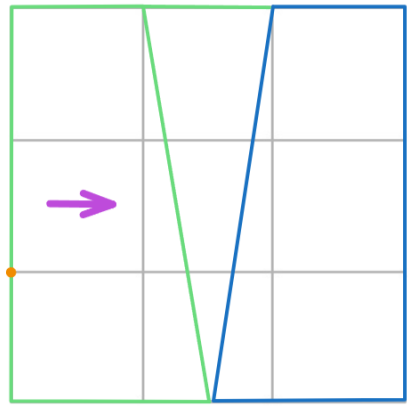


Fig. 11. Target region reachable in discrete case but not in the continuous

Example B.15 (Skipping regions). In Figure 11, we have an instance of LINGRID for which there exists a winning policy from the entry (orange) point but there is no such path in corresponding instance of CLINGRID. In the discrete case, while going to the right, we can jump from $(1, 1)$ directly to $(2, 1)$ which is in the target (blue) region. It is not possible to do this jump in the continuous case.

Let $\mathcal{L} = (n, \mathcal{R}, \mathcal{A}, e, T)$ be an instance of LINGRID and let $C = (l, \mathcal{R}, \mathcal{A}, e, T)$ be the corresponding instance of CLINGRID. To avoid the problem encountered in Example B.15, we assume n is large enough such that every path/policy in $\mathcal{L}$ is also a path in C. Increasing the size of the grid in $\mathcal{L}$ amounts to subdividing the grid into finer boxes. Let $s_1 a_1 \dots a_{p-1} s_p$ be the shortest path from the e to T in $\mathcal{L}$. Let $s'_1 \dots s'_{p'}$ be the corresponding path in C generated by the points on the edges of the regions where the path in $\mathcal{L}$ cross. Necessarily $s'_1 \dots s'_{p'}$ is not self-intersecting as we considered the shortest path. Furthermore, it belongs to a path of segments in a tree of its winning region. Therefore, with a programmatic policy for this path of segments, we also have a policy for $\mathcal{L}$: at each step, we can pick a gridpoint in the target segment which is at the boundary of two regions and navigate to this gridpoint.

In essence, although we cannot always use policies for CLINGRID as policies for LINGRID, by making the grid sufficiently big through subdivisions, there is a straightforward way to translate the policies to the discrete case.

C IMPLEMENTATION AND EVALUATION

Our implementation[2] in Python includes modules for generating instances of LINGRID and CLINGRID as well as implementation of the algorithms for the construction of the trees of the winning region and synthesis of policies. The `linpreds` module contains classes to generate random gridworlds with linear predicates. We do this by choosing at random linear predicates on a $n \times n$ grid where the endpoints of the linear predicates are in $[\![0, n]\!]$. 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

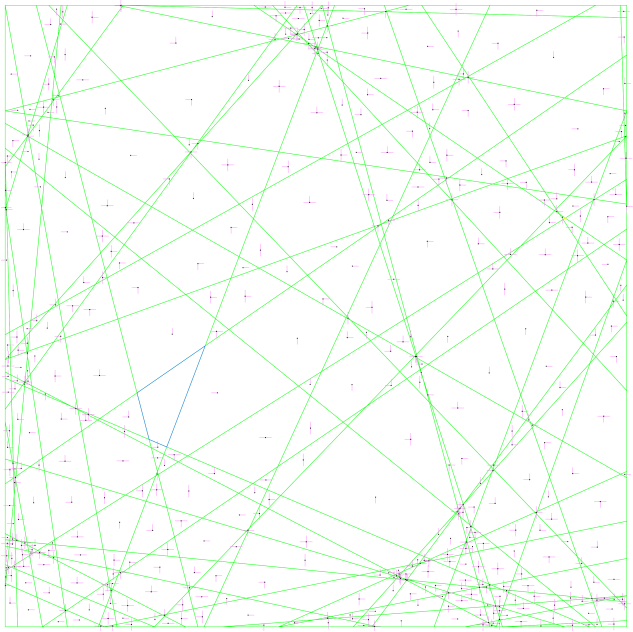
[2]https://github.com/guruprerana/smol-strats

Table 2. Size of synthesized policies (in bytes) for a set of generated benchmarks

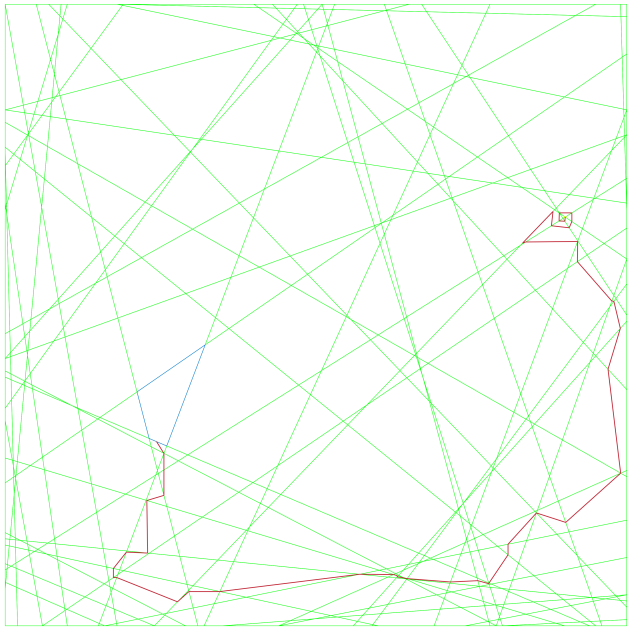
Benchmark	Gridworld size (bytes)	Policy size (bytes)	Regions
spiral	10833	20847	14
size3preds5loopy	8786	15744	11
size50preds10-1	30669	57926	32
size50preds20-1	105252	126263	113
size100preds20-1	119253	136257	126
size100preds20-2	108256	130591	115
size100preds30-1	228676	256031	233
size100preds30-2	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

as a planar tiling of the grid with polygons. The `backward_reachability` module implements Algorithm 1 to construct the trees of the winning region. The `game.continuous` module implements a reinforcement learning-like game environment which can simulate a policy for instances of `CLINGRID`. Lastly, `policy.subgoals` implements Algorithm 2 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 upto 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 2. Indeed, we can 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.



(a) Instance of CLINGRID



(b) Synthesized policy path

Fig. 12. The size100preds50-4 benchmark which is an instance of CLINGRID with 50 predicates and 542 regions