2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Parallel Heuristic Search as Inference for Actor-Critic Reinforcement Learning Models

Hanlan Yang[*1,2], **Itamar Mishani**[*1], **Luca Pivetti**[1,3], **Zachary Kingston**[2], **and Maxim Likhachev**[1]
[1]Carnegie Mellon University [2]Purdue University [3]University of Milano-Bicocca
∗ Equal Contribution

***Abstract*— Actor-critic models are a class of model-free deep reinforcement learning (RL) algorithms that have demonstrated effectiveness across various robot learning tasks. While considerable research has focused on improving training stability and data sampling efficiency, most deployment strategies have remained relatively simplistic, typically relying on direct actor policy rollouts. In contrast, we propose PACHS (*P*arallel *A*ctor-*C*ritic *H*euristic *S*earch), an efficient parallel best-first search algorithm for inference that leverages both components of the actor-critic architecture: the actor network generates actions, while the critic network provides cost-to-go estimates to guide the search. Two levels of parallelism are employed within the search—actions and cost-to-go estimates are generated in batches by the actor and critic networks respectively, and graph expansion is distributed across multiple threads. We demonstrate the effectiveness of our approach in robotic manipulation tasks, including collision-free motion planning and contact-rich interactions such as non-prehensile pushing. Visit p-achs.github.io for demonstrations and examples.**

I. Introduction

Reinforcement Learning (RL) has become a central paradigm for robot control, offering a way to learn behaviors that are difficult to manually specify through cost functions, dynamics models, or action abstractions. Despite this strength, RL methods still face significant challenges in generalization during inference and in solving complex problems that are common in robotic manipulation. Much of the prior work has focused on improving model architectures and training strategies, but comparatively little attention has been given to inference strategies. As a result, RL models are typically deployed as one-step predictors during execution, lacking the ability to perform multi-step forward reasoning or backtracking.

Consider the example shown in Fig. 1, where the objective is to push the T-shaped object to a target pose. While specifying and modeling this task with classical model-based planning algorithms is challenging due to complex contact dynamics, an RL model can learn non-prehensile manipulation behaviors through a simple reward function. Such models perform well in relatively simple settings, for example, when the goal is fixed and the environment is uncluttered. However, as problems become more complex and require inference generality (e.g., when multiple objects must be manipulated sequentially, or when goals depend on environmental constraints), RL often fails to learn sufficiently robust policies. In these cases, search methods provide an alternative: they can explicitly plan multiple steps into the future, enabling exploration of the state space through branching and backtracking.

Fig. 1. PACHS constructs an implicit lattice graph to plan a trajectory for pushing the T-shaped object to a target pose.

The goal of this work is to combine the advantages of traditional planning algorithms with those of learned models. This paper integrates off-the-shelf Soft Actor-Critic (SAC) models, trained to control a robotic arm, within a best-first heuristic search framework. In this integration, the actor network proposes candidate actions (i.e., edges) while the critic network serves as a learned heuristic function. The reinforcement learning approach underlying SAC enables the model to capture complex interactions—such as robot–object interactions, system dynamics, and environmental constraints—that are often difficult to encode using hand-crafted action spaces, heuristics, and cost functions. Furthermore, to improve efficiency, we parallelize the exploration phase of the search process by assigning separate threads to each expansion step and evaluating edges in parallel.

We evaluate the proposed approach on two applications. The first scenario involves a shelf environment where the robotic arm must move its end-effector to a desired spatial position along a collision-free path. The second scenario considers a PushT task, where the arm must interact with a T-shaped object to push it to a specific position and orientation. Here, we trained the model in an obstacle-free environment and then evaluated the algorithm in environments with added obstacles to study generalizability. The results demonstrate substantial performance improvements, enabling the effective use of imperfectly trained models in complex scenarios through search.

Our contributions are:

- A novel algorithm, PACHS, enabling best-first search planning with learned actor-critic RL models.
- Multi-layered parallelization strategies (CPU thread-level and GPU batch-level) that achieve significant computational efficiency gains.
- Experimental evaluations, demonstrating how PACHS improves the deployment and generalization of RL models by enabling robust performance in complex environments.

II. Related Work

Our approach combines reinforcement learning (RL) with heuristic search algorithms, building on prior work in both domains. Actor-critic methods like SAC have proven effective for learning complex robotic behaviors through the combination of policy-based and value-based techniques [1]. However, these methods lack the structural exploration capabilities of search algorithms when deployed. Conversely, classical search methods like A* [2] provide robust planning guarantees but struggle with continuous action spaces and complex robotic dynamics that are difficult to capture with hand-crafted heuristics. Zero-shot generalization approaches in reinforcement learning have been widely studied (see [3] for an overview); however, compared to common approaches, this paper proposes an algorithm that does not require retraining and can be used without modifying the policy.

A. Combining Learning and Search

The integration of learning and search has been most prominently demonstrated in domains where simulating roll-outs is trivial, such as in game-playing domains. AlphaGo [4] combines a policy network, value network, and Monte Carlo Tree Search (MCTS) in an iterative process: the policy network proposes promising moves, the value network estimates expected outcomes of future board states, and MCTS performs lookahead search to refine decision-making. While conceptually similar to our approach, several key differences distinguish our work: AlphaGo operates in discrete environments, significantly reducing the complexity of state expansion and search; computations are processed sequentially, making each decision computationally expensive; and the policy and value networks are trained independently, requiring separate training procedures.

Learning heuristic functions has been explored extensively, particularly for domains where designing effective heuristics is challenging. For instance, DeepCubeA [5] uses a deep neural network to learn heuristic functions for solving Rubik's Cube, which are then integrated into an A*-like search algorithm. Similar approaches have been applied to various combinatorial problems [6], but these methods are typically restricted to value-based approaches and discrete action spaces. Recent work has developed new priority functions for search that integrate learned components, such as local heuristics [7]. Other learned heuristics expansion methods, such as [8], focus on learning to guide node expansion, using learned models to prune unfavorable nodes during search.

More recently, several works have explored combining policy-based methods with search. AlphaZero [9] extends the AlphaGo framework to learn both policy and value functions from scratch through self-play. MuZero [10] further advances this by learning a model of the environment dynamics. However, these approaches remain focused on discrete domains and sequential computation, limiting their applicability to continuous robotic control problems where parallel execution and real-time constraints are critical.

Our work addresses these limitations by integrating SAC, which naturally handles continuous action spaces, with parallel A*-like search. Unlike prior approaches that require independent training of policy and value networks, we leverage the joint training inherent in actor-critic methods. Furthermore, our parallel search framework enables efficient exploration in continuous domains, making the approach practical for real-time robotic applications.

B. Heuristic Search and Parallelization

Advances in computing hardware, such as the increasing number of cores in modern processors, have driven significant efforts to parallelize best-first search algorithms, which inherently require extensive exploration through node expansion and edge evaluation. In robotic domains, edge evaluations are particularly expensive, involving collision checking, physics simulation, and complex dynamics computations, making parallelization essential for practical performance [11] [12].

Search-based methods, like A* and beam search, can leverage parallelization by generating successors concurrently during state expansion, though their scalability is often constrained by the domain's branching factor. For instance, Beam search [13], which traverses a graph layer by layer, selects a fixed number of nodes (the beam width) at each layer for expansion. This expansion work can be distributed across multiple workers. However, parallel Beam search remains inherently synchronous because selecting nodes for expansion requires constructing the complete candidate list beforehand. Monte Carlo Tree Search (MCTS) has also been explored for parallelization. PMBS [14], for example, parallelizes the simulation rollout process in batches. However, MCTS requires multiple cycles of simulation roll-out and backpropagation to update the heuristic for a single-step decision-making, which makes it computationally intensive and limits the planning horizon given the same computation budget when compared to other search methods.

Heuristic best-first search algorithms, particularly A* and its variants, are non-trivial to parallelize due to their sequential expansion nature and the need to maintain theoretical guarantees. Early approaches like Parallel A* [15] use a centralized *OPEN* list (priority queue) and assign the current best states to available CPU cores. PRA* [16] and HDA* [17] improve upon this by giving each CPU core its own *OPEN* list, with HDA* adding an asynchronous message passing system to reduce communication blocking. GPU implementations follow similar principles, maintaining multiple parallel *OPEN* lists [18], [19]. However, all these approaches must allow states to be re-expanded to guarantee optimality, and the number of re-expansions can grow exponentially with parallelization degree, especially with weighted heuristics, causing significant performance degradation [20], [21].

More recent work addresses these limitations through alternative parallelization strategies. PA*SE [21] waives the re-expansion requirement by estimating state independence using admissible pairwise heuristics. The ePA*SE family [20], [22], [23] further improves efficiency by decoupling state expansion from edge evaluation, particularly beneficial for robotic domains with expensive edge evaluations (collision checking, simulation queries). However, these methods still face challenges in continuous robotic domains where physics simulation and contact-rich interactions cannot be easily delegated or approximated. While parallelization advances improve computational efficiency, they still rely on hand-crafted heuristics, action spaces, and cost functions that may not capture the complex dynamics of manipulation tasks.

III. Preliminaries

This section establishes the theoretical foundation for integrating reinforcement learning with heuristic search. We begin by formulating continuous robotic control as a Markov Decision Process (MDP), then review actor-critic methods and heuristic search algorithms. Finally, we demonstrate the fundamental connection between these paradigms that enables our integration approach.

A. Problem Formulation

We consider robotic manipulation tasks modeled as Markov Decision Processes $(\mathcal{S}, \mathcal{A}, f, R, \gamma)$, where $\mathcal{S} \subset \mathbb{R}^d$ and $\mathcal{A} \subset \mathbb{R}^m$ represent continuous state and action spaces respectively, $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ defines the system dynamics, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. Given a trained Soft Actor-Critic model consisting of an actor network $\pi_\theta : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ (where $\mathcal{P}(\mathcal{A})$ denotes probability distributions over actions) and critic network $Q_\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and given an initial state $s_0 \in \mathcal{S}$ and goal region $\mathcal{G} \subseteq \mathcal{S}$, we seek to find a feasible trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$ that reaches a goal state while minimizing accumulated cost. Formally, we aim to solve:

$$\min_{\tau} \quad \sum_{t=0}^{T-1} c(s_t, a_t) \tag{1}$$

$$\text{s.t.} \quad s_T \in \mathcal{G} \tag{2}$$

$$s_t \in \mathcal{S}_{\text{valid}} \quad \forall t \in \{0, 1, \ldots, T\} \tag{3}$$

$$s_{t+1} = f(s_t, a_t) \quad \forall t \in \{0, 1, \ldots, T-1\} \tag{4}$$

where $c(s, a) = -R(s, a)$ converts rewards to costs and $\mathcal{S}_{\text{valid}} \subseteq \mathcal{S}$ denotes the constraint-satisfying state space (e.g., collision-free configurations).

To solve this optimization problem, we leverage reinforcement learning models that have learned both action selection strategies and value estimation from interaction data. Specifically, we build upon actor-critic architectures that provide both components needed for effective search: action generation and state evaluation.

B. Actor-Critic RL

Actor-critic methods [24], [25] address the limitations of pure policy gradient and value-based approaches by maintaining two complementary function approximators: an actor network π_θ that parameterizes the policy, and a critic network Q_ϕ that estimates the state-action value (Q-) function

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s, a_0 = a \right],$$

representing the expected cumulative reward when taking action a in state s and following policy π thereafter. This dual architecture enables stable learning through policy iteration, where the critic evaluates the current policy, and the actor improves the policy using gradients derived from the critic's evaluations.

Standard deployment of actor-critic models utilizes only the actor network for action selection, effectively discarding the critic's learned value assessments once training concludes [1]. This represents a significant underutilization of the available learned knowledge, as the critic contains valuable information about state quality and expected future returns that could inform more sophisticated decision-making during execution.

To fully utilize both components of the actor-critic model, we integrate them within a systematic search framework. Best-first search algorithms provide the structured exploration needed to solve the optimization problem while allowing us to leverage learned knowledge from both networks.

C. Best-first Search

Best-first search algorithms, particularly A* and its variants [20], [26], [27], provide algorithmic approaches to solving the optimization problem defined in Equations (1)-(4). These algorithms construct an implicit graph $G = (V, E)$ where vertices $v \in V$ represent states $s \in \mathcal{S}$ and edges $e \in E$ represent state-action pairs (s, a) that induce transitions $s' = f(s, a)$.

The A* algorithm maintains a priority queue of discovered states, ordered by an evaluation function:

$$f(s) = g(s) + h(s),$$

where $g(s)$ estimates the optimal cost-to-come from the initial state s_0 to state s:

$$g(s) = \min_{\tau : s_0 \rightsquigarrow s} \sum_{t=0}^{|\tau|-1} c(s_t, a_t),$$

and $h(s)$ is a heuristic estimating the cost-to-go from s to any goal state:

$$h(s) \leq \min_{s_g \in \mathcal{G}} \min_{\tau : s \rightsquigarrow s_g} \sum_{t=0}^{|\tau|-1} c(s_t, a_t). \tag{5}$$

At each iteration, A* selects the state with minimum f-value for expansion, generating successor states through available actions and updating their cost-to-come estimates. This process continues until a goal state is reached, guaranteeing optimal solutions when the heuristic is admissible.

Connection to Reinforcement Learning: Our approach is based on the mathematical equivalence between search heuristics and RL value functions. Specifically, the heuristic function $h(s)$ in Equation (5) estimates a quantity similar

to the state value function of RL when the rewards are formulated as negative values.

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}R(s_t,a_t) \mid s_0 = s\right],$$

when we set $c(s,a) = -R(s,a)$ (converting rewards to costs). This equivalence allows us to use the critic network $Q_\phi(s,a)$ as a learned heuristic, while the actor network $\pi_\theta(a|s)$ can guide action selection during state expansion, replacing hand-crafted action spaces with learned action distributions.

D. ePA*SE

One particularly important algorithm that motivated this work is ePA*SE [20]: a parallel version of edge-A* that addresses expensive edge evaluation through two key innovations. First, it decouples state expansion from edge evaluation by maintaining an open list of edges instead of states, using edge placeholders to separate successor generation from edge evaluation. Second, it enables parallelization of edge evaluations where each computational thread evaluates a single edge at a time, distributing the computationally expensive workload across threads for greater efficiency. The algorithm selects the edge $e = (s,a)$ for expansion with the smallest priority value $f(e) = g(s) + w \cdot h(s)$, where $g(s)$ is the cost-to-come, $h(s)$ is the heuristic estimate of the source state s, and w is the heuristic weight, which controls the optimality bound and the greediness of the search. During the expansion of edge e, the algorithm evaluates the edge to determine the resulting successor state s', then generates all outgoing edges from s' and inserts them into the open list.

IV. Algorithmic Approach

To address the limitations of RL deployment and traditional best-first search, we present PACHS, a best-first heuristic search framework that enables multi-step reasoning during RL inference. Our approach capitalizes on the natural correspondence between RL and search concepts: the actor network generates candidate actions (edges) during state expansion, while the critic network provides learned heuristics to guide the search process. This integration leverages the knowledge acquired during RL training while extending its capabilities and performance during inference.

Enabling effective search in robotic domains requires addressing significant computational challenges. Neural network inference for both actor and critic models is computationally expensive, and robotic tasks often involve costly edge evaluations through physics simulation, collision checking, and dynamics computation. Without sophisticated parallelization strategies, search-based approaches can be prohibitively slow for real-time deployment. Therefore, PACHS incorporates multi-level parallelization to make search with learned modules practical for robotics (Fig. 2).

Inspired by ePA*SE, PACHS decouples edge evaluation from state expansion to allow for a finer-grained parallelization. PACHS uses the actor to generate actions in continuous space and a critic to evaluate the priority of generated actions. Since the actor and critic are trained with a physics simulator, they implicitly learn and embed the dynamics of the environment, waiving the engineering effort to design handcrafted heuristic functions and define discrete action spaces, resulting in learned, dynamically-generated motion primitives.

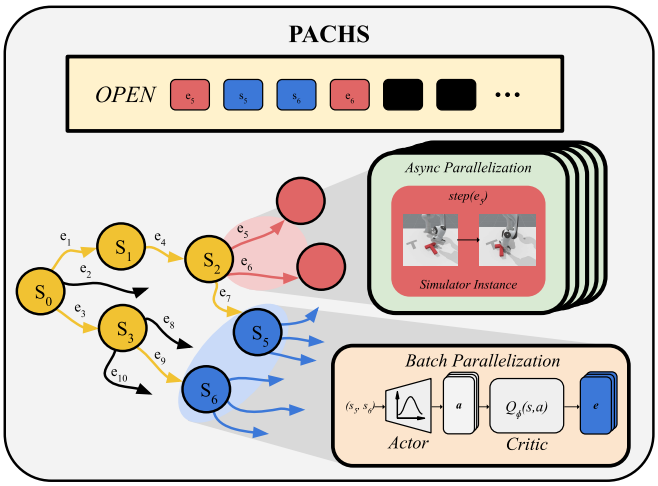

Fig. 2. PACHS multi-level parallelization. Yellow elements indicate completed expansions. The *OPEN* list stores edge candidates ordered by priority. Red edges (e_5, e_6) are evaluated in parallel across threads, while blue states (s_5, s_6) generate actions and heuristics in batches via the actor and critic networks.

PACHS addresses a key limitation in edge prioritization that arises when edge costs are unavailable until full evaluation. In domains using physics simulation to evaluate action outcomes, ePA*SE does not evaluate edges before selection from the *OPEN* list, and its edge priorities are of the form $f(e) = f((s,a)) = g(s) + w \cdot h(s)$, which lack action-specific information. This causes all outgoing edges from the same parent state to have identical priorities, making edge selection depend solely on tie-breaking strategies. PACHS overcomes this limitation by incorporating learned Q-values (the critic outputs) into the priority function. The edge priority in PACHS is $f(e) = g(s) + w \cdot Q_\phi(s,a) \approx g(s) + w \cdot (c((s,a)) + h(s'|a))$, where $Q_\phi(s,a)$ approximates the expected cumulative cost. This formulation leverages the critic's learned cost estimation to prioritize promising actions during search, enabling more informed edge selection while maintaining the computational benefits of lazy state and edge generation.

Algorithms 1 & 2 detail PACHS's operation. Algorithm 1 handles the main planning loop on the primary thread, while Algorithm 2 manages parallel edge expansion across worker threads.

Main Planning Loop (Alg. 1): The algorithm maintains an *OPEN* list of edges prioritized by their f-values. States are represented as edges by pairing them with dummy actions $\mathbf{a^d}$ (Line 12). A dummy edge $(\mathbf{s}, \mathbf{a^d})$ serves as a placeholder that indicates state $\mathbf{s}$ is ready for expansion–when selected, it triggers the generation of real outgoing actions from that state rather than executing a specific action. This design enables unified treatment of states and actions within the edge-based search framework. The main loop selects the highest priority edge (Line 23), checks for goal satisfaction (Line 24), and assigns edges to available worker threads for parallel processing. When a goal is reached, the solution path

is reconstructed by backtracking parent pointers from the goal state to the start state.

Worker Thread Operations (Alg. 2): Worker threads handle two types of edges. **Dummy edges** trigger state expansion: the actor generates action batches (Line 9) from a continuous action distribution, the critic computes the corresponding Q-values (Line 10), and real edges are inserted into the *OPEN* list with priorities $f((\mathbf{s}, \mathbf{a_i})) = g(\mathbf{s}) + w \cdot \mathbf{q_i}$ (Line 13). The state is then marked as *CLOSED* (Line 15). **Real edges** undergo evaluation (Line 18) to compute successor states and edge costs through physics simulation. When improvements are found, the algorithm updates the successor's g-value, parent pointer (Line 22), and inserts a dummy edge for the successor with updated priority (Line 24).

Parallelization: Critical data structure updates use locks to prevent race conditions, while expensive operations (neural network inference and physics simulation) run lock-free for maximum parallel efficiency.

A. Real-time Adaptation

To deploy PACHS in real-world scenarios involving robot–object interactions, the system must operate in a real-time, closed-loop manner. We therefore implement a real-time version of PACHS. Given a time budget for a single planning query (e.g., 3 seconds), the robot observes the current environment state, plans within the allotted time, and reconstructs the best path it has found. The algorithm may not always find a complete path to the goal within this limit. If the budget expires before a path to the goal is found, the algorithm selects the edge with the best f-value from the open list, reconstructs a partial path from the current state to this edge, and executes the first few actions of that path. This process repeats—plan, execute, observe—until the goal is reached.

Algorithm 1 PACHS: Planning Loop

1: **INPUT:**
2: π: Stochastic Actor Policy
3: $\mathcal{Q}_\phi$: Q-value Critic Network
4: $\mathbf{s}_0$: start state
5: $\mathcal{G}$: goal condition
6: N_t: number of threads
7: **OUTPUT:**
8: $\mathcal{P}$: Path from start to goal
9: **procedure** PLAN
10: *OPEN* = $\emptyset$
11: $terminate$ = False
12: insert $(\mathbf{s}_0, \mathbf{a^d})$ in *OPEN* ▷ Dummy edge from $\mathbf{s}_0$
13: LOCK
14: **while not** $terminate$ **do**
15: **if** *OPEN* $= \emptyset$ **then**
16: **if** NOWIP **then** ▷ None of the threads evaluate edges
17: $terminate$ = True
18: UNLOCK
19: **return** $\emptyset$ ▷ Exhausted open list, no solution found
20: **else**
21: UNLOCK
22: wait until *OPEN* change ▷ Open list is empty, wait until new edge is added to the open list
23: pop the min edge $(\mathbf{s}, \mathbf{a})$ from *OPEN*
24: **if** $\mathcal{G}(\mathbf{s})$ = True **then** ▷ $\mathbf{s}$ satisfy the goal condition
25: $terminate$ = True
26: UNLOCK
27: **return** $\mathcal{P}$ = BACKTRACK($\mathbf{s}$) ▷ Reconstruct the path
28: UNLOCK
29: **while** $(\mathbf{s}, \mathbf{a})$ has not been assigned a thread **do**
30: **for** $i = 1 : N_t$ **do**
31: **if** thread i is available **then**
32: **if** thread i has not been spawned **then**
33: Spawn EDGEEXPANDTHREAD(i)
34: Assign $(\mathbf{s}, \mathbf{a})$ to thread i ▷ Asynchronous operation
35: LOCK
36: UNLOCK
37: **procedure** NOWIP
38: **for** t in SpawnedThreads **do**
39: **if** t is Working **then**
40: **return** False
41: **return** True

Algorithm 2 PACHS: Expansion

1: **procedure** EXPANDEDGETHREAD(i)
2: **while not** $terminate$ **do**
3: **if** thread i has been assigned an edge $(\mathbf{s}, \mathbf{a})$ **then**
4: **if** $\mathbf{a} = \mathbf{a^d}$ **then**
5: EXPANDSTATE($\mathbf{s}$)
6: **else**
7: EXPANDEDGE($(\mathbf{s}, \mathbf{a})$)
8: **procedure** EXPANDSTATE($\mathbf{s}$)
9: $\tilde{\mathbf{a}} = \pi(\mathbf{s})$ ▷ Sample a batch of actions
10: $\tilde{\mathbf{q}} = \mathcal{Q}^\phi(\mathbf{s}, \tilde{\mathbf{a}})$ ▷ Compute the Q-value in a batch
11: LOCK ▷ Expensive neural network operations are non-blocking
12: **for** $\mathbf{a_i}$ in $\tilde{\mathbf{a}}$ **do**
13: $f((\mathbf{s}, \mathbf{a_i})) = g(\mathbf{s}) + w * \mathbf{q_i}$ ▷ Each edge has a different priority based on the estimation from critic
14: insert $(\mathbf{s}, \mathbf{a_i})$ in *OPEN* with $f((\mathbf{s}, \mathbf{a_i}))$
15: insert $\mathbf{s}$ in *CLOSED*
16: UNLOCK
17: **procedure** EXPANDEDGE($(\mathbf{s}, \mathbf{a})$)
18: $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) =$ EVALUATE($(\mathbf{s}, \mathbf{a})$)
19: LOCK
20: **if** $\mathbf{s}' \notin$ *CLOSED* and $g(\mathbf{s}') > g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$ **then**
21: $g(\mathbf{s}') = g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$
22: $\mathbf{s}'$.parent = $\mathbf{s}$.id
23: $f((\mathbf{s}', \mathbf{a^d})) = g(\mathbf{s}') + w * (\mathbf{q} - c((\mathbf{s}, \mathbf{a})))$ ▷ Reusing Q-value of the edge to calculate heuristic for the successor state
24: insert/update $(\mathbf{s}', \mathbf{a^d})$ in *OPEN* with $f((\mathbf{s}', \mathbf{a^d}))$ ▷ The dummy action indicates the edge is a representation of a state
25: UNLOCK

V. EXPERIMENTS

We evaluated PACHS using a 7-DoF Franka Panda arm in four different tasks across two domains as shown in Fig. 3. The first domain, *Panda-Shelf*, is collision-free motion planning in a shelf environment, where the planner must find a feasible path from a home configuration to a desired end-effector (EE) position within the shelf's reachable workspace. The second domain involves Push T tasks with three variants: *PushT-Fixed*, *PushT-Rand*, and *PushT-Obs*. All tasks begin with the T-shaped object in a random initial pose. In *PushT-Fixed* and *PushT-Obs*, the robot must push the object to a fixed target pose, while *PushT-Rand* requires pushing to a random goal pose. The *PushT-Obs* variant additionally includes static obstacles in the environment. We trained SAC models for the *Panda-Shelf*, *PushT-Fixed*, and *PushT-Rand* tasks using Maniskill [28]. To evaluate generalization capabilities, we applied the *PushT-Fixed* models to the *PushT-Obs* task without retraining. Training details and model specifications are provided in Sections V-B and V-C. All actors are trained to output a 7-DoF delta joint position control for the Panda arm. All models were implemented in PyTorch [29], while search algorithms were implemented in C++. Experiments were conducted on a system equipped

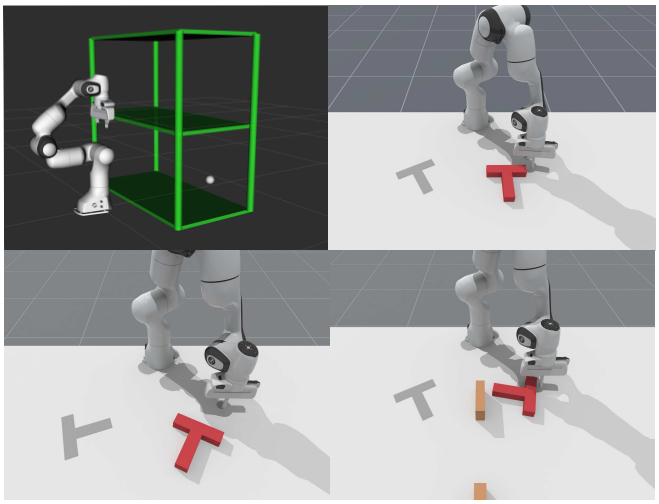

Fig. 3. Simulated environments used in our experiments. Upper left: *Panda-Shelf*—collision-free motion planning to a random end-effector target. Upper right: *PushT-Fixed*—push a T-shaped object to a fixed goal pose. Bottom left: *PushT-Rand*—push T from a random start pose to a random goal pose. Bottom right: *PushT-Obs*—push T to a fixed goal pose with obstacles.

with an Intel i7-11800H CPU and NVIDIA GeForce RTX 3070 laptop GPU.

A. Baseline Methods

We compare PACHS against four baseline algorithms to assess deployment efficiency and its ability to generalize in inference, without retraining. ***Single Rollout*** represents the standard sequential rollout approach that maintains a single environment and executes the Observe-GetAction-StepEnv cycle until finding a solution or reaching the maximum step limit, which we set equal to the training episode length. ***Parallel Rollout*** is a parallel rollout method that instantiates multiple environments with identical problem instances and performs batch rollouts. These environments diverge after the first rollout step due to the stochastic actor policy. We enforce a total evaluation step budget to ensure fair computational comparisons. When a rollout batch reaches the step limit without finding a solution, episodes restart and continue until either a solution is found or the budget is exhausted. ***Parallel Beam Search*** implements parallel beam search that uses the actor to sample actions, then selects a beam width of states based on Q-values estimated by the critic. We also compare against ***ePA*SE*** specifically for the *Panda-Shelf* environment, where a suitable heuristic function is readily available without extensive engineering effort. In this case, we employ a 3D breadth-first search algorithm to compute the voxel-based distance from the current EE state to the goal state.

B. Panda Shelf Motion Planning

The main purpose of the collision-free motion planning experiment is to test whether PACHS, which utilizes neural networks during search, is efficient enough compared to existing methods. We developed the *Panda-Shelf* environment based on the panda-gym package [30] and trained the SAC model using the stable-baseline3 framework [31]. We used two fully connected layers with 64 neurons for both

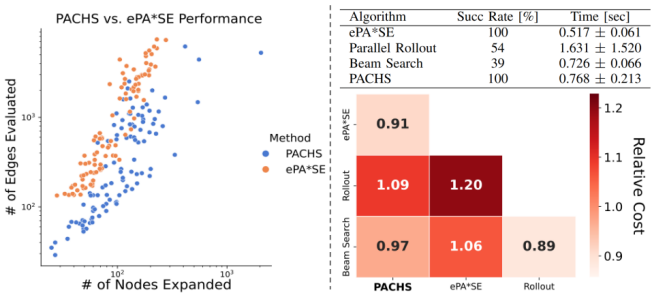

Fig. 4. Collision-free motion planning results in the *Panda-Shelf* environment. Top right: success rate and planning time; only ePA*SE and PACHS consistently solve all instances. Bottom right: cost comparison showing PACHS produces solutions similar to baselines. Left: Critic-guided prioritization reduces the number of evaluated edges.

the actor and the critic. We used a dense reward that is a linear combination of the negated Euclidean distance from the robot's current EE position to the desired end-effector position, and a penalty term that is the minimum distance between the bookshelf and the arm, to encourage motion that avoids collision. We trained the models with 1 million training steps.

We generate 100 problem instances of an EE target position within the shelf which is reachable by the robot. All planners are given a 60 seconds to solve each instance. Since the collision-free motion planning does not require simulating actions, we use a collision checker to evaluate edges. In the case of parallel rollout, the rollout instantly restarts when it leads to a collision. Results are shown in Fig. 4. The cost is the accumulated distance of the end-effector throughout the motion. Cost and time are reported only for instances where the planner found a solution.

We observe that ePA*SE and PACHS achieve 100% success rates, while parallel rollout and beam search show considerably lower performance. Despite the computational overhead from using neural networks, PACHS matches ePA*SE's solution quality and planning time. This efficiency comes from the critic network's guidance. For identical numbers of expanded nodes, PACHS evaluates substantially fewer edges, illustrating how its prioritization reduces computational waste.

C. PushT: Solution Finding Evaluation

The objective of this experiment is to assess the ability of PACHS, compared to baseline approaches, to find solutions to manipulation tasks that require interactions with the environment, and its ability to generalize during inference without network finetuning. The PushT environment tasks are based on the PushT task of ManiSkill. The 7-DoF Panda arm must manipulate a T-shaped object by moving it from an initial configuration to a target configuration on a planar surface (gray projection as shown in Fig. 3). The state of the object is defined by the position of its center of mass and its orientation. Task success is achieved when at least 90% of the goal configuration's shadow projection on the table is covered by the object.

For the easier problem, *PushT-Fixed*, we trained a 3-layer fully-connected network for both the actor and the critic, with 256 neurons in each layer and a ReLU activation function. For the harder *PushT-Rand* environment, each layer

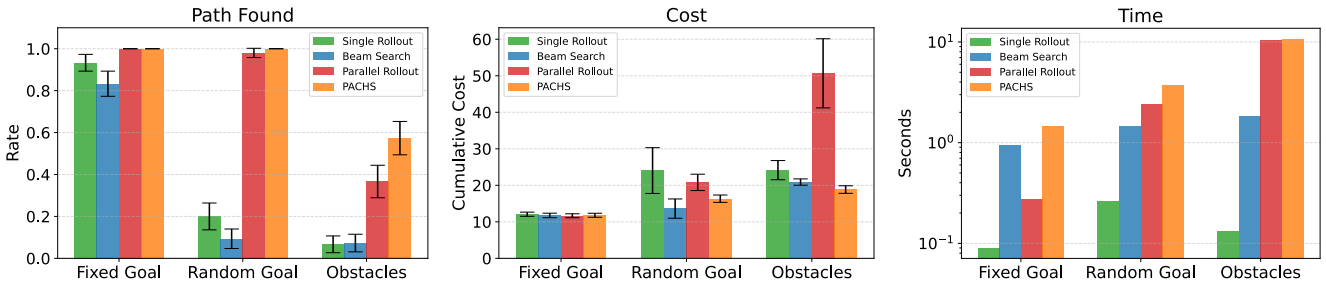

Fig. 5. Planner comparison across PushT tasks (*PushT-Fixed*, *PushT-Rand*, *PushT-Obs*) with a 100k edge-evaluation budget. PACHS achieves higher success rates across all tasks, reaching 100% in fixed and random goal settings and significantly outperforming rollout methods in the obstacle scenario. Planning time and cost are reported for successful runs only. PACHS achieves substantially lower costs compared to baselines.

size is 1024. The SAC model was trained with a negative dense reward formulation composed of three terms: the angular distance between the T-object projected on the table and its goal state, the distance function between the T-object and its goal state, and the distance between the end effector and the T-object (a commonly used reward function for the PushT environment). For all tasks in the PushT domain, we randomly sampled 30 problem instances. These correspond to 30 random initial T poses in *PushT-Fixed* and *PushT-Obs* and 30 pairs of random initial and goal T poses in *PushT-Rand*. The positions of the obstacles in *PushT-Obs* remain the same across different problems. We ran each of the 30 problem instances five times to obtain statistics for the success rate of the algorithm returning a solution. We gave PACHS and parallel rollout a total evaluation budget of 100k. The experimental results are shown in Fig. 5. We observe that PACHS consistently solves more problem instances (Path Found metric). Interestingly, despite the theoretical advantages of best-first search methods, beam search shows the lowest performance across all planners. This highlights that not all search strategies are beneficial for solving such tasks. Notably, PACHS achieves a significantly higher success rate in *PushT-Obs* task, demonstrating the generalization capabilities it can provide to RL models, allowing them to solve tasks beyond their training domain.

Fig. 6 shows success rate versus number of edges evaluated for PACHS and parallel rollout in the *PushT-Obs* task. Each dot represents the percentage of instances solved after

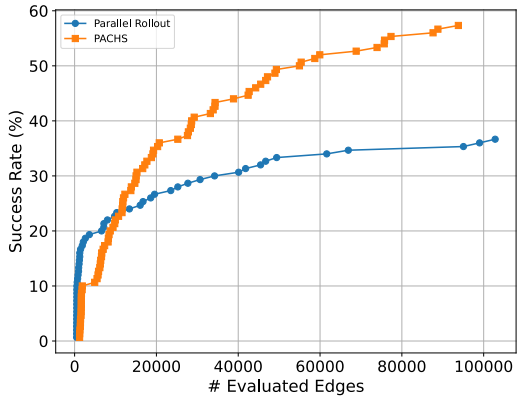

Fig. 6. Success rate vs. # of evaluations for PACHS and parallel rollout in the *PushT-Obs* task. Parallel rollout improves quickly at small budgets but plateaus, while PACHS continues improving with additional evaluations.

x edge evaluations. We observe that parallel rollout's success rate spikes quickly at lower evaluation budgets but plateaus as the edge evaluation budget increases. In contrast, PACHS requires slightly more evaluations to match parallel rollout's early performance, yet demonstrates steady improvement as the evaluation budget grows. This reveals an important insight: randomness provides only limited generalization during deployment, highlighting the advantages of structured exploration through search.

D. PushT: Execution Performance Evaluation

This experiment investigates whether PACHS's planning improvements translate to better closed-loop execution performance–a critical requirement for real-world robotic deployment. Building on Section V-C showing that PACHS finds solutions and improves generalization for RL deployment, we now evaluate whether these benefits extend to dynamic replanning scenarios where the robot must continuously adapt to environmental feedback. In this experiment, we maintain an independent simulator environment to mimic a real-world scenario, which we call the "world" simulator. Then, each algorithm is given a short planning budget to plan a full path or a partial path. After a path is returned, a certain horizon of actions from the plan are executed in the "world" simulator, which then provides the observation for the next planning query. We compare single rollout in the "world" simulator, parallel rollout, and PACHS. If no plan is found within the time budget, PACHS will return a partial path to the best state, described in Sec. IV-A. Similarly, parallel rollout will keep track of the rollout that has the best cumulated reward and will return that path if no rollout leads to a solution.

Similar to Sec. V-C, we run five rounds of evaluation with 20 problem instances. PACHS and parallel rollout are given 3 seconds of planning budget and an action execution horizon of 10 (i.e., after each planning query, 10 actions are executed in the "world" simulator). We limit the process to 30 replanning calls, with successful execution determined by reaching the goal in the "world" simulator.

Fig. 7 shows the success rate of reaching the goal in the "world" simulator over the three PushT tasks. The performance of PACHS is consistently superior compared to both rollout approaches, demonstrating a better sim2sim transfer. Interestingly, even though the parallel rollout achieved an impressive success rate at finding the solution in Fig. 5,

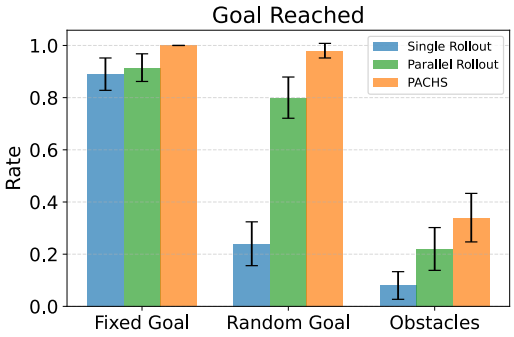

Fig. 7. Comparison of the closed-loop success rate of single rollout, parallel rollout, and PACHS for all PushT tasks. PACHS is consistently better than baselines across different tasks.

its performance drops significantly while PACHS maintains similar performance when executed in a closed-loop manner, indicating that the solutions found by rollout are inconsistent compared to PACHS.

VI. Conclusion

This work presents PACHS, a novel framework that integrates actor-critic reinforcement learning models with parallel best-first search to enable zero-shot generalization during inference. By leveraging the actor network for action generation and the critic network as a learned heuristic function, PACHS addresses fundamental limitations in RL deployment while requiring no model retraining. PACHS's parallelization strategy incorporates both CPU thread-level and GPU batch-level parallelism, resulting in efficient search-based inference that is practical for deployment. Experimental evaluation across collision-free motion planning and contact-rich manipulation tasks demonstrates improvement in success rates and generalization, particularly in complex environments with obstacles not encountered during training. In the future, we plan to deploy PACHS on real robots to evaluate generalization capabilities for sim2real transfer. We also plan to investigate additional heuristics or provable guarantees on the search.

VII. Acknowledgment

This work was in part supported by NSF grant IIS-2328671.

References

[1] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Int. Conf. on Machine Learning*, pp. 1861–1870, 2018.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[3] R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel, "A survey of zero-shot generalisation in deep reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 76, pp. 201–264, 2023.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[5] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019.

[6] S. J. Arfaee, S. Zilles, and R. C. Holte, "Learning heuristic functions for large state spaces," *Artificial Intelligence*, vol. 175, no. 16-17, pp. 2075–2098, 2011.

[7] R. Veerapaneni, M. S. Saleem, and M. Likhachev, "Learning local heuristics for search-based navigation planning," *Int. Conf. on Automated Planning and Scheduling*, vol. 33, pp. 634–638, Jul. 2023.

[8] D. Bokan, Z. Ajanovic, and B. Lacevic, "Slope: Search with learned optimal pruning-based expansion," *arXiv:2406.04935*, 2024.

[9] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv:1712.01815*, 2017.

[10] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[11] S. Mukherjee, *Parallelized Search on Graphs with Expensive-to-Compute Edges*. PhD thesis, 2023. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2025-09-11.

[12] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8749–8756, IEEE, 2024.

[13] P. S. Ow and T. E. Morton, "Filtered beam search in scheduling†," *Int. Journal of Production Research*, vol. 26, no. 1, pp. 35–62, 1988.

[14] B. Huang, A. Boularias, and J. Yu, "Parallel monte carlo tree search with batched rigid-body simulations for speeding up long-horizon episodic robot planning," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 1153–1160, 2022.

[15] K. Irani and Y.-F. Shih, "Parallel a-asterisk and ao-asterisk algorithms-an optimality criterion and performance evaluation," in *Int. Conf. on Parallel Processing*, pp. 274–277, 1986.

[16] B. Bonet, "Planning as heuristic search," *Artificial Intelligence*, 2001.

[17] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, parallel best-first search for optimal sequential planning," in *Int. Conf. on Automated Planning and Scheduling*, vol. 19, pp. 201–208, 2009.

[18] X. He, Y. Yao, Z. Chen, J. Sun, and H. Chen, "Efficient parallel a* search on multi-gpu system," *Future Generation Computer Systems*, vol. 123, pp. 35–47, 2021.

[19] Y. Zhou and J. Zeng, "Massively parallel a* search on a gpu," in *AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[20] S. Mukherjee, S. Aine, and M. Likhachev, "epa* se: Edge-based parallel a* for slow evaluations," in *Int. Symp. on Combinatorial Search*, vol. 15, pp. 136–144, 2022.

[21] M. Phillips, M. Likhachev, and S. Koenig, "Pa* se: Parallel a* for slow expansions," in *Int. Conf. on Automated Planning and Scheduling*, vol. 24, pp. 208–216, 2014.

[22] S. Mukherjee and M. Likhachev, "Gepa* se: Generalized edge-based parallel a* for slow evaluations," in *Int. Symp. on Combinatorial Search*, vol. 16, pp. 153–157, 2023.

[23] H. Yang, S. Mukherjee, and M. Likhachev, "A-epa* se: Anytime edge-based parallel a* for slow evaluations," in *Int. Symp. on Combinatorial Search*, vol. 16, pp. 163–167, 2023.

[24] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in Neural Information Processing Systems*, vol. 12, 1999.

[25] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[27] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a," *The Int. Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016.

[28] T. Mu, Z. Ling, F. Xiang, D. Yang, X. Li, S. Tao, Z. Huang, Z. Jia, and H. Su, "Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations," *arXiv:2107.14483*, 2021.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[30] Q. Gallouédec, N. Cazin, E. Dellandréa, and L. Chen, "panda-gym: Open-Source Goal-Conditioned Environments for Robotic Learning," *4th Robot Learning Workshop: Self-Supervised and Lifelong Learning at NeurIPS*, 2021.

[31] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.