MSVIPER: Improved Policy Distillation for Reinforcement-Learning-Based Robot Navigation

Aaron M. Roth¹,²*, Jing Liang¹, Ram Sriram², Elham Tabassi², and Dinesh Manocha¹

Abstract—We present Multiple Scenario Verifiable Reinforcement Learning via Policy Extraction (MSVIPER), a new method for policy distillation to decision trees for improved robot navigation. MSVIPER learns an “expert” policy using any Reinforcement Learning (RL) technique involving learning a state-action mapping and then uses imitation learning to learn a decision-tree policy from it. We demonstrate that MSVIPER results in efficient decision trees and can accurately mimic the behavior of the expert policy. Moreover, we present efficient policy distillation and tree-modification techniques that take advantage of the decision tree structure to allow improvements to a policy without retraining. We use our approach to improve the performance of RL-based robot navigation algorithms for indoor and outdoor scenes. We demonstrate the benefits in terms of reduced freezing and oscillation behaviors (by up to 95% reduction) for mobile robots navigating among dynamic obstacles and reduced vibrations and oscillation (by up to 17%) for outdoor robot navigation on complex, uneven terrains.

I. INTRODUCTION

Learning methods are increasingly being used for robot navigation in indoor and outdoor scenes. These include deep reinforcement learning (DRL) methods [1], [2], [3], [4], [5], [6], [7] and learning from demonstration [8], [9]. These methods have been evaluated in real-world scenarios, including obstacle avoidance, dynamic scenes, and uneven terrains. In these applications, the performance of the navigation methods varies based on the underlying learned policy. It turns out that the learned policy may have errors or be otherwise sub-optimal. Additionally, the learned policy is often represented in the form of a neural net, which can be rather difficult to analyze.

In this paper, we are mainly interested in developing approaches to Reinforcement Learning that use a decision-tree (DT) policy. A decision tree can be more easily analyzed and modified, and the cause-and-effect of state and action is apparent. Learning a successful tree policy is typically more difficult than learning with a neural net. Moreover, these issues are compounded when one attempts to produce a policy for use beyond toy problems and proof-of-concepts [10], [11], [12]. We further investigate using the tree policy to enable modifying and improving upon a policy when the original learning pipeline becomes unavailable. In the real world, there will be situations where we want to take a learned policy for an autonomous robot, and analyze its shortcomings (including shortcomings defined or noticed after initial training!) and then fix those shortcomings. Retraining will not always be possible, nor will the original dataset always be available.

We introduce a novel method, Multiple Scenario Verifiable Reinforcement Learning via Policy Extraction (MSVIPER), and use it for improved robot navigation in complex indoor and outdoor scenes. Our formulation extends VIPER [13] and learns an “expert” policy using a neural net (such as PPO [14]) before using imitation learning to fit a DT to replicate the expert policy. MSVIPER compensates for some of VIPER’s limitations and also adds new capabilities. There has been limited work on applying VIPER to more complex tasks and almost no work on applying it to any real-world applications. Even for the tasks to which it has been applied, there is no recourse for addressing any errors or issues discovered.

We use MSVIPER to improve the performance of reinforcement-learning-based robot navigation methods. Instead of using different heuristics or trying different parameters and repeating the time-consuming training step, we integrate expert policies, policy extraction, and decision trees. Our policy distillation approach directly analyzes and performs tree modification operations on the DTs. Some of the novel contributions of our work include:

1) We present MSVIPER to improve imitation learning of DTs for indoor and outdoor robot navigation policies. MSVIPER samples from trajectories in multiple environmental scenarios. MSVIPER also results in smaller DTs, which are easier to understand and analyze [15]. We also demonstrate the superior sampling complexity of MSVIPER.

2) We describe techniques to modify and improve an initial tree policy without retraining, which could not be performed using a standard neural net policy. We use these improved tree policies to generate smooth
paths for collision-free robot navigation, to fix freezing problems in crowded indoor scenarios, and to reduce vibrations and oscillations on complex outdoor terrains.

3) We highlight the performance of improved policies on indoor and outdoor navigation scenarios. The indoor navigation is performed using a Jackal robot and Turtlebot among moving pedestrians and we observe up to 95% improvement in terms of reduction in freezing and oscillation behaviors. The outdoor navigation is performed using a Clearpath Husky and Jackal on uneven terrains with obstacles. We observe up to 17% improvement in terms of reduced vibrations and oscillations.

II. BACKGROUND AND RELATED WORK

There is considerable work on using machine learning methods for robot navigation, interpretation of networks [16] and decision trees [12]. A soft decision tree can be distilled from a deep neural network [10] or learned directly via a Policy Tree [17]. A tree can be learned in an additive manner using a reinforcement-learning-style approach [18] and Conservative Q-Improvement [19].

Policy distillation involves transformation of a policy from one format to another while keeping the essential input-output as similar as possible. This could be transforming a neural net or Markov Decision Process (MDP) into a smaller network or MDP [20], or into a different form entirely, such as a saliency map [21] or tree [22], [13]. For policy distillation, our approach is based on VIPER [13] because it is flexible in terms of being able to use any method to learn the expert policy. In practice, VIPER’s usage has been limited to proof-of-concept problems such as Cartpole [23], Pong [13] and other simulations such as CARLA [24].

Autonomous systems need the ability to deal with uncertain and unforeseen scenarios. Some existing systems reduce risk or correct errors by gathering information from a human through methods such as learning from demonstration [25], [26], [27]. Human feedback can also incorporate missing state features into state representation [28]. One approach asks a user to label states of interest as a means of gaining information during training [29]. The above-mentioned approaches incorporate human expertise while remaining black boxes. They typically require retraining, which can be time consuming. In our approach, we can evaluate the effectiveness and efficiency of the learned policy and directly improve without retraining.

III. PROBLEM FORMULATION AND NOTATION

As is typical in approaches that make use of reinforcement learning methods, we model the underlying task in the application as a Markov Decision Process (MDP), which could be represented by a 5-tuple \((S, A, P, R, \gamma)\). \(S\) is state space, \(A\) is action space, \(P\) is rewards, and \(P\) is state transition dynamics: \(S \times A \rightarrow S\), and \(\gamma \in (0,1]\) is the discount factor. Distribution of initial state is represented as \(\rho_0\). We want to generate an optimal policy \(\pi\) that maximizes discounted reward function:

\[
\eta(\pi) = \mathbb{E}_\pi \left[ \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right],
\]

where \(T\) is the time horizon. The policy is represented by networks with parameter \(\theta_s\).

In the Deep RL learning step, we want the robot to learn to accomplish one of the three given tasks, involving navigating to a static or dynamic goal position, avoiding collisions with static or dynamic obstacles, and achieving a smooth trajectory that avoids vibrations and oscillations.

This policy is black-box, it may be suboptimal, and it may have errors a certain percentage of the time. We seek to address such issues during the distillation and modification procedures described in Section IV.

We assume the researcher or developer working with the model has access to the model itself in the form of a black box, but does not have the capability to retrain it (simulating technological, temporal, organizational, legal, or other constraints that arise in the real world).

The notation we use is shown in Table I.

| Symbol | Description |
|--------|-------------|
| \(s \in S\) | a state \(s\) is an array representing the state of the world and the robot in it |
| \(a \in A\) | an action \(a\) is a single discrete action in the set of possible actions \(A\) |
| \(P\) | state transition dynamics |
| \(R\) | reward for a single \((s, a)\) pair or in general, respectively |
| \(\gamma\) | future discount factor |
| \(\pi\) | a policy (whether neural net or tree based) |
| \(\hat{\pi}\) | an expert policy learned in situ |
| \(\pi_1\) | a decision tree policy |
| \(l_i\) | length of trajectory to use when generating state-action pairs |
| \(M\) | number of trajectories used while generating state-action pairs |
| \(N\) | number of VIPER iterations used to ran per scenario |
| \(E\) | an ordered list of environments/scenarios (repetition is allowed) |
| \(n_{v_{0}}\) | the number of cross-validation trials used to compare the candidate final policies |
| \(l_{V_{1}}, l_{V_{2}}\) | unordered sets of state action pairs \(((s, a))\) |
| \(n_{s}\) | number of samples to use when generating \(l_{V_{1}}\) from \(l_{V_{2}}\) |
| \(\epsilon_{L}\) | Objective Efficiency, a measure of how efficiently a tree modification improved a policy; is higher when the target metric has higher improvement per node-modified |
| \(\epsilon_{R}\) | Relative Efficiency, a measure of how much a tree modification improved a policy; is higher when the target metric has higher improvement per percentage-of-nodes-modified |
| \(M_2, M_2\) | values of a target metric before and after tree modification |
| \(N_{\pi}\) | Number of nodes added, removed, or modified in a tree modification procedure |
| \(N_{\pi}\) | Number of nodes in \(\pi\) before modification |

TABLE I

SYMBOLS AND NOTATION USED IN THE PAPER

IV. OUR APPROACH: MSVIPER

We present a pipeline in Figure 2, the end-result of which is a modifiable decision tree policy for robot navigation. Section IV-A discusses use of deep reinforcement learning to create an initial learned navigation policy. Section IV-B describes policy distillation into a decision tree using MSVIPER. Section IV-C describes our tree modification procedure to improve the policy for robot navigation. We highlight its benefits in terms of addressing the issues related to freezing, oscillation, and vibrations in a robot’s trajectory.
A. Reinforcement Learning for Initial Navigation Policy

We solve the Deep RL task as formulated in In Section III We trained using a standard Deep RL algorithm such as the Proximal Policy Optimization (PPO) [14] algorithm. One important detail to note is that we trained our agent in a succession of increasingly difficult environment stages and not in a single stage. Many methods use Curriculum Learning (CL) [30] to learn good navigation behavior. In CL, instead of a single environment, the agent is trained in a succession of increasingly difficult environment stages. In CL, the policy trained by each scenario is reused as an initialization of the next scenario of training. In particular, the final policy from an earlier scenario becomes the starting policy for a later scenario. We used CL in the Deep RL stage, creating multiple scenarios of increasing difficulty in each case and use them to train a neural net format policy.

B. Policy Distillation: Conversion to Decision Tree Policy

We perform policy distillation on the initial policy to create a DT policy. First, we generate trajectories, which are sequences of state-action pairs \(\{(s_0, a_0), \ldots, (s_i, a_i)\}, s \in S, a \in A\) in a manner that is more sample efficient for some environments than previous approaches. Prior methods (VIPER [13]) would sample trajectories generated by applying the policy in a single simulated environment, and use those trajectories to train a decision tree. We refer to it as single source VIPER (SSVIPER). Instead, we sample the trajectories generated by applying the policy in multiple simulated environments of varying complexities or types. We try to ensure a large variety in terms of generating indoor and outdoor simulation environments. This variety ensures that the DTs that result from learning based on the sampled pairs will be more robust to different scenarios that might be encountered in the real world scenarios.

In MSVIPER (Algorithm 1), for each scenario \(e \in E\), we use this procedure: For a sequence of \(N\) iterations, we sample trajectories of length \(M\). All such pairs are put into a set. Pairs are randomly drawn from that set and used as a dataset to train a DT in a supervised learning manner using CART [31]. The overall dataset from which pairs are drawn is maintained between the stages. The resulting \(N\) decision trees are evaluated in simulation for \(n_{cv}\) trials, and the best-performing (highest reward) tree is the output of the method.

C. Tree Modification: Policy Optimization & no Retraining

We use our tree-based policy for analysis, verification, and modification. We modify the tree to improve the navigation characteristics of the policy without retraining. A key issue in terms of tree modification is to change as few nodes as possible. Details of our novel algorithms for different types of modifications to improve navigation can

---

**Algorithm 1:** Multi-Scenario VIPER: Our improved algorithm for learning a decision tree policy through imitation learning of an expert policy across multiple scenarios

1. **MSVIPER( \((S, A, P, R), \pi^*, M, N, E, l_t, n_{cv}, n_s)\):**
2. \(D \leftarrow \emptyset\)
3. for \(e \in E\) do
   4. \(s_0 \leftarrow \text{initial state after resetting } e;\)
   5. \(D_t \leftarrow \{ (s_j, \pi^*(s_j)) \sim d^{t-1} \}\) (sample \(M\) trajectories of length \(l_t\), by running policy \(\pi^*\) in environment scenario \(e\));
   6. \(D \leftarrow D \cup D_t;\)
   7. \(D' \leftarrow \{ (s, a) \mid (s, a) \in D \}\) (sample a dataset, retrieve a sample size of \(n_s\) pairs);
   8. Train decision tree \(\hat{\pi}_t\) using \(D'\) as dataset;
5. end
11. Return best policy \(\hat{\pi}_t \in \{\hat{\pi}_1 \to \hat{\pi}_N\}\) using \(n_{cv}\) trials to check each of them.
Our formulation is able to treat the vibration threshold as a hyperparameter and tune it to decrease the vibration in the learned policy. We can modify the tree by adjusting the threshold of certain nodes or by changing the action (and thus motion primitive) to reduce linear and/or angular velocity for that node. We are also able to modify the tree to reduce the oscillations, similarly to indoor navigation.

### D. Analysis: Upper Bound Divergence

Compared to SSVIPER, MSVIPER makes smaller and more efficient DTs.

**Theorem 1:** For MSVIPER, the loss function is:

\[
\hat{L}_i(s, \pi) = V_t^{(\pi^*)}(s) - \min_{a \in A} \hat{Q}_t^{(\pi^*)}(s, a),
\]

where

\[
\hat{Q}_t^{(\pi^*)}(s, a) = \mathbb{E}_\pi \left[ \sum_{c \in E} \left( \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}^c | S_t = s, A_t = a \right) \cdot w_e \right]
\]

\[
\hat{Q}_t^{(\pi^*)}(s, a)
\]

is the quality function, \(V_t^{(\pi^*)}(s)\) is the value function of the policy, \(R_t^c\) is reward at timestep \(i\) in environment \(e\), and \(w_e\) weights the contribution of each environment. In order to guarantee improved performance, we need to decrease the upper bound on \(\hat{L}_i(s, \pi)\).

**Lemma 1:** for a given \(\pi^*\), sampling multiple scenarios:

\[
J(\hat{\pi}) \leq J(\pi^*) + T \epsilon_N + \hat{O}(1)
\]

where \(J(\hat{\pi}) = -V(\hat{\pi}(s))\) be the cost-to-goal from the state \(s\). The \(\epsilon_N\) is the training loss defined in [13]. \(T\) and \(N\) are time steps and total training iteration numbers, respectively. \(\hat{O}(1)\) is a constant function. Equation \(4\) holds true.

The proof of the theorem is Appendix E.

### V. Results

#### A. Environments

We tested the performance of MSVIPER in three environments: i) an indoor environment with obstacles for collision-free navigation; ii) an indoor warehouse environment where a robot must find and follow a human, and iii) an outdoor environment with uneven terrain a robot must traverse.

In the indoor Obstacle Avoidance environment, the robot starts in a random location and must navigate around static and dynamic obstacles to a random goal location. We desire that the policy used for interpretation should perform well in terms of avoiding the pedestrians and obstacles.

In the indoor Find and Follow environment, the robot spawns in a complex multi-room environment with obstacles, a warehouse full of haphazardly arranged boxes. It spawns in a room. It must exit the room and find and follow a human in a white shirt around a warehouse. There are three stages in this task: i) learning the exit the room, ii) learning to exit the room and finding the human, iii) following the human as they move.

The outdoor navigation environment has a starting location and goal location and uneven terrain between them. The robot must navigate over the terrain safely and efficiently.
TABLE II
IMPROVED NAVIGATION USING MSVIPER

| Environment          | Policy Type | Final Avg Reward (test) | Size (# Nodes) | Depth | Runtime(s) |
|----------------------|-------------|-------------------------|----------------|-------|------------|
| Obstacle Avoidance   | Expert NN   | 0.23                    | n/a            | 25    | 8.84       |
| Obstacle Avoidance   | VIPER       | 0.30                    | 2651           | 16    | 7.59       |
| Obstacle Avoidance   | MSVIPER     | 0.097                   | 2073           | 22    | 51.00      |
| Find & Follow        | Expert NN   | 0.464                   | n/a            | 90.375 | 34.02     |
| Find & Follow        | VIPER       | 0.19                    | 1681           | 20    | 34.02      |
| Find & Follow        | MSVIPER     | 0.19                    | 1681           | 20    | 34.02      |

There are no obstacles in this environment but the challenge is optimally navigate over inclined or uneven outdoor terrain.

B. MSVIPER: Improved Navigation

MSVIPER achieves a higher average reward, smaller trees, and faster runtime than SS VIPER. As shown in Table II, we compared the performances by expert policies in simulation using MSVIPER and SSVIPER. MSVIPER generated DTs for the indoor navigation obstacle avoidance environment had a simulated reward near the expert policy level. Moreover, it has less nodes (smaller size) than the SSVIPER policy. MSVIPER exhibits better performance than SSVIPER. In the obstacle avoidance environment, The robot trained by MSVIPER has a better chance of going to the goal without any collisions. MSVIPER results in a smaller tree than VIPER, making it tree modification easier. In the Find and Follow scenario, MSVIPER results in smaller DT and improves the final average reward compared to SSVIPER. The superior reward for MSVIPER also indicates shorter navigation time for the robot. In all indoor environments, MSVIPER takes advantage of the curriculum stages to increase sample diversity and cover more critical states. It also reduces the chances of overfitting.

Figure 3 shows the sizes of trees as sample complexity changes. More samples result creation of trees that more closely mimic the expert policy. MSVIPER results in a smaller size tree than VIPER and that makes it easier in terms of tree modification.

C. Indoor Navigation Results

For indoor navigation, we validated our learned decision tree policy’s ability to handle navigation in a real-world environment using two robot platforms, Clearpath Jackal [36] and Turtlebot. We tested the policy on both robots in lab environments with various furniture and moving humans serving as obstacles. We observe significant reductions in the freezing behaviors with dynamic obstacles and fewer oscillations using MSVIPER and tree modifications (see Table II). We observe significant improvements, by 90–95%, by comparing the paths computed by the initial navigation policy verses the modified policy.

D. Outdoor Navigation Results

We tested our trees and also the expert policy in the real-world outdoor environment with uneven terrain using a Clearpath Husky robot. The trained trees have very similar performances to the expert policy. After the tree modification, the robot runs much more stably and smoothly. We highlight the benefits of policy distillation and tree modification in terms of reduced vibrations and oscillations on uneven outdoor terrains by 17% (see Table IV).

E. No-Retraining Tree Modification: Efficiency Metrics

First, for each navigation behavior (freezing, oscillation, vibration) we create a domain-specific procedure that can detect whether a given tree policy contains nodes that could result in those behaviors, and which detects those nodes. We are able to perform this analysis with a DT, but would be much harder with a neural net structure. Second, given the above, we can modify the policy to mitigate or fix the issue. We evaluate the policies according to navigation metrics before and after the trees are modified, to determine whether there was improvement. We can also measure the improvement with tree modification efficiency metrics.

In particular, we evaluate the benefits of tree modification using two metrics. Table III and IV show the Objective Efficiency $e_O$ and the Relative Efficiency $e_R$ of the modifications. Each metric is formulated with respect to a target metric $M$ that the tree modification is attempting to accomplish. In our demonstrations, $M$ is the percentage of trials where freezing occurs (for freezing), $C_{osc}$ (for oscillation), and $V_b$ (for vibration). If we need to make a larger change
in the initial policy to achieve the desired change in the target metric, it results in lower efficiency. These metrics are computed as:

\[
e_O = \frac{|M_2 - M_1|/M_1}{N_+}, \quad e_R = \frac{|M_2 - M_1|/M_1}{N_+/N_1},
\]

(5)

where \(M_1, M_2\) are the values of the target metric (either % of trials where freezing-occurs or % of trials where oscillation-occurs) before and after tree modification, respectively. \(N_1\) is the number of nodes before modification, and \(N_+\) is the number of nodes modified (changed or added). As shown, \(e_O\) is higher when the target metric has higher improvement per node-modified, and \(e_R\) is higher when the target metric has higher improvement per percentage-of-nodes-modified.

For example, in the best-case freezing fix for MSVIPER, running the freezing-detection algorithm revealed that 30 out of 199 nodes of the tree were configured such that they could lead to a freezing issue in the initial learned policy.

We analyzed many aspects of computing efficient DTs and tree modifications. One additional benefit of the tree modification metric, in addition to quantifying improvement, is higher efficiency of MSVIPER over SSVIPER. For oscillation reduction in indoor navigation, the tree improved using our MSVIPER method has a higher objective and relative efficiency than that improved using SSVIPER (the MSVIPER \(e_O\) and \(e_R\) are shown in Table III) VIPER has \(e_O = 0.032, e_R = 32.9\), both lower that MSVIPER’s). Since the initial MSVIPER policy closely mimics the expert policy, only 11 nodes require modification to fix the oscillation, as compared to 31. To improve the freezing behavior, the most-improved SSVIPER policy had an \(n_s\) of size 5K (shown in the Table) compared to the best performing MSVIPER at \(n_s = 1\)K. Freezing behavior was not present in all initial policies. MSVIPER at a 1K sample size resulted in no freezing behavior. In other words, MSVIPER resulted in a policy with fewer errors from fewer samples, and there were fewer errors to correct in the best-case policy (in the case of freezing). This highlights many benefits of MSVIPER in terms of efficient DTs and policy distillation.

F. Discussion of Implications

We have shown that policies from MSVIPER can be improved by tree modification without retraining. This is of critical importance in real-life applications. In some applications, we are often interested in the initial training process. In scenarios where the users will train and use RL models, however, there is great utility in a pipeline that allows for modifying a policy after-the-fact. This has applicability in situations where training is prohibitively expensive, for example, as well as situations where customization is useful. In particular, it allows improving and modifying policies by a software engineer or other developer without specific knowledge of reinforcement learning.

The examples involved here involved improving upon a particular metric, correcting an error, and even converting a failing policy into a successful one. One can imagine application developers creating solutions to error-prone or optimal learned policies and improving upon them further, delivering results beyond those that could be achieved by man or machine alone.

VI. CONCLUSION, LIMITATIONS, FUTURE WORK

We introduce MSVIPER, which learns a decision-tree style policy for complex navigation tasks by imitating a neural-net style policy and enable post-training modification. We demonstrate the validity of approach in simulation environments and in real-world scenarios in the context of a robot navigating obstacles in a crowd, a robot following a person in a warehouse and robot navigation in complex outdoor terrains. We show examples of our policy structure allows for modifying a policy after-the-fact. This has applicability in situations where training is prohibitively expensive, for example, as well as situations where customization is useful. In particular, it allows improving and modifying policies by a software engineer or other developer without specific knowledge of reinforcement learning.

Limitations exist: MSVIPER requires a discrete action space, a numerical array format state space, and hyperparameter tuning (its predecessors share these limitations). Our approach is best suited for scenarios where training can be expensive and it would be useful to reduce the overhead of retraining with different parameters.

Future work could involve using MSVIPER for applications of reinforcement learning beyond robot navigation. Additionally, our tree modification could be used on trees generated via other means.
[48] J. Zhang, B. Cheung, C. Finn, S. Levine, and D. Jayaraman, “Cautious adaptation for reinforcement learning in safety-critical settings,” in International Conference on Machine Learning. PMLR, 2020, pp. 11 055–11 065.

[49] G. Kahn, A. Villaflor, V. Pang, P. Abbeel, and S. Levine, “Uncertainty-aware reinforcement learning for collision avoidance,” arXiv preprint arXiv:1702.01182, 2017.

[50] J. Liang, Y.-L. Qiao, T. Guan, and D. Manocha, “Of-vos: Efficient navigation among pedestrians using commodity sensors,” IEEE Robotics and Automation Letters, 2021.

[51] T. Nyberg, C. Pek, L. Dal Col, C. Norén, and J. Tumova, “Risk-aware motion planning for autonomous vehicles with safety specifications,” in IEEE Intelligent Vehicles Symposium, 2021.

[52] M. Gleirscher, “Risk structures: Towards engineering risk-aware autonomous systems,” arXiv preprint arXiv:1904.10386, 2019.

[53] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[54] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, 2003, pp. 196–207.

[55] S. Primatesta, G. Guglieri, and A. Rizzo, “A risk-aware path planning strategy for uavs in urban environments,” Journal of Intelligent & Robotic Systems, vol. 95, no. 2, pp. 629–643, 2019.

[56] A. Hakobyan, G. C. Kim, and I. Yang, “Risk-aware motion planning and control using cvr-constrained optimization,” IEEE Robotics and Automation Letters, vol. 4, no. 4, pp. 3924–3931, 2019.

[57] B. C. Shah, P. Švec, I. R. Bertaska, A. J. Sinisterra, W. Klinger, K. von Ellenrieder, M. Dhanak, and S. K. Gupta, “Resolution-adaptive risk-aware trajectory planning for surface vehicles operating in congested civilian traffic,” Autonomous Robots, vol. 40, no. 7, pp. 1139–1163, 2016.

[58] S. Primatesta, G. Guglieri, and A. Rizzo, “A risk-aware path planning method for unmanned aerial vehicles,” in 2018 International Conference on Unmanned Aircraft Systems (ICUAS). IEEE, 2018, pp. 905–913.

[59] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu, “Interpreting deep learning-based networking systems,” in Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, 2020, pp. 154–171.

[60] M. Vasic, A. Petrovic, K. Wang, M. Nikolic, R. Singh, and S. Khurshid, “MoC: et: Interpretable and verifiable reinforcement learning via mixture of expert trees,” arXiv preprint arXiv:1906.06717, 2019.

[61] H. Ning, R. Yin, A. Ullah, and F. Shi, “A survey on hybrid human-artificial intelligence for autonomous driving,” IEEE Transactions on Intelligent Transportation Systems, 2021.

[62] S. Ross, G. Gordon, and D. Bagnell, “A reduction of imitation learning and structured prediction to no-regret online learning,” in Proceedings of the fourteenth international conference on artificial intelligence and statistics, 2011, pp. 627–635.
A. Additional Related Work

In this section find additional related work and discussion or previously mentioned related work.

Additional approaches for interpreting neural networks include Feature Leveling [37] and Saliency Graphs [38], [39], involve varying the weights of the network and drawing conclusions about the meaningful features. Formal verification can be performed when the state space of the algorithm (the space of all possible inputs to the network) can be fully enumerated [40], or by using other verification techniques that interrogate a black box, such as inductive synthesis [41]. One of the goals of interpretation is predictability and safety. This can be approached in a different way, by incorporating constraints into the learning procedure, as in Constrained Policy Optimization [42].

Autonomous systems must be able to deal with uncertain and unforeseen scenarios (risk reductions). One attempt to incorporate risk into RL is to include a risk penalty in the reward function [43]. Combining planning and RL (with a planner able to override, influence, or augment a learned policy) is a way to incorporate existing risk-based methods into learning methods, affecting executed actions [44], [45], [46], [47]. Similarly, safety constraints can be learned and used to affect action selection at execution time [48]. Safety constraints can also be used at training time (restricting or influencing action choices during training based on calculated risk) in order to train in the real world with less risk [49]. In contrast to the above methods, we combine autonomous learning with human domain knowledge. [50] models the Gaussian distributions of the errors in sensor measurements and object detection, but this modeling might be very conservative when the variances are high. [51], [52] estimate the severity of safety violations regarding state estimations. Risk management models are intended to analyze other models and systems and identify which subcomponents or sub-models are safe or dangerous. This risk measurement should also be also computationally efficient [51] to give the real-time evaluations.

Safety-critical applications can make use of decision trees, because (in addition to the benefits discussed in Section II) numerous existing methods [53] can analyze decision trees from a safety and verification standpoint [54].

One attempt to incorporate risk into RL is to include a risk penalty in the reward function [43]. Safety constraints can be incorporated by restricting actions during training [44], [45], [46], [47] or execution time [48]. In contrast to the above methods, we combine autonomous learning with human domain knowledge.

In the planning domain, there is extensive work on risk-aware systems [55], [56], [57], [58], [47]. The same is true when we look at learning methods for autonomous systems, including reinforcement learning.

We choose to extend VIPER [13] because, unlike other techniques, it is flexible in terms of being able to use any method to learn the expert policy. Moreover, it results in a single decision tree, and can provide verifiable guarantees. First, a neural net policy (or other similar policy) can be learned using reinforcement learning or another method (such as a planning method). This is considered the expert policy $\pi^*$. (Find a guide to notation in Table I.) Next, the state-action pairs are generated by executing the policy in the environment. Executing the policy generates trajectories, or sequences of state-action pairs. The state-action pairs from these trajectories are all combined into a large database. From the database (which includes all the pairs recorded), we in turn sample multiple datasets of state-action pairs. This is a crucial step, upon which our method improves. The Classification and Regression Tree (CART) method [31] creates a decision tree policy $\hat{\pi}$, from these datasets. In this supervised learning procedure, the elements of the state in each pair are the features of the tree and the action is the label. For each tree policy created, multiple trials are conducted and the reward is scored for each. The decision tree policy with the highest average reward is considered the best policy $\hat{\pi}$, which should best duplicate the expert policy. The structure of $\hat{\pi}$ is that of a binary tree. Each branch node evaluates a conditional expression regarding the feature space (which corresponds to the state space). Each leaf node class corresponds to one of the discrete actions.

Similar approaches have also been used to analyze Networking Systems [59] and Carla simulations [24]. MoET [60] is another work that seeks to improve on VIPER by using a mixture of multiple trees instead of a single tree. Our approach is motivated by these developments and is designed to handle complex tasks corresponding to systems with high degrees-of-freedom or challenging environments (e.g., a robot navigating through a dense crowd) using a single-tree approach.

B. Further Notes on Environments

In all cases, the state space includes the goal location in polar coordinates (from the perspective of the robot), the previous action taken, and the physical space around the robot as sensed by the lidar (and preprocessed into an occupancy grid). The lidar we use scans 512 ranges from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ radians (with 0 corresponding to straight ahead). In our preprocessing step this gets reduced to a radial occupancy grid. The dimensions of the occupancy grid can vary, and others could use different sizes than we chose. In our implementation we used 10 columns of equal radians and 7 rows, beginning 10 cm from the center-point of the robot and with heights of (listed in order from closest to most distant from the robot) 0.2 m, 0.2 m, 0.2 m, 0.3 m, 1 m, 1 m, 1 m. The occupancy grid information from the current timestep and the prior two timesteps is included in the state space. Therefore there are 210 features indicating obstacle position and movement, 2 features describing relative goal position, and 1 feature noting the action chosen by the agent in the previous timestep (total of 213 dimensions in the state space). We use a small discrete action space with motion primitives with varying discrete forward or angular velocities.
We have a simulated and real-world setup for the Indoor Obstacle Avoidance and Outdoor Navigation environments, and simulated environment for Indoor Find & Follow.

C. Tree Modification Algorithms

Unlike some other approaches that incorporate human input (such as [61]), in our approach we use human input to correct the errors while keeping the robot a fully autonomous entity at execution time. In this manner, we enable a synthesis of human and machine intelligence.

1) Freezing: A common issue in robot learning for navigation is the “Freezing Robot Problem.” [32], [33], [34], [35] Here the robot is presented with a set of obstacles and chooses to remain immobile. The purpose of choosing to remain motionless is to avoid crashing into an obstacle. However, it is also clearly undesirable since the robot has ceased to progress in the direction of the goal. One critical risk is that in the case of static obstacles causing this error, the failure case is one from which the robot cannot extricate itself.

The following procedure identifies nodes in a decision tree that cause (or could potentially cause) the freezing error, and then a subsequent procedure changes the attributes of the identified nodes to reduce the risk of the freezing error. The algorithm for determining which nodes are at fault is found in Algorithm 2.

| Algorithm 2: Detect Freezing |
|--------------------------------|
| 1 Detect Freezing Nodes( $\pi^f$, $a_F$, $m_A$ ) : |
| 2 $\mathcal{N} \leftarrow \emptyset$; |
| 3 for node $n \in \pi^f$ do |
| 4 | if $n$ is a leaf node then |
| 5 | $m_C \leftarrow$ the number of cells in the occupancy grid in which movement occurs; |
| 6 | if $m_C < m_A$ and $n[\text{action}] = a_F$ then |
| 7 | Add $n$ to $\mathcal{N}$; |
| 8 | end |
| 9 end |
| 10 Return $\mathcal{N}$ |

where $\pi^f$ is the tree policy, $a_F$ represents an action or collection of actions corresponding to a “stop” action, and $m_A$ is a configurable parameter indicating “the maximum number of polar occupancy grid cells that can contain movement while still considering the obstacles as static.” (See Appendix Section E for an explanation of the occupancy grid.) The algorithm analyzes each leaf node in the tree. If the node’s action is the Stop action, and if the occupancy grid described by the state bounds of that node indicate stationary obstacles (within a tolerance controlled by $m_A$), then the node is considered to be a problematic node and is added to the list of potential-freezing nodes. (Sometimes we may not want to be too strict about disallowing movement among the obstacles, necessitating the $m_A$ parameter. Setting $m_A = 0$ means that a node will not be considered problematic unless the obstacles are perfectly still, and setting $m_A$ at the maximum will cause the algorithm to place all nodes with the stop action $a_F$ into the problematic set regardless of the obstacle movement and position implied by the state subspaces. By “subspace” we mean a bounded subset of the state space.) An edge case is when the node’s subspace dimensions encompass both moving and non-moving possibilities. (In other words, the boundaries of a leaf node include states where a particular occupancy grid cell is static and states where that particular occupancy grid cell indicates movement over the three timesteps.) In this case, the algorithm will evaluate the condition as true (“stationary”) so long as the bounds of those dimensions remain the same for all timesteps. This is because even though movement might occur sometimes, the situation where an obstacle is in fact motionless is included in this subspace. In other words, while there might be some movement in a particular cell associated with this leaf node, a static obstacle in that cell is also associated with this same leaf node, and so a problematic state-action pair is associated with this leaf node even though non-problematic state-action pairs are also associated with it.) The algorithm intended to alleviate this issue is found in Algorithm 3, where $a_R$ and $a_L$ are actions corresponding to pure right and left rotation (no linear velocity), respectively. This should safely allow the robot to find an observed state where it can extract itself from stasis. (This may result in a non-optimal path, but it will not be less optimal than failing to complete the task completely. Some paths that do complete the task may be made slightly less optimal as a result of these changes. In the real world, there are times when this tradeoff will be desired.)

2) Oscillation: The next issue we address with regards to correcting the imperfect expert policies is the issue of oscillation. While traveling around obstacles, the robot will occasionally alternately rotate too far towards and away from the impediment. This behavior is inefficient and off-putting to a human observer. As before, we develop a procedure to mitigate the issue. We run the policy in simulation, records the robot’s behavior, and note which parts of the decision-tree policy are activated when the undesirable behavior occurs (contributing to the behavior). We modify the tree by modifying existing nodes or adding nodes, replacing existing actions with replacement actions that have lower linear and angular velocities. The procedure for detecting the nodes re-
Algorithm 4: Detect Oscillation

1. Define Oscillation Nodes( \( \pi^\dagger, E, F_O, n_e, L \)):
2. Initialize an empty queue \( Q \):
3. \( \pi^\dagger \leftarrow \emptyset \):
4. \( O_C(i) \leftarrow \emptyset \quad \forall \quad i \in \{ \text{ids of nodes in } \pi^\dagger \} \):
5. \( O_X(i) \leftarrow \emptyset \quad \forall \quad i \in \{ \text{ids of nodes in } \pi^\dagger \} \):
6. For each episode do:
   1. Reset environment \( E \):
   2. While \( E \) does not indicate episode done do:
      1. \( s \leftarrow \text{get current state from } E \):
      2. \( a \leftarrow \pi^\dagger(s) \):
      3. Append \((s, a)\) to \( E \), removing the oldest if the length of the queue is \( > L \):
      4. If \( F_O(\cdot(i)) \) then:
         1. Add \( \{n_i, n_{i+1}, \ldots, n_{i+L}\} \) to \( \pi^\dagger \):
         2. Add all \( s \in \mathbb{H}^L \) to \( O_C(d_i), O_C(d_{i+1}), \ldots, O_C(d_{i+L}) \), where \( d_i \) is the corresponding id of each node in \( \{n_i, n_{i+1}, \ldots, n_{i+L}\} \):
   5. Else:
      1. \( n_i \leftarrow \text{leaf node in } \pi^\dagger \) corresponding to \( s \):
      2. Add \( s \) to \( O_X(d) \), where \( d \) is the id of \( n_i \):
   6. Execute action \( a \) in environment \( E \):
7. Return \( \{O_C(i), O_X(i)\} \):

Algorithm 5: Alleviate Oscillation

1. Alleviate Oscillation Nodes( \( \pi^\dagger, N, C, O_X, z \)):
2. If \( O_X(i) = \emptyset \) or \( z \) then:
   1. Return updated \( \pi^\dagger \):
3. Else:
   1. Add all states visited on this node are oscillation
   2. \( n_i[\text{action}] \leftarrow \text{action with linear and angular velocity of reduced magnitude; } \)
   3. \( n_i \) is turned into a branch node with \( X \) and \( C \) as children, splitting on the best split that best separates the states in \( O_C(i) \) to node \( C \)
   4. and states in \( O_S(i) \) to node \( X \):
5. Return updated \( \pi^\dagger \):

Algorithm 6: Detect Vibration Method

1. Detect Vibration Relevant Nodes( \( \pi^\dagger, V_E \)):
2. \( N \leftarrow \emptyset \):
3. For each node \( n \in \pi^\dagger \) do:
   1. If \( n \) is not a leaf node then:
      1. Add \( n \) to \( N \):
4. Return \( N \):

Algorithm 7: Detect Vibration Method

1. Define Vibration Method
2. For each node \( n \) in \( \pi^\dagger \) do:
   1. If \( n \) is not a leaf node then:
      1. Add \( n \) to \( N \):
3. Return \( N \):

3) Vibration: We introduce two tree modifications aimed at addressing excessive vibration. These can be used to address vibration when the learned policy does not sufficiently reduce vibration below the threshold, or when one wants to modify the vibration threshold after the fact without retraining.

The first approach involves treating the threshold of nodes

\[
V_b = \sum_{t=t-3}^{t} \gamma^{t-t'}(|\omega_r| + |\omega_p|) \tag{6}
\]

sponsible for the undesirable behavior is found in Algorithm 4, where \( E \) is an environment, \( F_O \) is a function that accepts a state-action pair sequence and returns a boolean describing whether the trajectory sequence experienced oscillation, \( L \) is the sequence length, and \( n_e \) is the total number of episodes to monitor. The procedure to mitigate the excessive oscillation is found in Algorithm 5. Each node that has a subspace corresponding to an observation of oscillation is given two children nodes (and itself becomes a branching node). One of the children corresponds to the problematic subspace and the other corresponds to the subspace where no problems were observed. The latter node is given the action of the original node. The new child node corresponding to the problematic behavior sub-space is assigned a lower magnitude velocity action. In Algorithm 5, each \( O_C(i) \in O_C \) is a set of states in state subspace of node \( i \) where oscillation occurs, each \( O_X(i) \in O_X \) is a set of states in state subspace of node \( i \) where oscillation does not occur, and \( z \) is a boolean.

3) Vibration: We introduce two tree modifications aimed at addressing excessive vibration. These can be used to address vibration when the learned policy does not sufficiently reduce vibration below the threshold, or when one wants to modify the vibration threshold after the fact without retraining.

The first approach involves treating the threshold of nodes
and notes that this also describes a closed hyperdimensional object. Specifically, for the four timesteps and two angular velocity values present above, it describes an eight-dimensional object. This object is a hyperrectangle and a convex hull. We call this the "Vibration Space."

Table V: Expanded Action Space 2, used in the outdoor scenarios only

| Action ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|-----------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| Linear Velocity | 1 | 0 | 1 | 0 | 0 | 0.4 | 0 | 0 | 1 | 1 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 |
| Angular Velocity | -1 | -1 | 0 | 0 | 1 | 1 | 0 | -0.4 | 0.4 | -0.4 | 0.4 | -1 | 1 | -0.4 | 0.4 |

In Algorithm 9, see the procedure for modifying the tree based on the identified nodes. The algorithm makes a modification to the action of all of the leaf nodes descended from the nodes which have boundaries intersecting the vibration space. In general, to reduce vibration, magnitude of velocities is decreased. The specific mapping is provided by $M_c : T \rightarrow T$, where $T$ is the action type (in our case integer). In our case, we used $M_c = \{0 \rightarrow 13; 1 \rightarrow 7; 2 \rightarrow 6; 3 \rightarrow 3; 4 \rightarrow 14; 8 \rightarrow 6; 7 \rightarrow 8; 9 \rightarrow 13; 10 \rightarrow 14; 11 \rightarrow 13; 12 \rightarrow 14; 13 \rightarrow 13; 14 \rightarrow 14\}$, with the action space described in Table V.

D. Limitations

MSVIPER requires a discrete action space and a state space in the format of a numerical array. In addition, MSVIPER (like its predecessors) uses hyperparameters that must be tuned for optimal results. These include sample size $n_S$, number of iterations $n$, and trajectories generated per
When we train the decision tree in multiple environments, $\hat{e}$ is the quality function. In order to guarantee a better policy, we have the weight for each environment $l$.

According to the Q-Dagger algorithm, the hinge loss function would be guaranteed to have a better imitating performance. After training by training set $E$, we have:

$$Q^*_{\pi}(s,a) = \mathbb{E}_{\pi} \left[ \sum_{t=0}^{T} \gamma^t R_{t+k+1}^e | S_t = s, A_t = a \right] \cdot w_e$$

Replace $Q^*_{\pi}(s,a)$ with $\hat{Q}^*_{\pi}(s,a)$ we have:

$$\hat{l}_t(s, \pi) = V_t^*(s) - \min_{a \in A} \hat{Q}_t^*(s,a).$$

According to [13], while in training, the loss function can be calculated by:

$$Tl(\pi) = J(\pi) - J(\pi^*)$$

where $\pi$ is any policy trained by the decision tree, and $\pi^*$ is the policy optimally imitating expert policy. For multiple scenarios, we obtain:

**Lemma 1:** Let $\pi^*$ be held constant, Equation [17] holds when using samples from different scenarios.

$$J(\hat{\pi}) \leq J(\pi^*) + T \epsilon_N + \hat{O}(1)$$

**Proof:** [13] proposed equation [17] for single stage training. (See [13], [62] for more details on the $\hat{O}(1)$.) According to [62], for each trajectory in different scenarios $J(\pi) = T \mathbb{E}_{s \sim d^\pi} [C_{\pi}(s)]$ holds. Since sampled trajectories are independent, we can simply add the trajectories of different scenarios into states distribution class D, then the function $\epsilon_N$ holds. As a result, the policy $\hat{\pi}$ satisfies equation [17].

As per Lemma [1] the difference between an expert policy and a well-trained policy by MSVIPER is also upper bounded. In the next section we will prove that MSVIPER can decrease the upper bound of a loss function, and thus that MSVIPER has a better performance than VIPER.

**2) Sampling Complexity of MSVIPER:**

**Theorem 3:** $P_M(\epsilon | S_{C}, l, m, E) \geq P_V(\epsilon | S_{C}, l, m)$ and $u_M \leq u_V$ (MSVIPER has a sample complexity equivalent or superior to that of VIPER) if $\tilde{\gamma}_k^{n,t} < \tilde{\gamma}_k^{n,E,t}$ for any $k$ and $i \neq n_E$.

We define “sampling complexity” as a measure of how many state-action pair tuples (and thus trajectories generated) are required from the expert policy to learn a decision tree that achieves a given threshold of performance (as measured by average reward achieved compared to the expert policy).

We assert that, given certain assumptions stated in this section, MSVIPER has a sample complexity as good as or better than VIPER. This is shown when the upper bound $u$ for the divergence in reward between the expert policy and the tree policy is, in the MSVIPER case, smaller than or equivalent to the VIPER case.

Sample complexity is improved when the likelihood that high accuracy is achieved on critical states increases.
We look at this upper bound \( u \), described above and in [13] as

\[
Q_t^{(\pi)}(s, a) - Q_t^{(\pi^*)}(s, \pi^*(s)),
\]

for all \( a \in A, s \in S \), and \( t = 0, \ldots, T - 1 \). (18)

It is noted that \( u \) “may be \( O(T) \) ...if there are critical states such that failing to take action \( \pi^*(s) \) in state \( s \) results in forfeiting all subsequent rewards.” This is the case, for example, for the navigation environment, where a collision results in an immediate failure. The states where possible collision is imminent are the “critical states.” The same principal can be applied to other domains. In this instance \( u = O(T) \). It is further noted that this bound is \( O(T) \) “as long as \( \hat{\pi} \) achieves high accuracy on critical states.” It follows that any improvement to the method that results in increasing the likelihood that high accuracy is achieved on critical states improves the likelihood of the bound being \( O(T) \) as a opposed to a higher value.

Let \( S_C \subseteq S \) be the set of all critical states. Let \( k_i \in S_C \) for \( i = 1 \) to \( K \) be a list of all critical states. We assume that the definition of whether a state is considered a critical state is the same across scenarios. We assume that \( S_C \) is a finite set.

Consider the sampling step of our algorithm

\[
D_i \leftarrow \{(s_j, \pi^*(s_j)) \sim d^{n-1}\}
\]

(19)

Assume that the expert policy \( \pi^* \) chooses correct actions for critical states, and that the decision tree \( \hat{\pi} \) could thus only learn to fail on critical states if it fails to sample from critical states during trajectory generation.

Let \( S_{k,l} \) be a binary random variable, with the associated probability distribution being the Bernoulli distribution with probability \( p = p_k \) (as in, \( P(S_{k,l} = 1) \)). (Note \( 0 \leq p_k \leq 1 \). For a given generation of a trajectory of length \( l \) drawn from environment \( e \), \( S_{k,l} = 1 \) represents the outcome that the critical state \( k \) is sampled as part of the trajectory. \( S_{k,l} = 0 \) represents the outcome that the critical state \( k \) is not among the trajectory (although other critical states may be). Let \( S_{k,l} \) be a similar binary random variable, where an outcome of \( S_{k,l} = 1 \) means that state \( k \) is not among the states within the drawn trajectory.

Let \( P(e|S_C, l) \) be the probability that the number of critical states present in a drawn trajectory of length \( l \) is a fraction of the total number of critical states \( S_C \) equal to or greater than \( \epsilon \), for \( 0 < \epsilon < 1 \).

The number of critical states drawn from a trajectory of length \( l \) from environment \( e \) can be represented by the sum \( S_{k_1}^{e,l} + S_{k_2}^{e,l} + \ldots + S_{k_K}^{e,l} \) and the fraction of total states drawn is

\[
\frac{S_{k_1}^{e,l} + S_{k_2}^{e,l} + \ldots + S_{k_K}^{e,l}}{K}
\]

(20)

For a single environment \( e \),

\[
P(e|S_C, l) = P \left( \frac{S_{k_1}^{e,l} + S_{k_2}^{e,l} + \ldots + S_{k_K}^{e,l}}{K} \geq \epsilon \right)
\]

(21)

For single-scenario VIPER using a training that ends up incorporating \( m \) total trajectories, the probability of a fraction of the total critical states being \( \geq \epsilon \) is

\[
P_V(e|S_C, l, m) = P \left( \frac{1}{K} \sum_{k \in S_C} (\frac{S_{k,l}^{e,l}}{m}) \geq \epsilon \right) \text{ repeated } m \text{ times}
\]

(22)

\[
P_M(e|S_C, l, m, E) = P \left( \frac{1}{K} \sum_{k \in S_C} \left( \sum_{j=1}^{m} X_j \sim B \left( m, \left( \frac{S_{k,l}^{e,l}}{m} \right) \right) \right) \geq \epsilon \right)
\]

(23)

Note that \( P_V(e|S_C, l, m) = P_M(e|S_C, l, m, E) \) if \( E \) consists only of \( e_{n_E} \).

If any \( S_{k_i}^{e,l} < S_{k_i}^{e_{n_E},l} \) for any \( k \), for any \( i \neq n_E \) (where \( e_{n_E} \) indicates the final stage) then we can say that

\[
P_M(e|S_C, l, m, E) \geq P_V(e|S_C, l, m)
\]

(24)

Given this relationship between the probabilities, we can say that for any \( \epsilon \) where the assumptions noted above hold, the probability that MSVIPER utilizes this \( \epsilon \)-fraction of critical states in its training of the tree is greater than or equal to the probability that VIPER does so. Thus, under these conditions the upper bounds \( u \) for the divergence between expert and tree are equivalent between VIPER and MSVIPER, or MSVIPER has a smaller bound. (\( u_M \leq u_V \)) Thus, MSVIPER has equivalent or superior sample complexity, provided that any \( S_{k_i}^{e,l} < S_{k_i}^{e_{n_E},l} \) for any \( k \) and \( i \neq n_E \).