Learning Sparse Rewards with Rapidly Exploring Random Trees

Ryan Lee*, David Martinez*, Tyler McCown*, Peter Racioppo*
Department of Mechanical and Aerospace Engineering
University of California, Los Angeles
580 Portola Plaza, Los Angeles, CA, United States
ryanlee126@ucla.edu, dmrm2312@ucla.edu, tylermccown@engineering.ucla.edu, pcracioppo@ucla.edu

**Equal contribution.*

***Abstract*—Deep reinforcement learning has been successfully applied to a variety of complex decision-making tasks and perfect-information games, but a number of major technical challenges remain. Reinforcement learning algorithms suffer from inefficient sampling, typically require a vast amount of training data, and often generalize poorly. In cases where rewards are very sparse, or can only be obtained after a long sequence of actions, greedy exploration techniques may fail to converge or may converge on sub-optimal behavior. In practice, complex reward functions often need to be specified through trial and error during training. This paper explores the application of rapidly-exploring random trees to the Q-Learning problem to bias exploration toward unexplored regions of the state space. Intermediate rewards are placed randomly in the state space to combat the difficulties posed by sparse rewards and bottlenecks. Q-Learning is used to grow a tree between these intermediate rewards. Once the goal is found, Q-Learning is used to sequentially backup along the tree and obtain a global solution to the two-point boundary value problem. To validate this algorithm, we show simulations on maze grid worlds with stochastic transition dynamics. For sufficiently large mazes, the algorithm is shown to consistently converge in fewer iterations than unassisted Q-Learning.**

***Index Terms*—Reinforcement learning, Q-Learning, Rapidly-exploring random trees, dynamic programming, motion planning**

I. Introduction

Reinforcement learning (RL) is a branch of machine learning dealing with problems in which a sequence of actions must be taken in a stochastic or uncertain environment. [1] Model-free RL algorithms can be applied to a wider class of problems than graph search methods because they enable the agent to learn properties of the system such as dynamics and rewards.

Q-Learning, introduced by Chris Watkins in 1989 [2], solves the reinforcement learning problem by building a Q-table, which characterizes the value of state-action pairs, using dynamic programming. For problems with large or continuous state spaces, the values of Q may be approximated by taking weighted sums of basis functions, or with the use of deep neural networks as function approximators. This method, popularized as "deep reinforcement-learning" by Google's DeepMind laboratory in 2014 [3], has been successfully applied to a wide array of problems, including perfect-information games such as chess [4], [5] and Go [6], and, more recently, imperfect-information games such as Poker [7], [8], and Starcraft II [9].

During training, reinforcement learning agents must trade off the need to explore new regions of the state-action space and to exploit the state-actions that have so-far been found to be most valuable. This can be accomplished in Q-Learning via an ϵ-greedy strategy, in which the agent takes a random action with probability ϵ at any given state. Otherwise, it follows the optimal policy dictated by its best estimate for Q. The ϵ-greedy approach suffers from its use of a Q-table which may be very far from the optimum. More fundamentally, this approach does not represent the temporal dependence between state-actions and therefore does not efficiently explore the state space. In other words, an agent should learn to systematically explore new regions of the state space rather than randomly walk, which will result in sampling nearby states many times [1].

Most RL approaches, including ϵ-greedy strategies, have difficulty in situations in which rewards are sparse — that is, in which rewards can only be obtained after a long series of actions [10]. The Atari game Montezuma's Revenge, in which a player must find their way through a long series of pathways, collecting keys and opening doors in sequence, is a classic example of this type of sparse reward problem [11].

This is closely related to the issue of bottlenecks: situations in which the agent must pass through a small subset of the state space. Random exploration strategies such as ϵ-greedy strategies require many iterations to traverse bottlenecks, which can cause reinforcement learning to fail to converge to an optimal policy.

One method used to help RL solve problems with sparse rewards is to reward an "exploration bonus" for visiting new states, or to penalize visiting the same states multiple times [1]. These methods become less effective as the dimension of the state space increases, because the number of times each state is visited decreases. To combat this problem, hash functions have been used to track visitation to general areas in the state space [12]. Other papers have approximated these counts using pseudo-counts [13]. Recently, a method using differences between neural networks to generate the exploration bonus has shown promising results for solving the classic Atari Game Montezuma's revenge [11].

In practice, performing a complex task is often accomplished by designing a "shaped reward" function that biases the agent toward certain actions. For example, to move a robotic system to a target position, we might give a continuous reward for minimizing the L_2 distance to the goal. Because reward shaping tends to bias the agent, this approach can lead to globally sub-optimal behavior.

For systems with known dynamics and large state spaces, graph-based planning methods are commonly used. These algorithms build graphs of states sampled from the free work space, which are connected by viable trajectories. Once a graph is generated, path planning from point to point can be quickly achieved using graph search algorithms. These sample-based algorithms are especially useful for large and high-dimensional work spaces because the computation can be scaled independently of the size or dimensionality of the state space [14], [15].

One commonly used graph-based method is the Probabilistic Road Map (PRM). PRMs are constructed by randomly placing nodes at points in the free sections of the configuration space and then using a local planner to create edges that connect the node to other nodes already in the graph. PRMs work best with deterministic dynamics and require a means of checking ahead of time if a candidate node is in a reachable subset of the state space [14].

In contrast, Rapidly-exploring Random Tree (RRT) algorithms branch out from a single starting state. At each step, RRTs grow by sampling a point from the configuration space, usually within a neighborhood of the tree, and then adding a directed edge from the closest node in the tree to the randomly sampled point. The RRT graph is always connected and tends to grow toward unexplored regions of the state space [15]. RRTs have been shown to be efficient for quickly searching high-dimensional spaces with algebraic and differential constraints [16].

Recent research by Faust et al. describes a method for performing long-term path planning for a robot affected by noisy dynamics [17]. The authors first train a RL agent to locally avoid obstacles and steer toward a point with a known range and bearing. This method is used to build a long-range graph-based path planner. Here, local reinforcement learning steps act as steering functions to determine the viability of edges [17], or as biasing factors for an RRT [18].

II. Background

In the reinforcement-learning paradigm, a dynamical system is represented as a Markov decision process, a four-tuple (S, A, P_a, R_a), where S, the state space, is the set of all possible states; A, the action space, is the set of all possible actions; $P_a(s, s')$ is the probability of a transition from state s to state s' after taking an action a; and $R_a(s, s')$ is a reward for transitioning from state s to state s' having taken action a [19].

The goal of the reinforcement learning problem is to find a policy $\pi(s)$ that prescribes an action for every possible state in order to maximize the expected sum of discounted future rewards, given by

$$\underset{\pi}{E} \sum_{t=1}^{H} \gamma^t R_a(s_t, s_{t+1}),$$

where H is a time horizon and γ is a discount factor that encodes a preference for obtaining rewards in fewer time steps. Tabular Q-Learning methods address this question by assigning a value to all state-action pairs:

$$Q(s,a) = \underset{\pi}{E}\left[R_a(s,s') | s, a, \pi\right]$$

The basis of Q-Learning is the application of a dynamic programming update:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\big(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\big)$$

where $\alpha \in (0,1)$ is a "learning rate" and $r_t = R_a(s_t, s_{t+1})$ is the reward obtained at time t [19]. The full Q-Learning algorithm is described in greater detail in Algorithm 4 in the appendix.

Rapidly-exploring random trees (RRTs) are a class of algorithms for building space-filling trees in high-dimensional state spaces under differential constraints. RRTs, which were introduced by Stephen LaValle in 1998 [20], have been shown to work well on a wide range of motion planning problems with kino-dynamic constraints that commonly arise in robotics. Kuffner and LaSalle studied the space-filling properties of RRTs, which tend to expand into the largest Voronoi regions of a graph [21]. RRTs are also guaranteed to be probabilistically complete, and the variant RRT* algorithm also achieves asymptotic optimality [22]. A more detailed description of the RRT algorithm can be found in Algorithm 5 in the appendix.

III. Problem Statement

In this paper, we demonstrate our algorithm on 2D bounded $n \times n$ grid worlds. Let $S = \{s_1, s_2, \ldots, s_{n^2}\}$ be the set of all n^2 possible states and let $A = \{a_1, a_2, a_3, a_4, a_5\} = \{\text{Up}, \text{Right}, \text{Down}, \text{Left}, \text{Stay}\}$ be the set of five possible actions. Define a deterministic transition model T that outputs a state s' for every state-action pair: $s' = T(s,a)$. Define a Bernoulli probability distribution of transitioning from state s to s' under action a:

$$q(s,a,s') = \begin{cases} c & \text{if } s' = T(s,a) \\ 1-4c & \text{o.w.} \end{cases}$$

where $0 < c < 1$. We define a set of "obstacle" states S_O to which the agent cannot transition, and we define a second probability distribution $p(s,a,s')$, where $p(s,a,s') = 0$ if $s' \in S_O$ and the chance of staying in state s is incremented by the number of adjacenct obstacle states. That is,

$$p(s,a,s') = \begin{cases} 0 & \text{if } s \in S_O \\ q(s,a,s') + kc + m(1-4c) & \text{if } a = a_5 \\ q(s,a,s') & \text{o.w.} \end{cases}$$

where $m = |T(s,a) \cap S_O|$ is the number of "desired actions" for which s' is an obstacle, and $k = |T(s,a') \cap S_O|$ s.t. $a' \in A \backslash a$ is the number of "non-desired actions" for which s' is an obstacle.

The system receives a reward r for reaching the goal state s_g from any state-action pair:

$$R_a(s,s') = \begin{cases} r & \text{if } s' = s_g \\ 0 & \text{o.w.} \end{cases}$$

IV. Methods

The proposed algorithm combines the strengths of Q-Learning and RRT in an algorithm which we call Exploring Sequential Q-Learning (ESeQ). This method decreases the amount of training needed to find solutions to model-free problems with sparse rewards and fixed starting states. The ESeQ algorithm is split into two halves: Q-RRT, which attempts to replicate the exploration biases of RRTs by placing local rewards and using Q-Learning to add edges to an RRT, and SeQ, which uses the paths identified by the RRT to solve the full Q-Learning problem. ESeQ is designed to solve problems which involve model-free systems with sparse rewards placed at unknown locations and fixed starting locations.

The Q-RRT algorithm involves placing temporary intermediate rewards in order to encourage exploration throughout the state space. These intermediate reward locations are selected similarly to the nodes for a RRT. The agent begins in a given starting location in the state space, forming the single node in a graph with no edges. At each iteration, a new state is sampled at random and assigned a small reward. Q-Learning is then used to search for a policy which navigates from the starting state to this new point. If the learning converges and a trajectory can be found, then the new state is added as a node in the graph and the trajectory between the initial state and the new state is added as an edge. The intermediate reward is also removed. At the next iteration, a new node is again randomly sampled and assigned a small reward, and Q-Learning is used to search for a policy which navigates from the closest existing point in the graph to the new point. If learning converges, the new node and edge are added to the graph, and this process is repeated until the true goal state is found. These steps are summarized in Algorithm 1.

Solving these smaller Q-Learning problems guarantees a valid path of nodes from the starting state to the goal, when it is found. It is important that the intermediate rewards be much smaller than the original goal reward so that, if the system finds the original goal, it will ignore the intermediate reward and create a policy to obtain the original goal. This process is not exactly RRT, since RRT does not require the system to converge on the proposed node, but instead requires that we solve a two-point boundary value problem for finding trajectories with Q-Learning. The result is a tree which includes the starting state and goal state as nodes. It is then possible to find a path Ω connecting these two nodes along the graph.

Algorithm 1: RRT Q-Learning

```
Data: state-space $S$, start state $s_0 \in S$, auxiliary
      reward $r$, horizon $H$, discount factor $\gamma$, learning
      rate $\epsilon$, tree growth rate $\lambda$, Q-learning
      convergence parameter tol
Result: a tree $T = \{V, E\}$ rooted at $s_0$
Init: $V \leftarrow \{s_0\}$
$E \leftarrow \emptyset$
$s_{new} \leftarrow s_0$
while $|Q| < tol$ do
    $Q \leftarrow$ Q-learning($s_{new}$)
    $s_{new} \leftarrow$ random($S$)
    $d \leftarrow$ dist($V, s_{new}$)
    $s_{nearest} \leftarrow$ argmin($d$)
    if $d > \lambda$ then
        $s_{new} \leftarrow \lambda \times$ heading($s_{nearest}, s_{new}$)
    end
    $Q_s \leftarrow$ Local Q-learning($s, s_{new}, r$)
    $e_{new} \leftarrow \{s_{nearest}, s_{new}, Q_s\}$
    $V \leftarrow \{V, s_{new}\}$
    $E \leftarrow \{E, e_{new}\}$
end
```

Algorithm 2: Sequential Q-Learning

```
Data: a path $\Omega = \{v_0, \ldots, v_n\}$ from $s_0$ to $s_f$
Result: $Q$
Init: $i \leftarrow 1$, $Q_1 \leftarrow$ random;
while $i \leq n$ do
    $Q_{i+1} \leftarrow$ Q-learning from $v_{n-i}$ to $s_f$, initialized
      with $Q_i$;
    $i \leftarrow i + 1$;
end
```

Because Q-Learning has converged along every edge of this path, the nodes in Ω can be used as a set of initial conditions for a global Q-Learning sweep. To find the overall policy, Q-Learning is run starting from the last node in the tree until it finds the end goal. Next, another Q-Learning step is run from the previous node in Ω until it finds the end goal, but with the Q table initialized to the final result from the previous node. The process is repeated to "back up" sequentially along the tree. The steps of this approach, which we term Sequential-Q (SeQ), are summarized in Algorithm 2. The combination of Q-RRT followed by SeQ is referred to as Exploring Sequential Q-Learning (ESeQ), and the process is organized in Fig. 1.

V. Results and Discussion

The algorithm was tested against standard Q-Learning with tuned hyperparameters. These tests were performed in two different types of environments at a range of grid world sizes. For each environment, the total number of iterations in the Q-learning steps before convergence was recorded. All hyperparameters for these tests are recorded in Table III in the appendix.

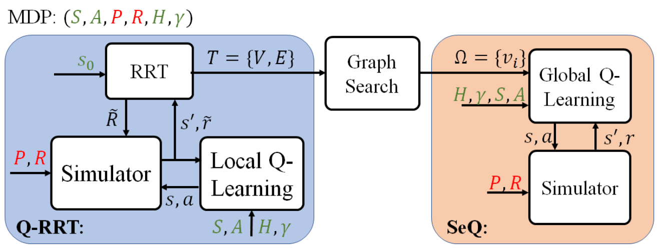

Figure 1: A block diagram for the ESeQ algorithm. The RRT-guided Q algorithm is displayed in the blue box on the left (Q-RRT) and the Sequential Q (SeQ) algorithm for finding global approximations of the Q values is in the yellow box on the right.

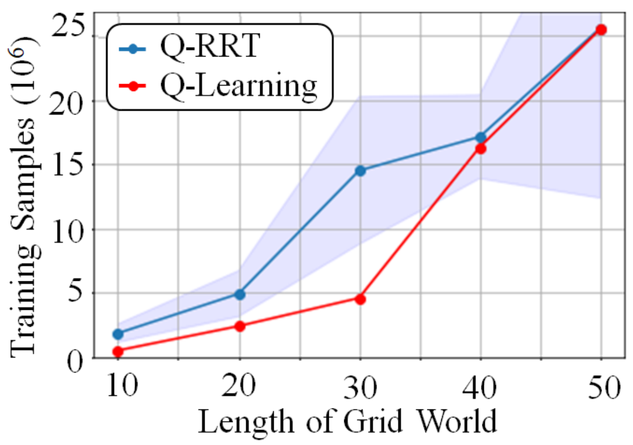

Figure 2: Number of training steps required for both Q-RRT and a standard ϵ-greedy Q-Learning algorithm to converge, for square grid worlds of dimension between 10 and 50.

First, both algorithms were tested in obstacle free environments of varying sizes to gauge the growth rates of each algorithm unhindered by obstacles. In these tests, the agent begins in the corner opposite the global goal. The hyper-parameters of the unassisted Q-Learning algorithm were optimized by trying 27 different combinations for each grid size. Due to limitations in system memory, grid worlds were restricted to sizes below 70×70.

Similarly, when using Q-RRT, hardware was the main limitation. As we did not write our script to be GPU accelerated, run-times exceeded 1 hour for grids of dimension 50×50 and above. Hyperparameters for this algorithm were tuned using a grid search, as for the unassisted Q-Learning.

In addition to the experiments in the obstacle-free state-space, the algorithms were tested in mazes of varying sizes composed of a series of rooms connected by narrow bottlenecks, like the one shown in Fig. 3. The maze experiments are intended to test the performance of the algorithm when the path to the goal is restricted by bottlenecks.

Fig. 2 compares the results of the Q-RRT algorithm to standard Q-Learning for a range of environment sizes. For small grid worlds, the standard Q-Learning algorithm out performs Q-RRT by nearly a factor of two. This is as expected, because when the distance to the goal is short, the rewards are not very sparse and Q-Learning easily finds the final goal. For these cases, Q-RRT spends additional training steps to build a tree. This adds computation with no additional benefit. Additionally, there is a large standard deviation for size 50 grid worlds that needs to be addressed.

As the size of the grid world increases, the reward distribution becomes more sparse and Q-Learning takes longer to converge. For the 50×50 grid world, the mean performance of Q-RRT is about the same as Q-Learning. Here, the extra training steps required to train the Q-Tables for each edge of the tree are balanced out by the increased exploration bias provided by the tree.

Q-RRT proved to be effective when run in mazes like the one show in Fig. 3a. These environments contain a series of bottlenecks, and thus pose a much greater challenge to Q-learning than open grid worlds. In all of the mazes tested, ESeQ was able to converge on a solution to the maze with less training than the standard Q-Learning. Table I shows the computational cost of ESeQ versus standard Q-Learning (QLN), in terms of the number of iterations required to converge. Table II divides the ESeQ algorithm into the steps devoted to generating the graph in Q-RRT and those devoted to backing up along the path in SeQ. Fig. 4 in the appendix shows a comparison of the estimated value function found by ESeQ to the baseline found by value iteration. The heat maps shown in the figure are nearly identical, except for in the subset of the state space which is far from both the goal and the starting position, and therefore was not explored. Fig. 5 in the appendix shows more details of these mazes and the ESeQ process, including the policies between nodes and the overall trajectory from start to goal.

In a state space full of obstacles like a maze, it becomes increasingly likely that a randomly sampled point will lie within an obstacle. The shape of the random intermediate reward is adjusted to counter this. The proposed reward is a Gaussian distribution centered on the randomly sampled state, which assigns a reward to every state in the neighborhood of this state. Using this method, the algorithm can add states other than the sampled point to the tree; rather, the node which is added to the tree is the local maximum to which the agent converges. In this sense, this formulation of Q-RRT is more akin to traditional RRT.

While this formulation of Q-RRT successfully navigates the maze, it is clear that the final trajectory and policy are far from optimal. For example, the output from each step of ESeQ is shown in Fig. 3.b. The intermediate paths found by Q-RRT, shown in red, include extra deviations and backtrack through some corridors. However, the final ESeQ trajectory, shown in black, is very close to optimal. This change shows that the SeQ step is refining the trajectory to a shorter path for this environment. Note that SeQ does not guarantee global

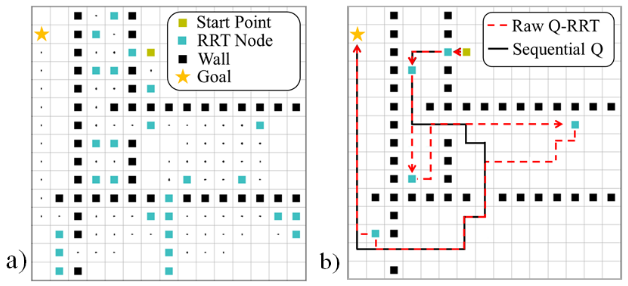

Figure 3: Solutions found by the ESeQ growth in a maze showing a) the Q-RRT growth filling the work space and b) the optimal path found by the Sequential Q refinement algorithm.

Table I: Comparison Between QLN and ESeQ by Random Walk Steps.

		QLN			ESeQ	
World	**Trials**	**Mean**	**Std**	**Trials**	**Mean**	**Std**
10×10	2	$4.02e6$	–	40	$2.12e6$	$8.08e5$
15×15	2	$9.05e6$	–	40	$8.27e6$	$3.71e6$

optimality, as SeQ only returns a Q matrix that is optimized locally around the path found by Q-RRT.

VI. Conclusion

Combining RRT with Q-learning provides us with a means of computing a locally-optimal trajectory in the presence of sparse rewards. However, the algorithm is more computationally intensive than standard Q-learning in regions of the state space where rewards are not sparse. This suggests that perhaps the branch rate of the RRT portion of the algorithm should be adjusted dynamically, to account for the sparsity pattern of the rewards.

To obtain trajectories which are closer to the global optimum, we propose an RRT*-like method for re-wiring the RRT. After the RRT has branched through a large part of the state space and has reached a reward, we select a random vertex in the RRT and try rewiring it to its neighbors in the tree (i.e. running Q-learning between the two nodes). We then measure the total discounted sum of rewards over all of the paths from start to goal produced by this process and select the path with minimum cost, iterating until convergence. This process is described in the appendix under Algorithm 3.

Another path of improvement is to look at feature extraction between policies. For every branch connecting two nodes in our algorithm, there is a respective Q-table approximation.

Table II: Iterations (ESeQ Component Random Walk Steps)

		Q-RRT		SeQ	
World	**Trials**	**Mean**	**Std**	**Mean**	**Std**
10×10	40	$1.07e6$	$5.89e5$	$1.05e6$	$3.48e5$
15×15	40	$4.87e6$	$4.87e5$	$3.40e6$	$1.11e6$

Comparing or analyzing each Q-table between "drive-to" exploration branches of the RRT could allow better Q-table initialization by encoding obstacles or traps found by previous exploration. Perhaps mutual information could provide insight to initialize the next Q-table and propose a bias of destination of travel during Q-RRT [23].

The ESeQ algorithm is not limited to a discrete state space. With the extension of the intermediate rewards to a Gaussian distribution, the algorithm could be extended to a continuous state space, using e.g. deep learning.

Within the allotted time for these experiments, much of the time was devoted to gathering simulation data. Because this was such a time-intensive task, tuning hyper-parameters with a standard grid search would take on the order of several days. To remedy this issue, code can be optimized for GPU acceleration.

Although the majority of this discussion was focused on Q-RRT, the performance of SeQ needs to be analyzed more thoroughly and improved. It may be possible to combine all the Q-tables on the edges in the path leading to the goal so that initializing a Q-Learning algorithm with this information across the entire state space would lead to convergence in fewer iterations [24].

References

[1] J. Hare, "Dealing with sparse rewards in reinforcement learning," 2019.
[2] C. J. Watkins, "Learning from delayed rewards (ph.d. thesis)," 1989.
[3] "Methods and apparatus for reinforcement learning, us patent #20150100530a1." 2015.
[4] M. Lai, "Giraffe: Using deep reinforcement learning to play chess," 2015.
[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.
[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," 2017.
[7] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, "Combining deep reinforcement learning and search for imperfect-information games," 2020.
[8] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games," 2016.
[9] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning," 2017.
[10] J. Hare, "Dealing with sparse rewards in reinforcement learning," 2019.
[11] L. Weitkamp, E. van der Pol, and Z. Akata, "Visual rationalizations in deep reinforcement learning for atari games," 2019.
[12] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, "#exploration: A study of count-based exploration for deep reinforcement learning," 2017.
[13] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," 06 2016.
[14] L. E. Kavraki, P. Svestka, J. 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.
[15] S. LaValle, "Rapidly-exploring random trees : a new tool for path planning," *The annual research report*, 1998.

[16] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, 2000, pp. 995–1001 vol.2.

[17] 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)*, 2018, pp. 5113–5120.

[18] J. Guzzi, R. O. Chavez-Garcia, M. Nava, L. M. Gambardella, and A. Giusti, "Path planning with local motion estimations," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 2586–2593, 2020.

[19] R. S. Sutton and A. G. Barto, "Reinforcement learning i: Introduction," 1998.

[20] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," October 1998.

[21] J. Kuffner and S. LaValle, "Space-filling trees," 2009.

[22] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," 2010.

[23] X. Lu, S. Tiomkin, and P. Abbeel, "Predictive coding for boosting deep reinforcement learning with sparse rewards," 2020.

[24] X. Lin, H. Baweja, G. Kantor, and D. Held, "Adaptive auxiliary task weighting for reinforcement learning," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 4772–4783.

VII. Appendix

Algorithm 3: Tree Re-wiring

```
Data: a tree $T = \{V, E\}$ rooted at $s_0$ with a leaf at
      $s_f$, a Q-learning tolerance tol
Result: $\Omega$, a path from $s_0$ to $s_f$
Init: $\Omega \leftarrow$ Path($T, s_0, s_f$)
while $|\Omega| > tol$ do
    $s_{rand} \leftarrow$ random($V$)
    $N \leftarrow$ neighborhood($s_{rand}$)
    for $s$ in $N$ do
        $Q_s \leftarrow$ Local Q-learning($s, s_{rand}, r$)
        $\{T, \Omega\} \leftarrow$ NewTrees($Q_s$) (Get collection of
         new trees and paths)
        $T, \Omega \leftarrow$ MinCost($\{T\}$) (Get tree with min-cost
         path)
    end
end
```

Algorithm 4: Q-Learning

```
Data: $s_0, \alpha, \gamma$, tol, $N_{steps}$
Result: Q
Init: $Q \leftarrow random$;
while $|\hat{Q} - Q| > tol$ do
    $s \leftarrow s_0$;
    for $i$ in $1 : N_{steps}$ do
        $u =$unif$(0, 1)$;
        if $u < \epsilon$ then
            $a_t = rand(A)$;
        else
            $a_t =$argmax$_A(R)$;
        end
        $s_new$ = Transition($s, a$);
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t +$
          $\gamma \max_a Q(s_new, a) - Q(s_t, a_t)\big)$;
    end
    $\hat{Q} \leftarrow Q$;
end
```

Algorithm 5: Rapidly-exploring random tree (RRT)

```
Data: state-space $S$, start state $s_0 \in S$, tree growth
      rate $\lambda$
Result: a tree $T = \{V, E\}$ rooted at $s_0$
Init: $V \leftarrow \{s_0\}$
$E \leftarrow \emptyset$
$s_{new} \leftarrow s_0$
while $Q = 0$ do
    $s_{new} \leftarrow$ random($S$)
    $d \leftarrow$ dist($V, s_{new}$)
    $s_{nearest} \leftarrow$ argmin($d$)
    if $d > \lambda$ then
        $s_{new} \leftarrow \lambda\times$ heading($s_{nearest}, s_{new}$)
    end
    $e_{new} \leftarrow \{s_{nearest}, s_{new}\}$
    if $e_{new}$ is a viable edge then
        $V \leftarrow \{V, s_{new}\}$
        $E \leftarrow \{E, e_{new}\}$
    end
end
```

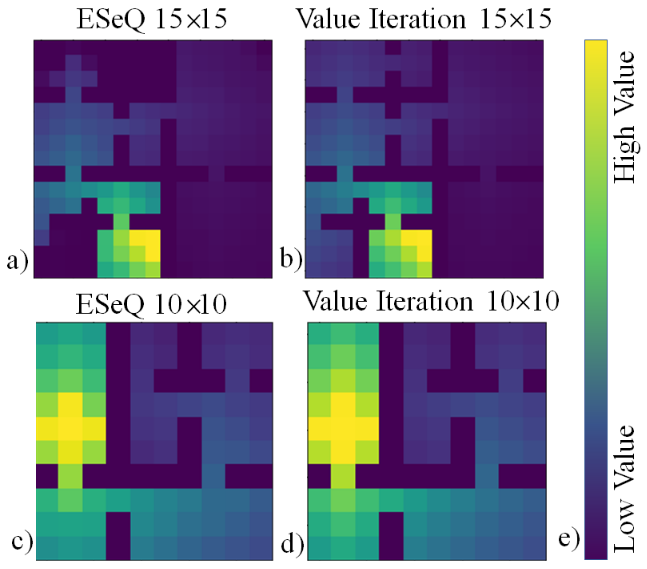

Figure 4: Final value functions computed in the 10×10 and 15×15 mazes with ESeQ and Value Iteration. a) Heat map of the global value function after the SeQ process for the 15x15 maze b) Reference value iteration heat map for the 15×15 maze c) Result of the SeQ algorithm for the 10×10 maze d) Value iteration for the 10×10 maze e) Gradient competitor bar indicating high value colors and low value colors.

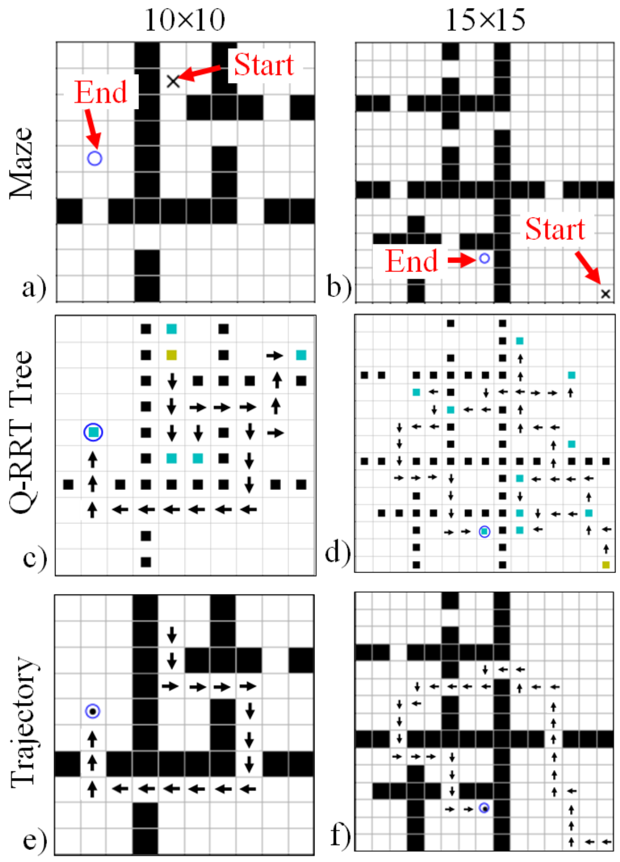

Figure 5: Shown above are the mazes used to compare the performance of ESeQ to Q-Learning, along with samples of typical ESeQ outputs. a) The 10×10 maze with the start and ending locations. b) The 15×15 maze with the start and ending locations c) An example of a Q-RRT Tree grown to explore the 10×10 maze d) An example of a Q-RRT tree grown to explore the 15×15 maze e) The optimal trajectory found by the ESeQ algorithm for the 10×10 maze f) The optimal trajectory found by the ESeQ algorithm for the 15×15 maze.

Table III: Hyperparameters used for an $n \times n$ Gridworld.There is a different set of hyperparameters for the open gridworld with no obstacles and the maze gridworld. These parameters are the total number of episodes (N_e), episodes per epoch (epoch), actions in a trajectory (N_t), loss tolerance allowed for convergence (tol), learning rate (α), maximum allowed intermediate reward distance (λ), greedy parameter (ϵ), discount factor (γ), and horizon time (H).

	QLN (open)	**ESeQ (open)**	**QLN (maze)**	**ESeQ (maze)**
N_e	5e8	2e4	5e8	4e6
epoch	50	50	50	50
N_t	n^2	n^2	n^2	n^2
tol	0.5	5	5	2
α	0.4	0.4	0.4	0.4
λ	–	7	–	7
ϵ	0.4	0.4	0.4	0.4
γ	0.9	0.9	0.9	0.9
H	–	200	–	200