LEARNING TO PLAN VIA NEURAL EXPLORATION-EXPLOITATION TREES

Binghong Chen*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
binghong@gatech.edu

Bo Dai*
Google Brain
Mountain View, CA 94043, USA
bodai@google.com

Le Song
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
lsong@cc.gatech.edu

ABSTRACT

Sampling-based planning algorithms such as RRT and its variants are powerful tools for path planning problems in high-dimensional continuous state and action spaces. While these algorithms perform systematic exploration of the state space, they do not fully exploit past planning experiences from similar environments. In this paper, we design a meta path planning algorithm, called *Neural Exploration-Exploitation Trees* (NEXT), which can utilize prior experience to drastically reduce the sample requirement for solving new path planning problems. More specifically, NEXT contains a novel neural architecture which can learn from experiences the dependency between task structures and promising path search directions. Then this learned prior is integrated with a UCB-type algorithm to achieve an online balance between *exploration* and *exploitation* when solving a new problem. Empirically, we show that NEXT can complete the planning tasks with very small search trees and significantly outperforms previous state-of-the-arts on several benchmark problems.

1 INTRODUCTION

Planning paths efficiently in a high-dimensional continuous state and action space is a fundamental yet challenging problem in many real-world applications, such as robot manipulation and autonomous driving. Since the general path planning problem is PSPACE-complete (Reif, 1979), one typically resorts to approximate or heuristic algorithms.

Tree-based sampling planner, such as probabilistic roadmaps (PRM) (Kavraki et al., 1996), rapidly-exploring random trees (RRT) (LaValle, 1998), and their variants (Karaman & Frazzoli, 2011), provide principled approximate solutions to a wide spectrum of high-dimensional path planning tasks. These algorithms can be summarized in the template Algorithm 1 with different `Expand` operators[1]. However, these algorithms typically employ a uniform proposal distribution for sampling which does not make use of the structures of the problem at hand and thus may require lots of samples to obtain an initial feasible solution path for complicated tasks. To improve the sample efficiency, researchers designed algorithms to take problem structures into account (Boor et al., 1999; Hsu et al., 2003; Shkolnik et al., 2009; Gammell et al., 2014; 2015), to name a few. Despite that, all these improved samplers are designed manually to address specific structural properties, which may or may not be valid for a new task, and thus, may lead to even worse performance compared to the uniform proposal.

Can we exploit past path planning experiences, and learn an efficient and generalizable sampling algorithm for future planning tasks? Pioneering works in this direction are limited in one way

*indicates equal contribution.

[1]Please refer to Appendix A for more context about tree-based sampling algorithms.

Algorithm 1: Tree-based Sampling Algorithm (TSA)

Data: Planning Task

U = initial state: s_{init}, goal state region: $\mathcal{S}_{goal}$, configuration state space: $\mathcal{S}$, free state space: $\mathcal{S}_{free}$, workspace `map`, cost function: $c(\cdot)$

Initialize Tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} \leftarrow \{s_{init}\}$ and $\mathcal{E} \leftarrow \emptyset$;
for $t \leftarrow 0$ **to** T **do**
  $s_{parent}, s_{new} \leftarrow$ `Expand`$(\mathcal{T}, U)$;
  if `ObstacleFree`(s_{parent}, s_{new}) **then**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{s_{new}\}$ and $\mathcal{E} \leftarrow \mathcal{E} \cup \{(s_{parent}, s_{new})\}$;
    $\mathcal{T} \leftarrow$ `Postprocess`$(\mathcal{T}, U)$; ▷ Optional
    if $s_{new} \in \mathcal{S}_{goal}$ **then**
      return $\mathcal{T}$;

or the other. Zucker et al. (2008); Zhang et al. (2018) treat the sampler as a stochastic policy to be learned and apply policy gradient methods to improve the policy. Finney et al. (2007); Ye & Alterovitz (2017); Bowen & Alterovitz (2014); Ichter et al. (2018); Kim et al. (2018); Kuo et al. (2018) apply imitation learning based on the collected demonstations to introduce bias for better sampler via multiple probabilistic models. However, each of these approach either simply rely on special hand-designed local features or assume the tasks are indexed by special parameters. As we will show in the experiments, such representations limit the generalization ability of these algorithms. Although deep learning based approach, such as value iteration networks (VIN) (Tamar et al., 2016) and gated path planning networks (GPPN) (Lee et al., 2018), can learn task representations, they are not directly applicable to continuous state and action spaces within high dimensional problems.

Another key issue in learning to plan which has been largely ignored is the online exploration-exploitation trade-off. Arguably the stochasticity in the biased sampler will provide a certain degree of exploration, and the inductive biases in the learned samplers will lead to exploitation. However, existing algorithms do not explicitly balance the exploration and exploitation in a principled online fashion in a new planning task.

In this work, we propose a substantially improved algorithm, *Neural EXploration-EXploitation Tree* (NEXT), for learning in path planning, leveraging recent advances in representation learning and infinite-armed bandit problem. Our algorithm contains a novel neural architecture which can learn from experiences the dependency between task structures and promising search directions. This learned neural prior is integrated with a UCB-type algorithm to achieve an online balance between *exploration* and *exploitation* when solving a new problem. Empirically, we show that NEXT can exploit past experience to reduce the sample requirement drastically for solving new planning problems, and significantly outperforms previous state-of-the-arts on several benchmarks.

2 Neural Exploration-Exploitation Trees

We will leverage UCB algorithms to explicitly take exploration versus exploitation trade-off into account, leading to a novel expansion operator in Algorithm 2, which will be improved through a meta-imitation learning with novel neural architecture in Algorithm 3. Integrating this with the tree-based sampling planner in Algorithm 1, we obtain the *Neural Exploration-Exploitation Trees (NEXT)*.

2.1 Guided Progressive Expansion

We first introduce the *Guided Progressive Expansion* (GPE) with the assumption that a value function oracle $V^*(s|U)$, defined as the cost of the shortest path from state s to the goal, is provided for any state s of a problem U. Since we do not have direct access to V^*, we will estimate a surrogate $\tilde{V}^*$ instead. We will postpone the learning of such value function from past experience to Section 2.2.

The purpose of the `Expand` operator is to expand the current search tree $\mathcal{T}$ with a new neighboring state $s_{new} \in \mathcal{B}(\mathcal{T})$, where $\mathcal{B}(\mathcal{T}) = \bigcup_{s \in \mathcal{V}} \mathcal{B}(s)$, $\mathcal{B}(s) = \{s' \in \mathcal{S} \mid \|s' - s\| \leqslant \eta\}$ with η is the radius of the neighborhood. We consider the expansion as a two-step procedure

(i) We select a state s from the existing tree $\mathcal{T}$;

(ii) We select a state s_{new} in the neighborhood of the selected s, and add it to $\mathcal{T}$.

Selection from $\mathcal{T}$. Recall that in every step, we always have a finite number of nodes in the search tree $\mathcal{T}(\mathcal{V}, \mathcal{E})$. Therefore step **(i)** shares some similarity with the multi-armed bandit problem by

Algorithm 2: `NEXT` :: `Expand`$(\mathcal{T}, U)$

Data: $\mathcal{T} = (\mathcal{V}, \mathcal{E}), U = (s_{init}, \mathcal{S}_{goal}, \mathcal{S}, \mathcal{S}_{free}, \texttt{map}, c(\cdot))$

$s_{parent} \leftarrow \text{argmax}_{s \in \mathcal{V}_t} \phi(s)$; ▷ Selection

candidates $\leftarrow$ Sample k states from $\pi(s'|s_{parent}, U)$; ▷ Guided candidates generation

$s_{new} \leftarrow \text{argmax}_{s' \in \text{candidates}} \phi(s')$; ▷ Progressive expansion

return s_{parent}, s_{new};

viewing the existing nodes $\{s\} \in \mathcal{V}$ as arms and the negative value function estimate $-\tilde{V}^*(s|U)$ as the rewards $r(s)$. However, as the algorithm executed, the number of states is increasing. Consequently, we cannot use the vanilla UCB algorithm. The problem can be solved by smoothing the reward $r(s)$ via Gaussian Process or kernel regression (Yee et al., 2016). Then, the upper confidence bounds (UCB) of the reward function, $\phi(s) : \mathcal{V} \to \mathbb{R}$ can be computed, and the selection of the next node $s \in \mathcal{V}$ to expand will be carried out using $\phi(s)$, *i.e.*, $s^{t+1} = \text{argmax}_{s \in \mathcal{V}} \phi(s)$. We denote the sequence of selected nodes from the current tree as $\mathcal{S}_t = \{s_1, s_2, \ldots, s_t\}$. Note that some nodes in the tree may be selected multiple times. We list two examples of constructing the UCB due to different smoothing parametrizations:

- **GP-UCB**: GP-UCB maintains an UCB of the reward after t-step as

$$\phi(s) := \bar{r}_t(s) + \lambda \sigma_t(s), \tag{1}$$

where $\bar{r}_t(s) = k_t(s)(K_t + \alpha I)^{-1} r_t, \sigma_t^2(s) = k(s,s) - k_t(s)^\top (K_t + \alpha I)^{-1} k_t(s)$, with $k_t(s) = [k(s_i, s)]_{s_i \in \mathcal{S}_t}$ and $K_t = [k(s, s')]_{s,s' \in \mathcal{S}_t}$.

- **KS-UCB**: kernel smoothing UCB (Yee et al., 2016) maintains the reward after t-step as

$$\phi(s) := \frac{\sum_{s' \in \mathcal{S}_t} k(s', s) r(s')}{\sum_{s' \in \mathcal{S}_t} k(s', s)} + \lambda \sqrt{\frac{\log \sum_{s' \in \mathcal{S}_t} w(s')}{w(s)}}, \tag{2}$$

with $w(s) = \sum_{s' \in \mathcal{S}_t} k(s', s)$.

Generating reachable state. We consider the reachable state generation as another bandit problem but with infinite arms in $\mathcal{B}(s)$. We approach it with a policy $\tilde{\pi}^*(s'|s, U)$ to generate some candidates and select the one that has the largest $\phi(s)$ value. The policy $\tilde{\pi}^*$ will provide some bias in sampling. As we will explain in more details in Section 2.2, $\tilde{\pi}^*$ is trained to mimic the optimal policy π^* from the previous experiences across different tasks.

With these details on step **i)** and **ii)** explained above, we obtain our novel `NEXT` :: `Expansion` in Algorithm 2, which is illustrated in Figure 5(b) and (c) in Appendix C. We next show how we can obtain the value function estimator $\tilde{V}^*(s|U)$ and learn the guiding policy $\tilde{\pi}^*(s'|s, U)$ in the expansion operator, as illustrated in Figure 5(d) using experiences from previous planning tasks.

2.2 Neural Architecture

In this section, we introduce the neural architecture and learning scheme for $\tilde{V}^*(s|U)$ and $\tilde{\pi}^*(s'|s, U)$. The design of the neural architecture is inspired by the VIN (Tamar et al., 2016) and GPPN (Lee et al., 2018), but with significant differences. The major challenge is that we only have access to low dimensional workspace map (as per our assumption), but we need to predict the value and construct policy for high dimensional states in configuration space. To address this challenge, we will design a novel attention-based configuration space embedding modules, which allows us to perform value iteration in the embedded space.

Configuration space embedding. Since the configuration space is high dimensional and continuous, VIN and GPPN cannot be directly applied. We will design a novel attention-based network to embed the configuration space into a 3d tensor, on which a planning module with recursive structure similar to value iteration networks can be applied.

More specifically, we will use s^w to explicitly denote the workspace part of state $s \in \mathcal{S}$, and s^h to denote the remaining dimensions of the state, *i.e.* $s = (s^w, s^h)$. s^w and s^h will be embedded using different neural architectures. For simplicity of notation, we will focus on the 2d workspace case here, where $s^w \subset \mathbb{R}^2$. However, we emphasize that our method applies to the 3d workspace as well.

- For s^w, the embedding $\mu_{\theta^w}(s^w)$ is a $d \times d$ matrix of the same size as `map`, and it is computed using k_w convolution layers, *i.e.*,

$$\mu_{\theta^w}(s^w) = \texttt{softmax2d}(f^w_{k_w}(s^w)), \quad f^w_{i+1}(s^w) = \texttt{relu}(\theta^w_i \oplus f^w_i(s^w)), \tag{3}$$

where $\oplus$ is the convolution operation, each θ_i^w is a convolution kernel of size 1×1, $f_i^w(s^w) \in \mathbb{R}^{d \times d \times k_i^w}$ for $i > 0$, k_i^ws are hyper-parameters, θ_0^w is a convolution kernel of size $1 \times 1 \times 4$, and $f_0^w(s^w)$ is a tensor of size $d \times d \times 4$ filled with

$$f_0^w(s^w)_{ijl} = \begin{cases} s_l^w, & \text{if } l \in \{1, 2\}, \\ i, & \text{if } l = 3, \\ j, & \text{otherwise.} \end{cases} \tag{4}$$

• For s^h, the higher dimensional configuration embedding $\mu_{\theta^h}(s^h)$ is computed using k_h fully-connected layers, *i.e.*

$$\mu_{\theta^h}(s^h) = \texttt{softmax}(f_{k_h}^h(s^h)), \quad f_{i+1}^h(s^h) = \texttt{relu}(\theta_i^{hw} f_i^h(s^h) + \theta_i^{hb}), \tag{5}$$

where $\mu_{\theta^h}(s^h) \in \mathbb{R}^{d_a}$ and $f_0^h(s^h) = s^h$.

We can obtain $\mu_\theta(s)$ by multiplying $\mu_\theta^w(s^w)$ with $\mu_\theta^h(s^h)$,

$$\mu_\theta(s)_{ijl} = \mu_\theta^w(s^w)_{ij} \cdot \mu_\theta^h(s^h)_l. \tag{6}$$

The overall architecture of the attention-based embedding module is illustrated in Figure 6 in Appendix C. The result of the embedding of a state s is a $d \times d \times d_a$ tensor attention map

$$\mu_\theta(s) \text{ with } \mu_\theta(s)_{ijl} \geqslant 0, \text{ and } \sum_{ijl} \mu_\theta(s)_{ijl} = 1,$$

where θ denotes the parameters in the embedding network.

Overall model architecture. With the attention-based embedding of the configuration spaces, we can apply planning module on top of it. First, we will produce the embedding $\mu_\theta(s_{init})$ and $\mu_\theta(s_{goal})$ of the initial state s_{init} and the goal state s_{goal} respectively. Then we parameterize $\tilde{V}^*(s|U)$ and $\tilde{\pi}^*(s'|s, U)$ in the following way.

The input to the planning module is computed by a CNN, $\tilde{V}^{*(0)}, \tilde{R} = \sigma(W_0 \oplus [\mu_\theta(s_{goal}), \texttt{map}])$, where $[\mu_\theta(s_{goal}), \texttt{map}]$ denotes the concatenation of $\mu_\theta(s_{goal})$ and map along the 3rd dimension, and W_0 is a 3d convolution kernel of size $k \times k \times (d_a + 1)$. After T iterations of Bellman update $\tilde{V}^{*(t)} = \min\left(W_1 \oplus \left[\tilde{V}^{*(t-1)}, \tilde{R}\right]\right)$, we have $\tilde{V}^{*(T)}$. Notice that since the Bellman update is performed in the embedding space, $\tilde{V}^{*t}$ is a tensor of size $d \times d \times d_e$, and W_1 is a 3d convolution kernel of size $k \times k \times (d_e + 1)$, where d_e is a multiple of d_a, $d_e = d_a \cdot p$.

Then we extract $\psi(s_{init}) \in \mathbb{R}^p$ by multiplying $\tilde{V}^{*(T)}$ (of size $d \times d \times d_a \times p$) with attention $\mu_\theta(s_{init})$,

$$\psi(s_{init})_e = \sum_{ijl} \tilde{V}_{ijle}^{*(T)} \cdot \mu_\theta(s_{init})_{ijl}, \tag{7}$$

and finally we obtain $\tilde{V}^*(s_{init}|U), \tilde{\pi}^*(s'|s_{init}, U) = h_{W_2}(\psi(s_{init}))$.

For the details of the parameterization in our implementation, please refer to the Appendix B. We illustrate the overall model architecture in Figure 7 in Appendix C. The parameters $W = (W_0, W_1, W_2, \theta)$ will be learned together.

2.3 META SELF-IMPROVING LEARNING

The learning of the parameters in $\tilde{V}^*(s|U)$ and $\tilde{\pi}^*(s'|s, U)$ are carried out *simultaneously* with planning. We do not have an explicit training and testing phase separation. Particularly, we use a mixture of RRT :: Expand and NEXT :: Expand with probability ϵ and $1 - \epsilon$, respectively, inside the tree-based planning algorithm in Algorithm 1. The RRT* postprocessing step will be used in the template. The ϵ is set to be 1 at the initial stage since the $\{\tilde{V}^*, \tilde{\pi}^*\}$ is not well-trained, and thus, the algorithm behaves like RRT*. As the training proceeds, we anneal ϵ gradually as the sampler becomes more and more efficient.

The dataset $\mathcal{D}_n = \{\mathcal{T}_j, U_j\}_{j=1}^n$ for the n-th training epoch is collected from the previous planning experiences. For a tree $\mathcal{T} \in \mathcal{D}_n$, let m denote the length of the solution path, we can reconstruct the successful path $\{s^i\}_{i=1}^m$, and the value for each state in the path will be the sum of cost to the end of the path, *i.e.*, $\left\{y^i := \sum_{l=i}^{m-1} c\left(\left[s^l, s^{l+1}\right]\right)\right\}_{i=1}^m$. We learn $\{\tilde{V}^*, \tilde{\pi}^*\}$ by optimizing,

$$\min_W \sum_{\mathcal{T} \sim \mathcal{D}_n} \ell\left(\tilde{V}^*, \tilde{\pi}^*; \mathcal{T}\right) := -\sum_{i=1}^{m-1} \log \tilde{\pi}^*\left(s^{i+1}|s^i\right) + \sum_{i=1}^{m} (\tilde{V}^*\left(s^i\right) - y^i)_2^2 + \lambda \|W\|^2. \tag{8}$$

We will apply the stochastic gradient descent to minimize (8) w.r.t. W.

Algorithm 3: `MSIL`: Meta Self-Improving NEXT Learning

Initialize dataset $\mathcal{D}$;
for epoch $i \leftarrow 1$ **to** N **do**
  Sample a planning instance U from some distribution;
  $\mathcal{T} \leftarrow \texttt{TSA}(U)$ with $\epsilon \sim \mathcal{U}nif[0,1]$, and $\epsilon \cdot \texttt{RRT} :: \texttt{Expand} + (1-\epsilon) \cdot \texttt{NEXT} :: \texttt{Expand}$;
   Postprocessing with $\texttt{RRT}^*$ :: `Postprocess`;
  $\mathcal{D}_n \leftarrow \mathcal{D}_n \cup \{\mathcal{T}, U\}$;
  for $j \leftarrow 0$ **to** L **do**
    Sample $\{\mathcal{T}, U\}$ from $\mathcal{D}_n$;
    Reconstruct optimal path $\{s^i\}_{i=1}^m$ and the cost of paths based on $\mathcal{T}$;
    Update the $W \leftarrow W - \eta \nabla_W \ell(\tilde{V}^*, \tilde{\pi}^*; \mathcal{T})$;
  Anneal $\epsilon = \alpha\epsilon$, $\alpha \in (0,1)$;
return W

On the one hand, the objective (8) is making the policy imitate the successful policy and improving the value function estimation based upon the outcomes from the NEXT algorithm itself on previous tasks. On the other hand, the updated $\{\tilde{V}^*, \tilde{\pi}^*\}$ will be applied in the next epoch to improve the performance of planning. Therefore, we named the learning algorithm as *Meta Self-Improving Learning* (MSIL). The learning procedure is summarized in Algorithm 3 and illustrated in Figure 5.

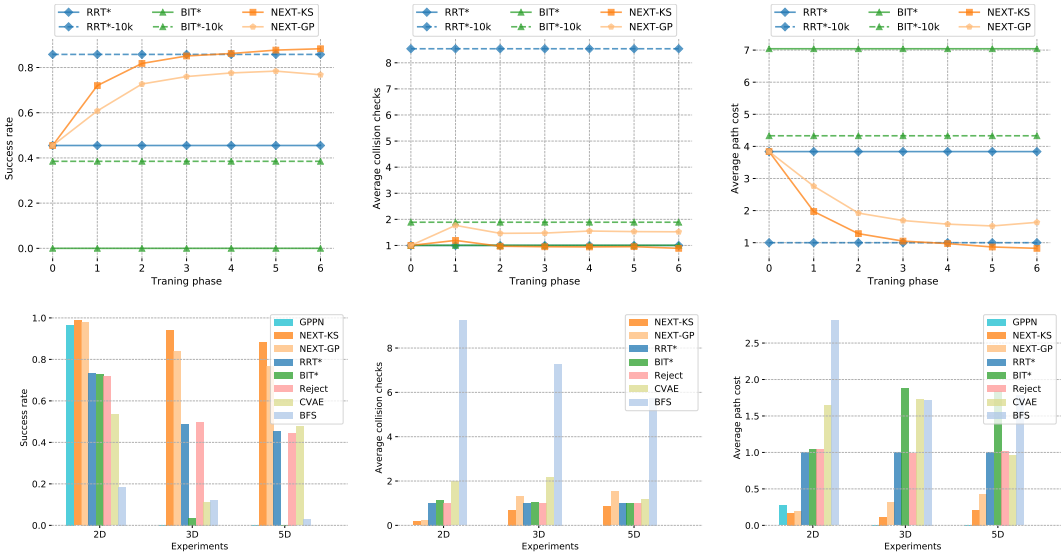


Figure 1: Top row: improvement curves evaluated at different training stages of the 5D experiment; Bottom row: comparison between the NEXT and existing algorithms. The performance of all algorithms is evaluated with the last 1000 tasks for each experiment. All algorithms are restricted to use only 500 samples, except RRT*-10k and BIT*-10k, which are allowed to use 10,000 samples.

3 Experiments

We compared NEXT with RRT* (Karaman & Frazzoli, 2011), BIT* (Gammell et al., 2015), CVAE-plan (Ichter et al., 2018) and Reject-plan (Zhang et al., 2018) in terms of planning time and solution optimality. We compared these algorithms on three types of planning tasks: workspace planning, rigid body navigation, and 3-link snake[2], whose configuration spaces are $\mathbb{R}^2$, $\mathbb{R}^3$ and $\mathbb{R}^5$, respectively. For each experiment, we generated 3000 different tasks from the same distribution. We trained the learning-based baselines CVAE-plan and Reject-plan using the first 2000 tasks, reserved the rest for testing. We let NEXT improve itself using MSIL over the first 2000 tasks. In this period, for every 200 tasks, we updated its parameters and annealed ϵ once.

[2]Please refer to Figure 9 in Appendix D.2 for an example of all three types of tasks.

To systematically evaluate the algorithms, we recorded the time (measured by the number of collision checks used) needed to find a collision-free path, the success rate within time limits, and the cost of the solution path for each run. The results of the reserved 1000 test tasks of each experiment are shown in the bottom row of Figure 1. Both the KS-UCB and the GP-UCB version of NEXT outperform its competitors by a large margin under all three criteria. We plot the performance improvement curve of our algorithms on the 5D planning tasks in the top row of Figure 1. For comparison, we also plot the performance of RRT* and BIT*. At the beginning phase of self-improving, our algorithms are comparable to RRT*. They then gradually learn from previous experiences and improve themselves as they see more tasks and better solutions. In the end, NEXT-KS is able to match the performance of RRT*-10k using only one-twentieth of its samples! For more experiments details and results, please refer to Appendix D.

ACKNOWLEDGEMENTS

LS is supported in part by NSF IIS-1218749, NIH BIGDATA 1R01GM108341, NSF CAREER IIS-1350983, NSF IIS1639792 EAGER, NSF IIS-1841351 EAGER, NSF CCF-1836822, NSF CNS-1704701, ONR N00014-15-1-2340, Intel ISTC, NVIDIA, Amazon AWS, Siemens and Google Cloud.

REFERENCES

Valérie Boor, Mark H Overmars, and A Frank Van Der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Robotics and automation, 1999. proceedings. 1999 ieee international conference on*, volume 2, pp. 1018–1023. IEEE, 1999.

Chris Bowen and Ron Alterovitz. Closed-loop global motion planning for reactive execution of learned tasks. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1754–1760. IEEE, 2014.

Sarah Finney, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Predicting partial paths from planning problem parameters. In *Robotics Science and Systems*, 2007.

Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. *arXiv preprint arXiv:1404.2334*, 2014.

Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 3067–3074. IEEE, 2015.

David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 3, pp. 2719–2726. IEEE, 1997.

David Hsu, Tingting Jiang, John Reif, and Zheng Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 3, pp. 4420–4426. IEEE, 2003.

Brian Ichter, James Harrison, and Marco Pavone. Learning sampling distributions for robot motion planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7087–7094. IEEE, 2018.

Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

Lydia E Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4), 1996.

Beomjoon Kim, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Yen-Ling Kuo, Andrei Barbu, and Boris Katz. Deep sequential models for sampling-based planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6490–6497. IEEE, 2018.

Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

Lisa Lee, Emilio Parisotto, Devendra Singh Chaplot, Eric Xing, and Ruslan Salakhutdinov. Gated path planning networks. *arXiv preprint arXiv:1806.06408*, 2018.

Rosanne Liu, Joel Lehman, Piero Molino, Felipe Petroski Such, Eric Frank, Alex Sergeev, and Jason Yosinski. An intriguing failing of convolutional neural networks and the coordconv solution. In *Advances in Neural Information Processing Systems*, pp. 9628–9639, 2018.

Jeff M Phillips, Nazareth Bedrossian, and Lydia E Kavraki. Guided expansive spaces trees: A search strategy for motion-and cost-constrained state spaces. In *IEEE International Conference on Robotics and Automation*, pp. 3968–3973, 2004.

John H Reif. Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pp. 421–427. IEEE, 1979.

Alexander Shkolnik, Matthew Walter, and Russ Tedrake. Reachability-guided sampling for planning under differential constraints. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 2859–2865. IEEE, 2009.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pp. 2154–2162, 2016.

Gu Ye and Ron Alterovitz. guided motion planning. In *Robotics research*, pp. 291–307. Springer, 2017.

Timothy Yee, Viliam Lisy, and Michael Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pp. 690–696. AAAI Press, 2016.

Clark Zhang, Jinwook Huh, and Daniel D Lee. Learning implicit sampling distributions for motion planning. *arXiv preprint arXiv:1806.01968*, 2018.

Matt Zucker, James Kuffner, and J Andrew Bagnell. Adaptive workspace biasing for sampling-based planners. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 3757–3762. IEEE, 2008.

Appendix

A TREE-BASED SAMPLING ALGORITHMS

Here we present a unifying view of many sampling-based planning algorithms including the RRT (LaValle, 1998), the RRT* (Karaman & Frazzoli, 2011), and the EST (Hsu et al., 1997; Phillips et al., 2004). These algorithms maintain a search tree $\mathcal{T}$ rooted at the initial point s_{init} and connecting all sampled points $\mathcal{V}$ in the configuration space with edge set $\mathcal{E}$. Then a path ξ from the initial position to any sampled point can be constructed via the shortest path in the search tree. Furthermore, this tree will be expanded incrementally by incorporating more sampled points until some tree leaf reaches the goal region $\mathcal{S}_{goal}$. At this point, a feasible solution for the path planning problem is found, corresponding to the shortest path between the root and this particular leaf. After that, more sampling can be conducted, and the tree can be further expanded to refine the path.

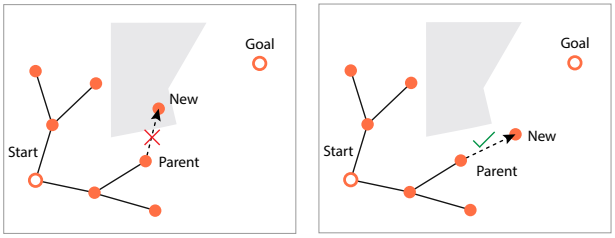


Figure 2: Illustration for one iteration of Algorithm 1. The left and right figures illustrate two different cases where the sample returned by the `Expand` operator is **unreachable** and **reachable** from the search tree.

The template of tree-based sampling algorithms is summarized in Algorithm 1 and illustrated in Figure 2. A key component of the algorithm is the tree `Expand` operator, which can be instantiated differently in different concrete algorithms (more discussion later). The `Expand` operator returns an existing node in the tree $s_{parent} \in \mathcal{V}$ and a new state $s_{new} \in \mathcal{S}$ sampled from the neighborhood of s_{parent}. Then the line segment $[s_{parent}, s_{new}]$ is passed to function `ObstacleFree` for collision checking. If the line segment $[s_{parent}, s_{new}]$ is collision-free (no obstacle in the middle, or called **reachable** from $\mathcal{T}$), then s_{new} is added to the tree vertex set $\mathcal{V}$, and the line segment is added to the tree edge set $\mathcal{E}$. If the newly added node s_{new} has reached the target $\mathcal{S}_{goal}$, the algorithm will return. Optionally, some concrete algorithms can define a `Postprocess` operator to refine the search tree. For an example of the `Expand` operator, as shown in Figure 5 (c), since there is no obstacle on the dotted edge $[s_{parent}, s_{new}]$, *i.e.*, s_{new} is reachable, the new state and edge will be added to the search tree (connected by the solid edges).

Now we will provide two concrete algorithm examples. For instance,

- If we instantiate the `Expand` operator as Algorithm 4, then we obtain the rapidly-exploring random trees (RRT) algorithm (LaValle, 1998), which first samples a state s from the configuration space $\mathcal{S}$ and then pulls it toward the neighborhood of current tree $\mathcal{T}$ measured by a ball of radius η:
$$\mathcal{B}(s, \eta) = \{s' \in \mathcal{S} \mid \|s' - s\| \leqslant \eta\}.$$
Moreover, if the `Postprocess` operator is introduced to modify the maintained search tree as in RRT* (Karaman & Frazzoli, 2011), the algorithm is provable to obtain the optimal path asymptotically.

- If we instantiate the `Expand` operator as Algorithm 5, then we obtain the expansive-space trees (EST) algorithm (Hsu et al., 1997; Phillips et al., 2004), which samples a state s from the nodes of the existing tree, and then draw a sample from the neighborhood of s.

Algorithm 4: RRT :: Expand$(\mathcal{T}, U)$

Data: $\mathcal{T} = (\mathcal{V}, \mathcal{E}), U = (s_{init}, \mathcal{S}_{goal}, \mathcal{S}, \mathcal{S}_{free}, \texttt{map}, c(\cdot))$

$s_{rand} \leftarrow \mathcal{U}nif(\mathcal{S})$; ▷ Sample configuration space

$s_{parent} \leftarrow \text{argmin}_{s \in \mathcal{V}} \|s_{rand} - s\|$; ▷ Pull to a tree node

$s_{new} \leftarrow \text{argmin}_{s \in \mathcal{B}(s_{parent}, \eta)} \|s - s_{rand}\|$;

return s_{parent}, s_{new};

Algorithm 5: EST :: Expand$(\mathcal{T}, U)$

Data: $\mathcal{T} = (\mathcal{V}, \mathcal{E}), U = (s_{init}, \mathcal{S}_{goal}, \mathcal{S}, \mathcal{S}_{free}, \texttt{map}, c(\cdot))$

$s_{parent} \sim \phi(s), s \in \mathcal{V}$; ▷ Sample a tree node

$s_{new} \leftarrow \mathcal{U}nif(\mathcal{B}(s_{parent}))$; ▷ Sample neighborhood

return $s_{nearest}, s_{new}$;

B Policy and Value Network Architecture

We explain the implementation details of the proposed parametrization for policy and value function. Figure 3 and Figure 4 are neural architectures for the attention module, the policy/value network, and the planning module, respectively.

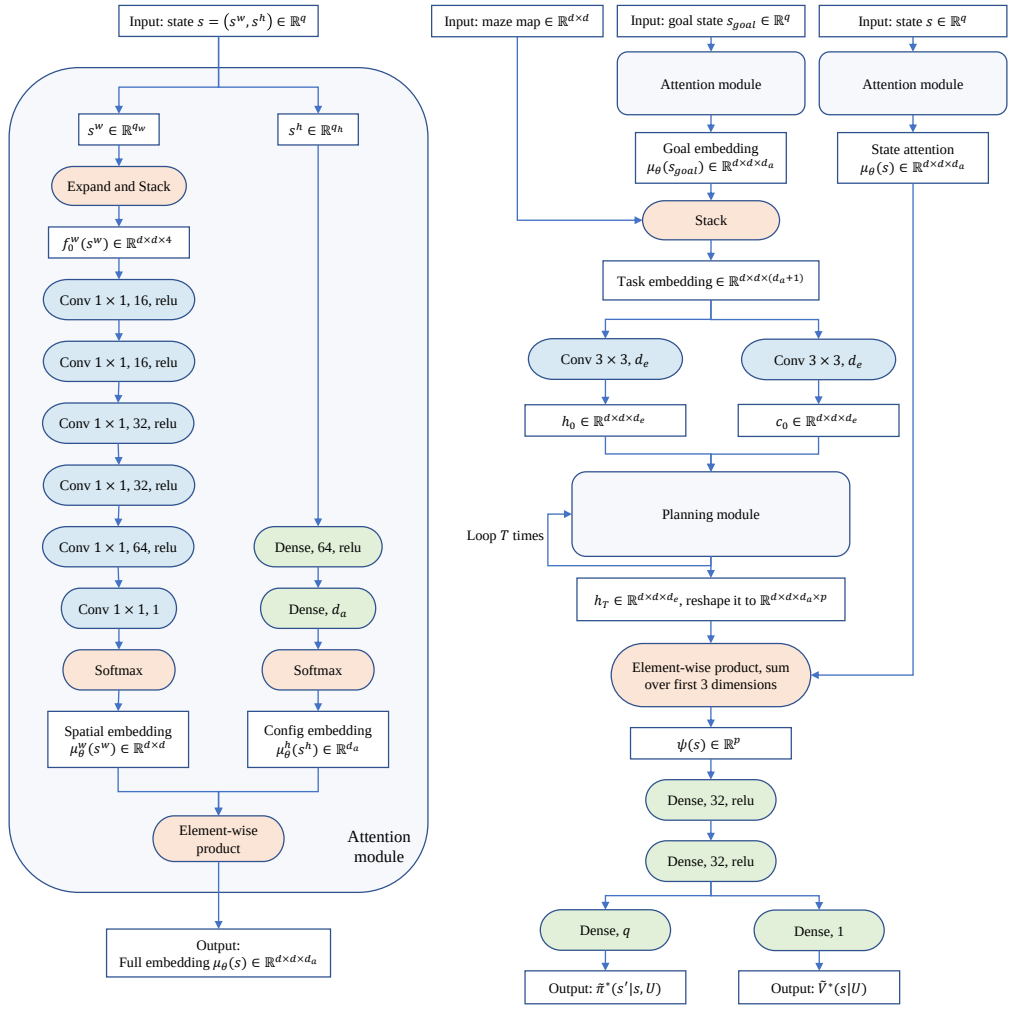


Figure 3: left: attention module, instantiating the Figure 6; right: policy/value network, instantiating the Figure 7.

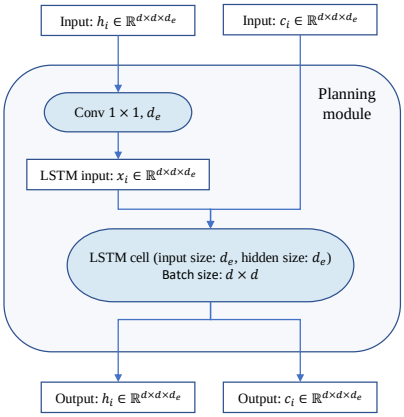


Figure 4: planning module

In the figures, we use rectangle blocks to denote inputs, intermediate results and outputs, stadium shape blocks to denote operations, and rounded rectangle blocks to denote modules. We use different colors for different operations. In particular, we use blue for convolutional/LSTM layers, green for dense layers, and orange for anything else. For convolutional layers, "Conv 1×1, 32, relu" denotes a layer with 1×1 kernels, 32 channels, followed by a rectified linear unit; for dense layers, "Dense, 64, relu" denotes a layer of size 64, followed by a rectified linear unit.

The attention module (Figure 3-left) embeds a state to a $d \times d \times d_a$ tensor. See equation (4) and (6) for details for computing $f_0^w(s^w)$ and $\mu_\theta(s)$. The planning module (Figure 4) is a one-step LSTM update which takes the result of a convolutional layer as input. Both the input and hidden size of the LSTM cell are d_e. All $d \times d$ locations share one set of parameters and are processed by the LSTM in one batch.

The main architecture is illustrated in Figure 3-right. It takes maze map, state and goal as input, and outputs the action and the value. Refer to equation (7) for details for computing $\psi(s)$. In our experiments, we set the values of the hyper-parameters to be $(d, d_e, d_a, p) = (15, 64, 8, 8)$.

C ALGORITHM ILLUSTRATIONS

We illustrate the learning process for NEXT, the attention module architecture, and the overall neural network architecture for policy and value function in Figure 5, Figure 6, and Figure 7, respectively.

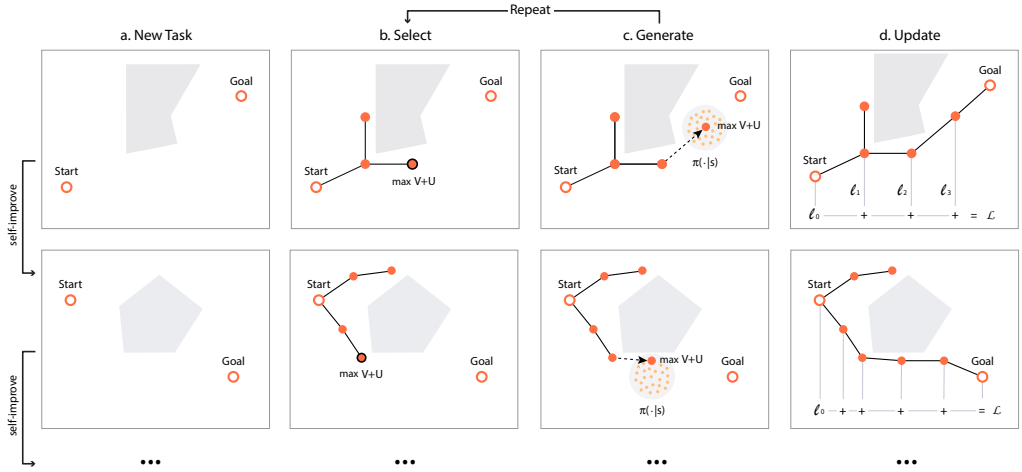


Figure 5: Illustration for the Meta Self-Improving Learning process for NEXT. Every row stands for one epoch in Algorithm 3. In each epoch, a new planning task is generated randomly, as in column (a). The algorithm takes the task and executes NEXT for planning. In every step of the planning, as in column (b)&(c), the search tree grows using the learned $\tilde{V}^*$ and $\tilde{\pi}^*$ for guidance. Such operation is repeated until either a solution is found or the maximum iteration is achieved. Eventually, when the planning path is obtained, the $\{\tilde{V}^*, \tilde{\pi}^*\}$ will be updated based on the successful path as in column (d) and the whole epoch ends. The value function and the policy promote the performance of the planner; meanwhile, the planner will generate samples to lift the value function and policy accuracy. Such planning and learning iteration is continued interactively.

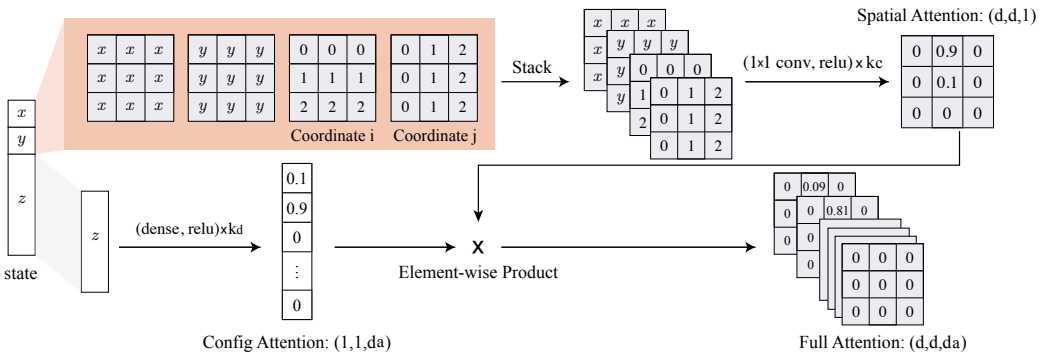


Figure 6: Attention-based configuration space embedding module. We use $s^w = (x, y)$ to denote the workspace coordinates of the input state s, and $s^h = \mathbf{z}$ to denote the rest dimensions of s. The upper part is inspired by the design of the CoordConv layer (Liu et al., 2018) to learn the spatial attention better. At the preparation phase, we stacked four matrices with the same shape as the workspace map as input. The first two channels are filled with x and y respectively, and the last two channels are filled with the i and j grid coordinates. Then we apply several 1×1 convolution layers to the input and obtain the spatial attention. The bottom part learns the higher dimensional configuration attention via several dense layers from $\mathbf{z}$. Finally, the full attention $\mu(s)$ is computed via an element-wise outer-product of the two sub-attentions.

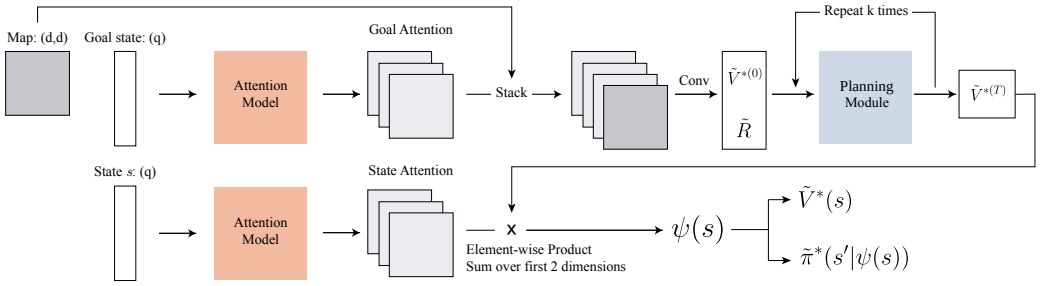


Figure 7: Overall model architecture. Both the initial state and the goal state are embedded with our attention-based embedding module. Then embedding of the goal state is concatenated with the task map to produce $\tilde{V}^{*(0)}$ and $\tilde{R}$ as the input to the planning module. Finally, the output of the value iteration network is aggregated with the embedding of the initial state s to produce feature $\psi(s)$ for defining $\tilde{V}^*$ and $\tilde{\pi}^*$.

D Experiment Results

D.1 Details of Quantitative Evaluation

More detailed results are shown in Table 1, 2, 3, including learning-based and non-learning-based ones, on the last 1000 tasks in each experiment. We normalized the number of collision checks and the cost of paths based on the solution of RRT*. The success rate result is not normalized.

The best planners in each experiment are in **bold**. Our algorithm, *i.e.*, NEXT-KS, achieves competitive or even better results with others using 20 times fewer samples. In all tasks, the proposed algorithms, including NEXT-KS and NEXT-GP outperform the current state-of-the-art planning algorithm with large margins.

	NEXT								#samples-10k			
	KS	GP	GPPN	RRT*	BIT*	BFS	CVAE	Reject	RRT*	BIT*	CVAE	Reject
2D	**0.988**	0.981	0.967	0.735	0.728	0.185	0.535	0.720	0.996	**1.000**	0.996	0.997
3D	**0.943**	0.841	-	0.490	0.036	0.121	0.114	0.498	0.937	0.758	0.654	**0.940**
5D	**0.883**	0.768	-	0.455	0.000	0.030	0.476	0.444	0.858	0.385	0.837	**0.859**

Table 1: Success rate results. The higher the better. The NEXT-KS achieves the best results using only one-twentieth of samples.

	NEXT							#samples-10k			
	KS	GP	RRT*	BIT*	BFS	CVAE	Reject	RRT*	BIT*	CVAE	Reject
2D	**0.177**	0.243	1.000	1.154	9.247	1.983	1.011	4.635	**1.030**	14.308	5.008
3D	**0.694**	1.334	1.000	1.033	7.292	2.162	0.988	5.023	**1.638**	57.430	5.318
5D	**0.888**	1.520	1.000	1.004	5.758	1.188	0.997	8.525	**1.888**	24.373	8.329

Table 2: Average number of collision checks results. The lower the better. The score is normalized based on the solution of RRT*. The NEXT-KS performs the best.

	NEXT								#samples-10k			
	KS	GP	GPPN	RRT*	BIT*	BFS	CVAE	Reject	RRT*	BIT*	CVAE	Reject
2D	**0.172**	0.193	0.272	1.000	1.050	2.811	1.649	1.050	0.167	0.188	**0.154**	0.165
3D	**0.116**	0.315	-	1.000	1.886	1.720	1.734	0.984	0.129	0.480	0.680	**0.123**
5D	**0.215**	0.426	-	1.000	1.835	1.780	0.961	1.020	0.261	1.128	0.299	**0.259**

Table 3: Average cost of paths. The lower the better. The score is normalized based on the solution of RRT*. The NEXT-KS achieves the best solutions.

We demonstrated the performance improvement curves for 2D workspace planning, 3D rigid body navigation Figure 8. As we can see, similar to the performances on 5D 3-link snake planning tasks in Figure 1, in these tasks, the NEXT-KS and NEXT-GP improve the performances along with more and more experiences collected, justified the self-improvement ability by learning $\tilde{V}^*$ and $\tilde{\pi}^*$.

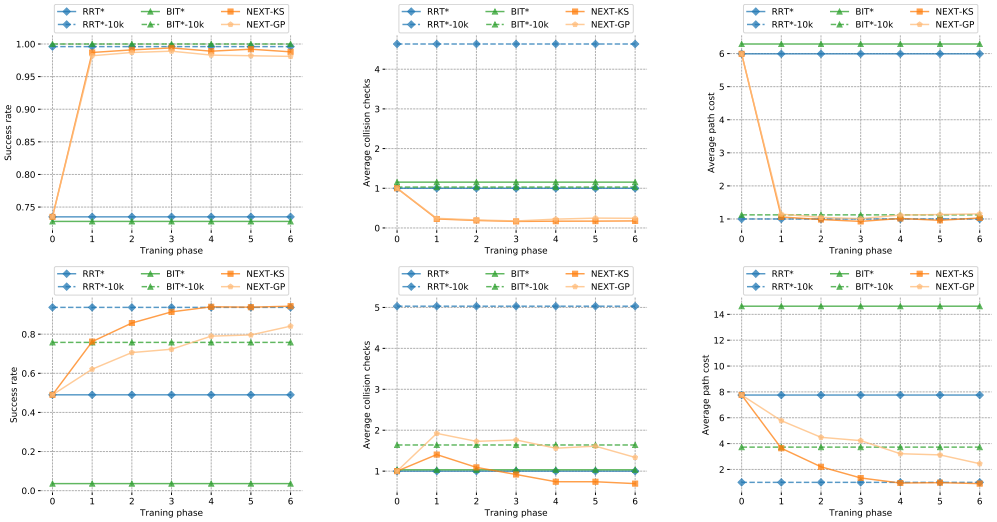


Figure 8: The first and second rows display the improvement curves of our algorithms on all 3000 tasks of the 2D workspace planning and 3D rigid body navigation problems. We compare our algorithms with RRT* and BIT*. Three columns correspond to the success rate, the average collision checks, and the average cost of the solution paths for each algorithm.

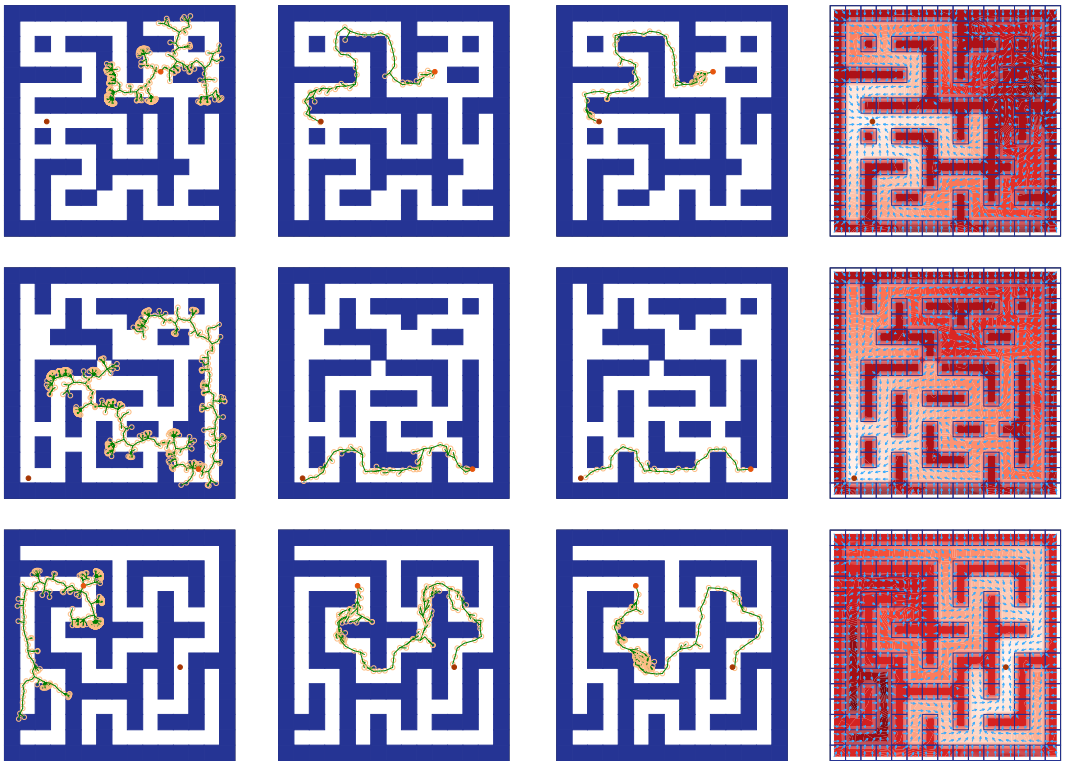
D.2 Solution Path Example

Here we provide an example of the solution paths found by NEXT on all three types of tasks.

Figure 9: The solution path produced by NEXT in a workspace planning task, rigid body navigation task, 3-link snake task from left to right. The orange dot and the brown dot are starting and goal locations, respectively.

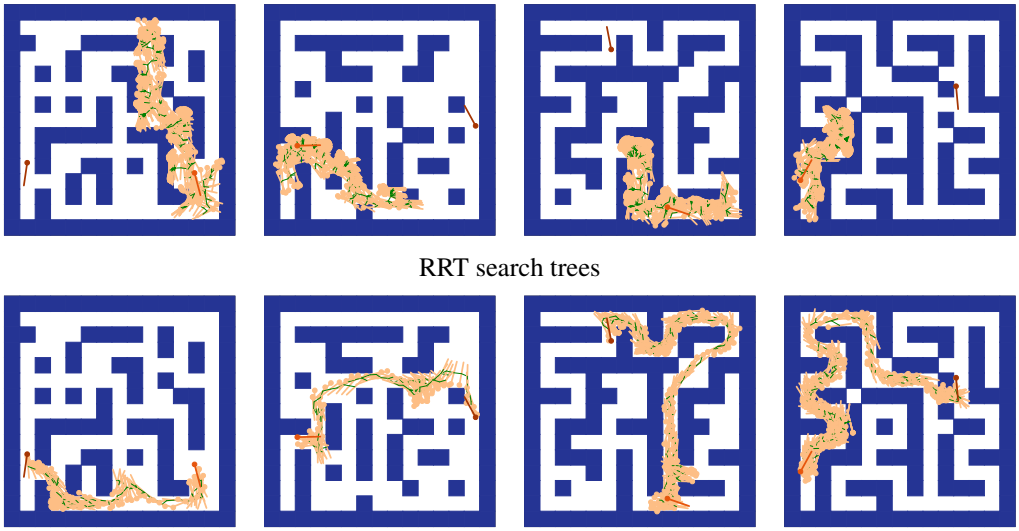
D.3 Search Trees Comparison

We illustrate the search trees generated by RRT* and the proposed NEXT algorithms with 500 samples in Figure 10, Figure 11, and Figure 12 on several workspace planning, rigid body navigation, and 3-link snake planning tasks, respectively. Comparing to the search trees generated by RRT* side by side, we can clearly see the advantages and the efficiency of the proposed NEXT algorithms. In all the tasks, even in 2D workspace planning tasks, the RRT* indeed randomly searches without realizing the goals, and thus cannot complete the missions, while the NEXT algorithms search towards the goals with the guidance from $\tilde{V}^*$ and $\tilde{\pi}^*$, therefore, successfully provides high-quality solutions.



(a) RRT* search tree (b) NEXT-KS search tree (c) NEXT-GP search tree (d) learned $\tilde{V}^*$ and $\tilde{\pi}^*$

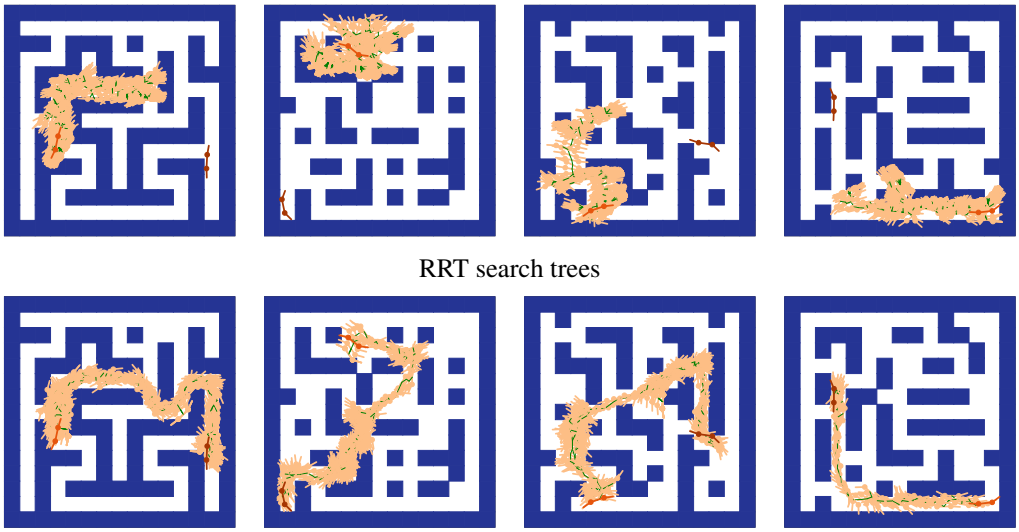
Figure 10: Column (a) to (c) are the search trees produced by the RRT, NEXT-KS, and NEXT-GP on the same workspace planning task. The learned $\tilde{V}^*$ and $\tilde{\pi}^*$ from NEXT-KS are plotted in column (d). In the figures, obstacles are colored in deep blue, the starting and goal locations are denoted by orange and brown dots, respectively. In column (a) to (c), samples are represented with hollow yellow circles, and edges are colored in green. In column (d), the level of redness denotes the value of the cost-to-go estimate $\tilde{V}^*$, and the light blue arrows point from a given state s to the center of the proposal distribution $\tilde{\pi}^*(s'|s, U)$. We set the maximum number of samples to be 500.



RRT search trees

NEXT-KS search trees

Figure 11: Each column corresponds to one example from the rigid body navigation problem. The top and the bottom rows are the search trees produced by the RRT and NEXT-KS, respectively. In the figures, obstacles are colored in deep blue, and the rigid bodies are represented with matchsticks. The samples, starting states, and goal states are denoted by yellow, orange, and brown matchsticks, respectively. Edges are colored in green. We set the maximum number of samples to be 500.



RRT search trees

NEXT-KS search trees

Figure 12: Each column corresponds to one example from the 3-link snake problem. The top and the bottom rows are the search trees produced by the RRT and NEXT-KS, respectively. In the figures, obstacles are colored in deep blue, and the rigid bodies are represented with matchsticks. The samples, starting states, and goal states are denoted by yellow, orange, and brown matchsticks, respectively. Edges are colored in green. We set the maximum number of samples to be 500.