Tree Search Using a Hierarchical Actor-Critic

Bahaa Aldeeb
dept. of Robotics
University of Michigan
Ann Arbor, Michigan
baldeeb@umich.edu

***Abstract*—Problems that involve complex environments and require extended time execution are becoming increasingly common in robotics. Reinforcement learning has been an appealing tool for learning robot controls and environment constraints to solve such problems but it still struggles with long horizon planning. The work presents a new algorithm that merges RL and RRT methods to try and mitigate the long horizon issues of RL while leveraging it to learn the environment dynamics and constraints. The algorithm is able to successfully plan a path but could not achieve results that rival the current state of the art. Avenues for improving the current results are discussed at the end.**

***Index Terms*—Hierarchical Reinforcement Learning, Rapidly-Exploring Random Trees, Motion Planning, Machine learning.**

I. Introduction

With the increased integration of robotics in industry and society, the number of tasks those robots are faced with is increasing in difficulty. The traditional standard procedure of modeling every aspect of the environment that the robot is expected to interact with is becoming impractical or infeasible. An example of such problems are ones where designing extensive control for a robot/agent is very difficult, or ones where understanding the dynamics of the environment and modeling them is very difficult. Consider the problem of tying a knot around an object or sourcing a wire in a certain pattern around objects. Apart from it being difficult to model the environments for those tasks, it is very hard to use search methods to plan a path towards the intended goal is such problems due to the vastness of the state and goal spaces. A realm of research of increasing volume is starting to explore the use of learning methods to learn tedious and ultimately generalizable details of this problem.

Learning methods have already become central to computer vision and perception domains. Recently reinforcement learning (RL) has become the method of choice for learning controls and dynamics. RL has recently shown impressive results in the domain of robot learning such as teaching an arm to solve a Rubics cube to allowing an robot to automatically learn tool use. Even so, goal conditioned RL still faces difficulties when attempting to execute long horizon tasks or multi-stage tasks. Additionally, when working in safety critical environments or around other people, being able to present a plan that can be validated is an appealing trait that reactive methods do not typically posses. That is one reason why algorithmic planning is still the more common option.

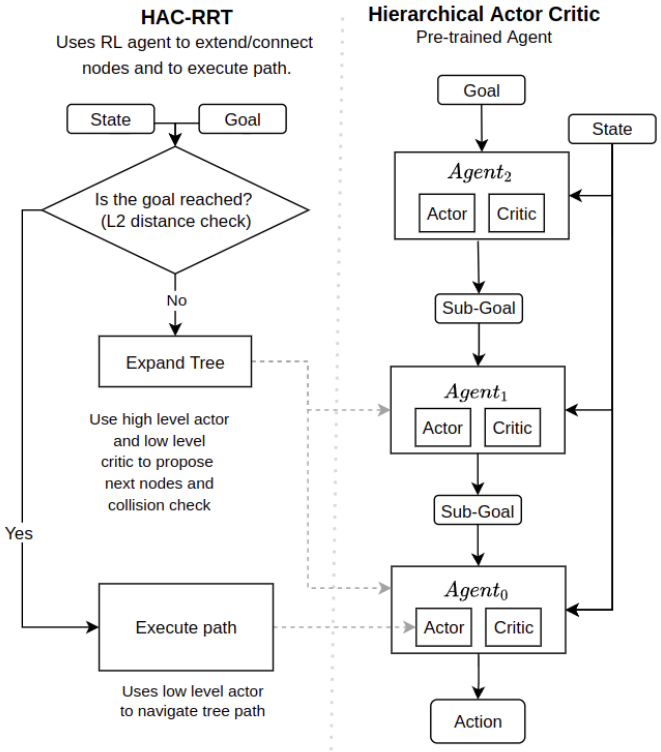

Fig. 1. A pre-trained Hierarchical Actor-Critic is used to implement an Rapidly-exploring Random Tree algorithm in an attempt to maximize the chance of deriving a successful path from the Reinforcement Learning Agent.

To benefit from the strengths of both learning and algorithmic methods, recent work has presented RL augmented planning algorithms. Those works were able to overcome the difficulty of dealing with non-holonomic complex agents and complex environments while being able to present long horizon plans. Recent work in this domain require that the environment is sampled and prompts the learned agent to determine if it is able to reach between environment samples. With the increase in the state space size and dimensionality, sufficiently sampling the environment becomes limiting. That observation is what gives RRT planning algorithms an edge over graph based ones. Additionally, in certain cases, a model of the environment is not available and so sampling the state or goal spaces becomes infeasible. This work presents a way of using a hierarchical RL agent to hallucinate a plan towards a

goal then execute it. The agent models state transitions as well as the agent controls and successfully presents plans given a starting state and a goal. To do so, a trained HAC [1] agent is used to execute an RRT [2] algorithm.

This novel way of using a hierarchical RL agent is shown to successfully produce plans but has not yet been able to match results produced by prior methods. Discussion on possible improvements and limitations are discussed at the end.

II. Related Work

A. Algorithmic Motion Planning

By delegating the task of perception and assuming a sufficiently extensive model of the environment is given, today's planning methods have evolved to deal with very complex cases. More recent renditions of the probabilistic road map [3] algorithm and rapidly-exploring random trees (RRT) [2] are capable of finding paths in complex environments with long horizon goals. RRTs proved effective at dealing with higher dimensional problems but their run-time complexity typically increases exponentially with the number of dimensions. Evolved versions of the RRT algorithm introduced asymptotically optimality [4] and the ability to handle non-holonomic motion [5] by leveraging heuristics and kinodynamic solvers.

One common requirement among algorithmic motion planning is the availability of a models of the environment and the agent as well as a measure of distance or reachability between different states in the environment. Some environments such as highly deformable objects or complex agents are very difficult to effectively model. Additionally, those algorithms assume the spaces are physically expressible, which is not the case when using images as state space. Machine learning has become a popular tool for dealing with those complexities. More recently, reinforcement learning methods have become an attractive tool in cases where interaction with the environment can be used to derive supervisory signals.

B. Reinforcement Learning

In the past few decades a large corpus of RL work has been dedicated to learning how to control complex agents and plan in un-modeled environments. Recent messages demonstrated how learning goal conditioned value functions, policies, or Q-functions (functions mapping a state goal and action to a value) [1], [6], [7] allows one to achieve a goal by reactivly taking the best action at a state towards a goal. Recent development in RL network and pipeline architectures [8]–[10] helped improve results and the robustness of training, making RL methods even more appealing. Hierarchical pipelines also proved helpful in training [11] and using [1] RL agents when dealing with temporally extended tasks.

Even with all the impressive results that RL agents have demonstrated, research towards improving their training robustness and ability to handle long horizon tasks is still highly active.

C. Planning-Augmented Reinforcement Learning

To tackle long-horizon issues that RL faces, many propose methods that merge RL and planning. In doing so, such work leverages the long horizon capabilities of algorithmic planning with the model learning abilities of RL agents. Most such methods sample the state space and use the RL agent to generate a PRM [12]–[16]. Some methods mixed RRT style planning with learned distance metrics and local policy [17].

This work is most similar to RRT algorithms [2], [17] and Hierarchical Actor-Critic (HAC) architectures [1], [14]. The intention is to avoid sampling the state space and instead capitalize on the use of the RL agent's planning intuition, while allowing for some randomness to avoid local minima. One unique observation is that training an HAC architecture yields the models necessary to transform the reactive RL agent into a rapidly exploring planning agent.

III. Problem Statement

Consider a situation in which we would like to fold a piece of cloth into some shape, or bend a rope in a certain way. Such a situation requires commanding an agent H to manipulate the environment, through an intricate and lengthy sequence states P towards a goal g. The plan in this case is assumed to be of long horizon.

$$
\begin{aligned}
\text{find } & P = [s_0, s_1, \ldots, s_n] && s_i \in S: \text{ state space} \\
\text{such that } & g = f_{S \to G}(s_n) && g \in G: \text{ goal space} \\
& a_{i-1} = H(s_{i-1}, g) && a_{i-1} \in A: \text{ action space} \\
& s_i = env(a_{i-1}) \\
& d(s_0, s_n) >>> 0
\end{aligned}
$$

Where $f_{S \to G}(s_n)$ is a mapping from state space to goal space and env expresses the environment's state transition as a function of an action being taken. The function d measures distance between states or goals while considering the environment constraints and agent dynamics.

The desired path P is not expected to be optimal but is expected to me one that is more likely to be successfully executable. One assumption is that the environment is complex and very tough to model, thus a designed representation of it is not available. Interacting with such an environment is considered possible so data driven methods can be used to learn it's dynamics and constraints.

IV. Background

This work is focused on the use of a Hierarchical Actor-Critic (HAC) agent. Such agents are better equipped at learning long horizon tasks. The aim is to leverage the agent in an RRT algorithm which require a model of state and goal transitions as a function of actions, a model for measuring distances between states, and model for selecting the actions needed to transition from one state to another.

A. Actor Critic Agents

An actor-critic agent is one that utilizes both a Q-function and a policy to perform off-line learning.

$$\pi \colon S \times G^* \to A^*$$
$$Q \colon S \times G^* \times A^* \to \mathbb{R}$$

The appeal of this architecture is its ability to train off-policy and learn both a distance-like metric that is the Q-function, as well as a policy. Off-policy learning is done by maintaining a replay-buffer which is a set of tuples describing experiences by tracking (state s, action a, goal g, reached state s', reward r, weight decay γ). Experience using any policies (random walks or human demonstrations) can be stored in the replay buffer and used to supervise the Q-function. Other policies are typically used to prime replay buffer then as the agent starts interacting with the environment its experiences replace the previously stored ones. If one supervises the Q-function with a penalty of -1 for every step away from the goal the current state is, the function learns to indicate the negative distance to the goal. Another benefit of having a replay-buffer is the ability to use goal-relabeling. For example, If an action a at state s resulted in a next state that satisfies a goal g' but not g, then the Q-function can be motivated to produce a favorable value (typically zero) when given (s, a, g').

Since the Q-function is a form of distance measure, the policy can be trained to minimize its value. The objective of the policy then is to minimize the distance between the current state and goal by proposing the optimal action.

$$\pi(s, g) = \text{argmax}_{a \in A}\ Q(s, a, g)$$

Learning a policy becomes especially important when dealing with continuous state spaces due to the difficulty of directly optimizing the Q-function over A.

B. Hierarchical Actor Critic

An HAC is one type of hierarchical RL agents presented in [14] that utilizes a type of Actor-Critic agents and leverages off-policy learning and goal-relabeling to break down the task and facilitate long horizon learning. An HAC is constructed of k layers l_i each of which is an actor critic. The layers are sequentially connected such that a layer before would propose a sub-goal $v \in V$ for its lower neighbor.

$$\pi_0 \colon S \times G \to V$$
$$\pi_{1 \to k-2} \colon S \times V \to V$$
$$\pi_1 \colon S \times V \to A$$

$$\pi_{HAC} \colon S \times G \to A$$

The lowest layer l_0 is tasked with learning the environment dynamics by mapping a state $s \in S$ and a sub-goal $v = f(g \in G)$ to an action $a \in A$ that can propagate s towards g. All other agents learn to map $(s, f(g))$ into a sub-goal $v \in V$. The HAC is run by calling onto the top layer to run at most N iterations to achieve the goal. Every layer will consequently call its downstream layer to do the same. Layer 0 finally produces an action and prompts the agent to move at every one of its iterations.

To train this agent, a replay buffer is maintained for every layer. Each layer assumes that layers below it are optimal and thus only tracks details directly pertaining to it. Data augmentations dubbed "hindsight" augmentations are designed to maximize the information gained from experience at every layer. Layers are then trained in the same way described in section IV-A.

The training process motivates different layers to learn different abstractions of the task. Intermediate layers will tend to produce a sub-goals that are reachable by their downstream neighbors. Since higher layer run exponentially more environment steps, those layers afford to reach farther sub-goals.

$$a_i = \pi_i(s, g) \quad \& \quad a_{i-1} = \pi_{i-1}(s, g) \tag{1}$$
$$d(f_{S \to G}(env(a_i)), g) < d(f_{S \to G}(env(a_{i-1})), g) \tag{2}$$

Another property worth noting is that the Q-functions of all layers in the HAC will produce a dynamics-aware distance at a different level of abstraction. Higher layers can move exponentially more steps so when given a reachable goal higher Q-functions ought to indicate a distance to that goal of exponentially smaller magnitude. Figure 3 depicts that effect.

$$s' = env(a_i) \tag{3}$$
$$q_i(a_i, s, g) \approx -d(f_{S \to G}(s'), g) \tag{4}$$

V. Method

In prior work, it was observed that the Q-functions were better able to evaluate reach-ability between nearby states [13], [14]. Those methods used Probabilistic Road Maps for planning, which is harder to construct and search over when dealing with high dimensional spaces. To avoid the limitations associated with constructing and using a PRM, this project introduces a way to utilize a trained HAC agent in order to implement an RRT search algorithm. To implement an RRT a state transition model has to be derived, a means of measuring distance is needed, as well as a method for performing collision checking.

A. Deriving Search Space Samples Using HAC

One insight of this work is that, by carefully selecting the sub-goals space V, one can acquire a state transition model as a side product of the hierarchical agent training process. To derive such a model we need to design our spaces such that we are able to map between state, goal, and sub-goal spaces. The assumption is that there exists easy to acquire functions $f_{S \to G}, f_{V \to G}, f_{G \to V}, f_{G \to S}$ that map samples between spaces. When working with high dimensional agents, it is likely that the desired goal can be mapped to a distribution of states, which can be overcome by sampling a satisfactory elements from that distribution. To handle implicit goal representations [14] one could derive a satisfactory mapping between state and goal by optimizing over Q_0.

With those mappings available, consider the function of the policies $k-1$ through $j>0$.

$$\text{Let } \Phi_j(s,g) = \pi_j(s, \pi_{j+1}(s, ...\pi_{k-1}(s,g)...)) \quad (5)$$

$$s' = f_{V\to S}(\Phi_j(s,g)) \quad (6)$$

$$\text{Then } d\left(f_{S\to G}(s'), g\right) < d(f_{S\to G}(s), g) \quad (7)$$

$$\text{or } f_{S\to G}(s') = f_{S\to G}(s) \quad (8)$$

Using Φ and the function mapping $G \to S$ we can sample the state space in an informed fashion, towards a goal.

The work in [14] touted the ability of Q_0 to better evaluate close proximity samples. To ensure the short proximity between the consecutive samples produced by Φ_1 and presented to layer l_0, further assumption are considered. By limiting the number of steps allowed by each layer to some low number while training the HAC agent, one can assume that consecutive samples produced by Φ_1 can be reached roughly through linear progression in state space. That is because the layers are motivated to propose sub-goals reachable by their lower neighbor. If layer 0 can only take 5 steps, layer 1 will propose a closer sub-goal. With that, interpolating between the consecutive samples of Φ_1 should often produce valid sub-goal. Instead of seeking a function Φ_1 we seek to derive a $\hat{\Phi}_1$ which produces a sub-goal interpolated between the current state $f_{S\to G}(s)$ and $f_{G\to S}(\Phi_1(s,g))$, and is closer to the former. The distance of interpolation is a hyper-parameter of this algorithm.

$$g_1' = f_{V\to G}(\Phi(s,g))$$
$$g_2' = f_{V\to G}(\hat{\Phi}(s,g))$$
$$dist(g_1', g) < dist(g_2', g) < dist(s, g)$$

B. Collision Checking Using Q-functions

After acquiring consecutive states, layer 0 is prompted to determine if it is able to reach from one to the other. In a way similar to how [14] and [13] connect samples in the replay buffer, a threshold on Q_0 is used to determine whether the agent ought to attempt to progress to a given sub-goal or not. This thresholding of Q-values is akin to collision checking in an RRT. Using samples generated by $\hat{\Phi}_1$, Q_0 as a distance measure, and the reachability assessment for collision checking, an RRT like algorithm can be constructed.

C. Tree Search via HAC

To instantiate the algorithm a goal g is given and a projection of the initial state $f_{S\to G}(s_0)$ is added as a first node in the search tree T. To allow exploration and avoid dead-ends, a randomness factor is introduced by having every layer elect, with some probability, to pursue a random goal at every iteration. The result is Algorithm 1 which promotes exploration while planning and presents the added benefits of allowing a hierarchical agent's layer to communicate back and forth before pursuing a target.

Algorithm 1 HAC-RRT
Input: Initial State s, Goal g
, Max Training Steps For Layer N

```
T = Tree
T.insert(f_{S→G}(s))
while (within time limit) do
    Expand(k − 1, g, T, N)
    if (g ∈ T) then
        return Path
    end if
end while
return None
```

Algorithm 2 Expand
Input: Layer Index "i", Goal "g", Tree "T",
Max Training Steps For Layer "N"

```
for N steps or until timeout do
    g* = SelectGoal(g, p_goal bias)
    s = SelectNearestStateInTree(T, g*)
    v = ProposeNextSubgoal(s, g*)
    if (−Q_{i−1}(s, v) < threshold_{i−1}) then
        if (i > 1) then
            Expand(i − 1, v, s)
        else if (i = 1) then
            AddToTree(f_{V→G}(v))
        end if
    end if
end for
```

The agent attempts to extend the tree using Algorithm 2. At every iteration, a layer determines the closest node to extend from by using its Q-function as measure. That process is described in Algorithm 3. After selecting the nearest node, the state at that node is derived, then the policy is asked to propose the desired sub-goal given that state and the pursued goal. The downstream layer's Q-function is then prompted to determine the reachability of the proposed sub-goal. If it is deemed reachable, that sub-goal is passed down for the lower layer as a goal to step towards. This process goes on until layer 1 is reached. Layer 1 proposes an interpolated (short proximity) sub-goal which is added to the tree if layer 0 finds it reachable. Unlike previous collision checking thresholds, layer 0 has a threshold scaled in proportion to extent of interpolation; the closer the interpolated point is from the node the lower the threshold of layer 0's collision checking is.

Every layer in this process is given N steps to reach their delegated goal. The algorithm runs until the goal is added to the tree or the maximum execution time is reached.

Algorithm 3 Select Nearest State In Tree
Input: Goal g, Tree T, Distance Measure Q
Output: Tree is not empty

for (g_{node} in T) **do**
  $d = -Q(f_{G\to S}(g_{node}, g)$
  if ($d < d_{min}$) **then**
    $g_{nearest} = g_{node}$
    $d_{min} = d$
  end if
end for
return $f_{G\to S}(g_{nearest})$

Algorithm 4 Propose Next Subgoal
Input: State s, Goal g

$v = \pi_i(s, g^*)$
if ($i == 1$) **then**
  $v = \text{Interpolate}(f_{S\to V}(s), v)$
end if
return v

VI. Results

Due to this being a semester-long course project, this section will discuss most explored routes, successful and abandoned. Results shown below serve as a progress report on the current state of the this work.

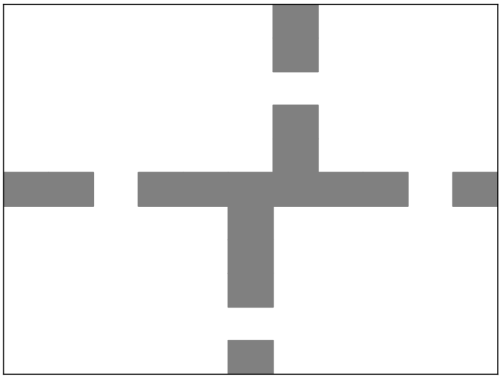

Fig. 2. Maze used for all the project experiments.

A. Training An HAC In Maze

The first step towards using an HAC RRT algorithm is training a hierarchical actor-critic agent. Although previous works have shown success in training complex agents [1], [14], I found it non-trivial to do so in this project.

When training the HAC, the same hindsight ideas and network models that were presented by [1] were used. Initial attempts of training an HAC utilized a PyTorch implementation of the agent [18] and a 10x10 version of the FourRooms maze environments used by [13] and shown in Figure 2. Using a 2D space and a point agent facilitate the process of validating the agent and the RRT augmentation.

To evaluate the success of the agent's training, Q-maps were derived. Q-values in a trained HAC agent can be treated as a dynamics-aware negative distance measure. We expect the Q-values to be higher in magnitude when the path between the given state and goal is obstructed. A Q-map is a 2D plot of distances at every state given a goal. To generate that plot, a random goal along with a large number of states are sampled, then the Q-value of every state-goal pair is evaluated. Since $G \in \mathtt{R}^2$, we can discretize it to generate a grid. Then $f_{S\to G}(s_i)$ are used as indices and the Q-value associated with the state sample s_i is assigned to that index of the grid. As evident from Q-maps visualized in Figure 3, the agent could not learn sufficient details of the maze layout even after training for $1e6$ epochs. Without reducing the maze to a 10x10 (small) maze, the agent failed to learn sufficient Q-functions.

When attempting to train on a larger or more complex maze, the agent's Q-maps seemed to collapse. Collapsed Q-maps indicate that the agent had no understanding of the environment constraints or of the agent's dynamics. The reason behind slow and failed training attempts might be due to poor tuning. After further reading it became evident that using a DDPG (type of Actor-Critic agents) with an ensemble of Q-functions is one method that could improve training. Using distributional Q-functions also seemed to improve training. Other architectures such as a Soft-Actor-Critic [10], could also be more effective building blocks to construct the HAC layers.

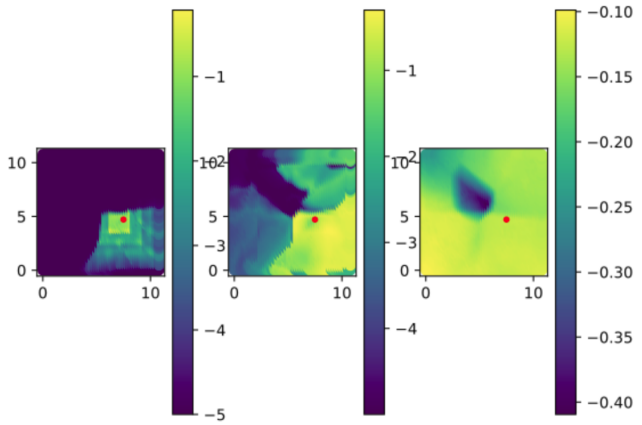

Fig. 3. Q-maps of a 3 layer HAC agent given the red dot as a goal. The left most image is that of Q_0 of layer 0, the middle is Q_1, and to the right is the map of Q_2. Brighter areas indicate where the agent thinks is reachable in fewer steps. As expected, higher agents are able able to cover more space in less steps because they run multiple iterations of the agent below them and thus exponentially many steps.

B. Using Original HAC Environment

To progress towards evaluating the core contribution of this work (the RRT augmentation) we reverted to the original HAC model and environment published by [1]. That agent was implemented in Tensorflow, so the work had to be accordingly adapted.

For this experiment the Open-AI Gym's 4-legged ant agent was used, each leg having 2 joints that offer 3 constrained degrees-of-freedom. Along with the pose of the agent the state

space compounds to an 18 dimensional vector. The portion of this vector that represents the 3D location of the agent can be thought of as the projection of the state onto goal-space G.

The HAC is designed to propose 6 dimensional sub-goals representing the target position of the agent along with the velocity of its center of mass. The goals given to the agent are simply the 3 dimensional position of its center. As shown in Figure 4 the HAC-RRT was able to plan paths executable by the agent.

Although the platform managed to train a more complex quadrupedal ant-like agent, it seemed like some shortcuts were taken. For example, the start and goal were sampled from a set of 4 points in the 3D space.

Even with the successful generation of a plan, we observe that the tree generated is limited in its spread. It is possible that the limitations imposed on the start and goal by the used environment caused the agent to over-fit to generating sub-goals along paths it deemed most viable. It is possible that constraints as such facilitated the training of the layers concerned with long horizon planning. Those hypotheses suggest it would be beneficial to re-evaluate the agent on a more general environment. Even so, an evaluation was run to compare the success rate of the vanilla agent to the RRT augmented one.

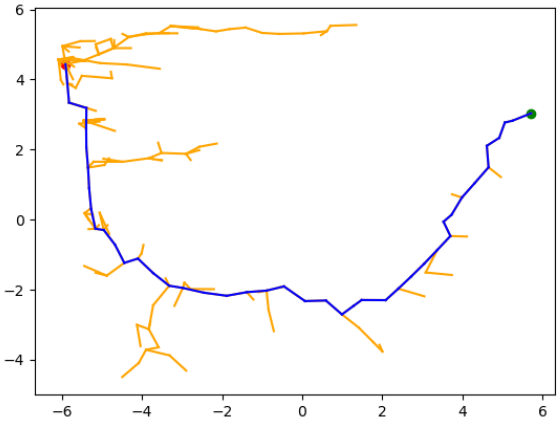

Fig. 4. An example of a path planned between the start (red) point and the goal (green) point.

C. Agent Comparison

To evaluate the effect of the RRT augmentation on the agent's ability to navigate, the HAC is run with and without augmentation for 50 episodes. At each episode a start and goal are sampled, a plan is derived when the RRT is used, then the agent is asked to reach the goal. We observe that in its current state the RRT augmentation leads to a decrease in accuracy. While the vanilla HAC is successful 90% of the time, the RRT augmented HAC is only successful 50% of the time. The significance in the different of success rates might suggest implementation issues in the RRT algorithm or issues caused by the HAC training.

It is worth noting that the RRT augmentation requires significant planning time most of which is spent on finding the closest node to extend from. The distance calculation in the augmented agent requires running π and Q and doing so multiple times at every layer given a goal.

VII. Discussion

A. Importance of Optimality

After generating a valid path, in certain runs the agent would flip while attempting to follow the path. The lack of smoothness presented by the randomness of the algorithm would sometime lead to states that the agent was not used to dealing with. For example, if a random point with higher elevation proceeded a point with lower elevation, while the ant was walking at a significant speed, it would cause the agent to tilt and flip while directing the speed upwards. This might be an issue caused by the disconnect between training towards reactive operation, and attempting to strictly execute a noisy plan.

Another issue observed concerning optimality was that in certain cases, a random goal causes the algorithm to find a path that is much lengthier than the optimal path. That lead to more failures simply because the agent had more time to fail.

B. Search Space Limitations

A core concept of this method is the use of a portion of the agent as a state transition function. That assumption is not too strong until we consider using image embeddings as a goal or subgoal space. With spaces that vast it becomes much harder to propose transitions of the embedding that result in a recognizable and valid portion of the space. It would be interesting to see how well the hierarchy would be able to learn such a transition function.

VIII. Conclusion

This work presents a novel method that transforms a Hierarchical Actor-Critic agent into a planning agent by integrating it into an RRT like algorithm. This method avoids the limitations associated with directly sampling the environment and building a search graph and allows communication between the hierarchical agent's layers. Although current results are not very promising, there are still a few imperfect details that might improve the outcome. In conclusion the project leaves plenty of open questions worth exploring: Is it still more effective to use sparse PRM graphs or SLAM-augmented exploratory RL agents? Could an HAC yield a state transition function in image space that is worth considering? Are there interesting problems where mapping between state and goal space significantly difficult?

References

[1] A. Levy, R. Platt, and K. Saenko, "Hierarchical actor-critic," *arXiv preprint arXiv:1712.00948*, vol. 12, 2017.

[2] S. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[3] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[4] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robotics Science and Systems VI*, vol. 104, no. 2, 2010.

[5] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[6] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *International conference on machine learning*. PMLR, 2015, pp. 1312–1320.

[7] L. P. Kaelbling, "Learning to achieve goals," in *IJCAI*, vol. 2. Citeseer, 1993, pp. 1094–8.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[9] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[10] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[11] E. Chane-Sane, C. Schmid, and I. Laptev, "Goal-conditioned reinforcement learning with imagined subgoals," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1430–1440.

[12] A. Faust, K. Oslund, O. Ramirez, A. Francis, L. Tapia, M. Fiser, and J. Davidson, "Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 5113–5120.

[13] B. Eysenbach, R. R. Salakhutdinov, and S. Levine, "Search on the replay buffer: Bridging planning and reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[14] R. Gieselmann and F. T. Pokorny, "Planning-augmented hierarchical reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5097–5104, 2021.

[15] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel, "Value iteration networks," *Advances in neural information processing systems*, vol. 29, 2016.

[16] L. Lee, E. Parisotto, D. S. Chaplot, E. Xing, and R. Salakhutdinov, "Gated path planning networks," in *International Conference on Machine Learning*. PMLR, 2018, pp. 2947–2955.

[17] H.-T. L. Chiang, J. Hsu, M. Fiser, L. Tapia, and A. Faust, "Rl-rrt: Kinodynamic motion planning via learning reachability estimators from rl policies," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4298–4305, 2019.

[18] N. Barhate, "Nikhilbarhate99/hierarchical-actor-critic-hac-pytorch: Pytorch implementation of hierarchical actor critic (hac) for openai gym environments," 2022. [Online]. Available: https://github.com/nikhilbarhate99/Hierarchical-Actor-Critic-HAC-PyTorch