Supplementary Material

This document presents additional implementation details, visualizations, and conceptual discussions that were excluded or briefed due to space limitations in the main paper. The organization mirrors the sections from the main paper (with the exact same order and numbering), and the exposition generally maintains a question-answer format. It aims to provide significantly more details, at a degree of clarity sufficient for re-implementation. As such, we recommend referring to this document whenever any section in the main paper can benefit from further elaboration. The supplementary material consists of:

- `vizdoom_nav` — navigation videos in VizDoom.
- `habitat_roxbox_nav` — navigation videos in the training apartment Roxbox in Habitat.
- `habitat_annawan_nav` — navigation videos in the test apartment Annawan in Habitat.
- `maze2d_experiments` — goal reaching videos in Maze2D.
- `planning_vis` — videos visualizing the execution of R-RRT and R-RRT*.
- `palmer_supmat.pdf` — this document.

Table of Contents

- What are the details for our implementations of SoRB and SPTM?

4. Experiments

Setup

- What are the details for the experimental setup in VizDoom?

Validating Perceptual Experience Retrieval (PER)

- What is the exact evaluation process that produced Fig.4 in the main paper?

Robust Distances

- What is the exact evaluation process for the right panel of Fig.5 in the main paper?

Proposed Planning Algorithms

- Why does the policy $\pi_{\mathcal{M}^*}$ use R-PRM for planning?
- What are the details for the π_{mpc} baseline?

Experiments in Habitat

- What are the details for the experimental setup in Habitat?
- Why does the agent occasionally take random-looking actions in the habitat navigation trials?

5. Discussion and Future Directions

- How is PALMER related to the "Options Framework (Sutton et al.)" and "Skill-Chaining (Konidaris et al.)"?
- How is PALMER related to "LQR-Trees (Tedrake et al.)?

1 Introduction

Why learn a latent distance metric for nearest neighbor retrieval: In a low-dimensional state space such as 3D positions, L2 distance (i.e., euclidean distance) directly correlates with *local* physical-reachability (i.e., we emphasize local, because euclidian distances still do not match geodesic distances globally). Therefore in such state-spaces, grouping together two nearby states and treating them as the same single state for downstream global planning should still result in a feasible planned trajectory. By feasible, we mean that the gaps and approximations introduced by state grouping are functionally inconsequential and can be handled reasonably well by a local policy tracking the global planned trajectory. This property doesn't hold in high-dimensional state spaces such as images, since the L2 distance doesn't correlate with physical reachability. The main purpose of f_ϕ is to project such high-dimensional spaces into a low-dimensional representation space where this property holds, so that nearby states can be fused together to make sampling-based planning computationally tractable.

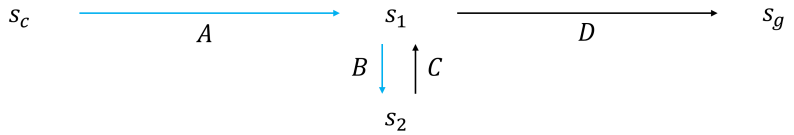

Figure 1: Visualization of stitching together trajectories. If an agent has previous experience of separately going through segments (A, B) and (C, D), it should be able to go from s_c to s_g through the segments (A, D).

What does stitching transitions together mean: As shown in Fig1, if there are two separate sequences of transitions in an offline memory buffer that traverse segments (A, B) and (C, D), an agent should be capable of going from s_c to s_g through the segments (A, D) event if such a direct path of transitions

was never actually observed. Traditional deep Q-learning methods achieve this by combining and propagating value estimates through TD-updates (i.e., $argmax_a Q(s_1, a, s_g)$ points to segment D after TD updates over the path (C, D), therefore the argmax policy would follow the path (A, D) when going from s_c to s_g. [**?**] provides a further discussion). In our approach, this is achieved by setting edge distances for (A, B, C, D) in a planning graph through perceptual experience retrieval, and then performing a shortest path computation to retrieve the path (A, D).

2 Perception-Action Loop with Memory Reorganization

2.1 Perceptual Representations that Capture Local Reachability

What does local reachability mean: At a high-level (and for the special case of image-based navigation) what the term 'local reachability' intends to convey is that if two images I_1 and I_2 are from physically close positions, $d_\phi = |f_\phi(I_1) - f_\phi(I_2)|$ should be small. This in turn provides a metric for grouping together states that is better than the L2 distance in image space, which has no correlation with physical reachability. Such a metric is necessary to make search and sampling-based planning planning over the memory buffer computationally tractable. This learned metric d_ϕ serves the exact same purpose as the hand-crafted image compression criterion employed in [**?**] to initialize cells from states.

2.2 Representation Learning via Reinforcement Learning

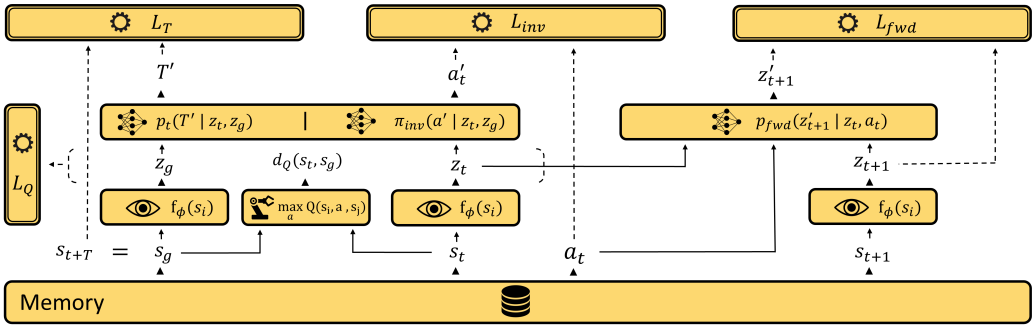

Figure 2: Visualization of the model architecture, reproduced here for ease of reference.

How are the model components trained, and what are their exact inputs and outputs: More detailed descriptions for all model components are given below (Fig.2 presents a visualization):

- The architecture and training of $Q(s_t, a_t, s_g)$ is completely decoupled from the other components. It consists of cascaded convolutional and fully-connected layers with batch normalization and ReLU activations between each layer. It takes as input the concatenated images for current and goal states (i.e., shape $B \times C \times H \times W$), and outputs a vector of Q-values for each action (i.e., shape $B \times num\_actions$). It is trained through offline DDQN [**?**] with hindsight goal-relabelling [**?**]. We first sample $t \sim Uniform(0, dataset\_size)$ and $T \sim Geom(p)$, and then retrieve from the replay buffer a transition and a goal state as $(s_t, a_t, s_{t+1}, s_g := s_{t+T})$. We then minimize the TD error $[\, Q_\theta(s_t, a_t, s_g) - (\mathbb{1}_{s_{t+1}=s_g} + \gamma \mathbb{1}_{s_{t+1} \neq s_g}\, max_a\, Q_{\theta^-}(s_{t+1}, a, s_g))\,]^2$, as in [**?**].
- The perceptual backbone $f_\phi(s)$ uses a standard Resnet-18 architecture. It takes as input the images for a given state (i.e., $B \times C \times H \times W$), and outputs a low-dimensional representation vector $z = f_\phi(s)$ (i.e., shape $B \times D$). All other components take as input these low-dimensional representations, rather than operating over images.
- $p_{fwd}(z'_{t+1} \mid z_t, a_t)$, $\pi_{inv}(a'_t \mid z_t, z_g)$, and $p_t(T' \mid z_t, z_g)$ all consist of fully-connected layers with ReLU activations. To train them, we first sample $t \sim Uniform(0, dataset\_size)$. We then sample T according to $T \sim Uniform(0, T_{max})$ or $T \sim Uniform(T_{max}, dataset\_size - t)$, half the time from the former distribution, half the time from the latter. We retrieve from the replay buffer a transition and a goal state as $(s_t, a_t, s_{t+1}, s_g := s_{t+T})$, and project them into low dimensional representations (z_t, a_t, z_{t+1}, z_g) using f_ϕ. These are concatenated and passed to models $p_{fwd}(z'_{t+1} \mid z_t, a_t), \pi_{inv}(a'_t \mid z_t, z_g), p_t(T' \mid z_t, z_g)$ in a way compatible with their

arguments. p_{fwd} outputs the mean for the predicted next state distribution (i.e., shape $B \times D$), and is trained using the MSE loss L_{fwd} with z_{t+1} as the target. π_{inv} outputs a vector of probabilities over actions (i.e., shape $B \times num\_actions$), and is trained with the cross-entropy loss L_{inv} with a_t as the target. p_t also outputs a discrete probability distribution over $[0, T_{max}]$ that predicts the distribution of time-steps to reach the goal, where the last bin T_{max} serves as a catch-all for all values above it. It is trained using the cross entropy loss L_T, with T as the target. As mentioned in the main paper, all components $f_\phi, p_{fwd}(z'_{t+1} \mid z_t, a_t), \pi_{inv}(a'_t \mid z_t, z_g), p_t(T' \mid z_t, z_g)$ are trained jointly, with an additional loss function L_Q that regularizes f_ϕ.

What do the terms in the contrastive loss-function L_Q mean: The loss function $L_Q(s_t, s_g) = l_{hinge}(d_\phi(s_t, s_g) - d_p)\ \mathbb{1}_{d_Q(s_t,s_g) \leq c_Q} + l_{hinge}(d_p - d_\phi(s_t, s_g))\ \mathbb{1}_{d_Q(s_t,s_g) \geq c_Q}$ consists of two penalty terms $l_{hinge}(d_\phi(s_t, s_g) - d_p)$ (i.e., only active when $d_\phi \geq d_p$) and $l_{hinge}(d_p - d_\phi(s_t, s_g))$ (i.e., only active when $d_\phi \leq d_p$). These penalty terms are gated through two complementary indicator functions $\mathbb{1}_{d_Q(s_t,s_g) \leq c_Q}$ and $\mathbb{1}_{d_Q(s_t,s_g) \geq c_Q}$. This essentially means that for L_Q to be zero, $d_\phi \leq d_p$ should hold (i.e., perceptual representations are close) if and only if $d_Q(s_t, s_g) \leq c_Q$ holds (i.e., states are physically close). The reason for employing such a conservative switching mechanism with a hinge loss in L_Q (i.e., rather than a continuous penalty term as in [**? ?**]) is because Q-value estimates are quite inaccurate (especially when s_c and s_g are far apart), and their exact value is generally unreliable (i.e., they can indicate whether two-states are close sufficiently well, but cannot robustly answer how close they are). To pick the hyperparameter c_Q, we compute the average Q-value between states in the replay buffer that were observed to be within one-step proximity, and use a fraction of this value to as a conservative estimate. While conceptually the choice for the hyperparameter d_p is arbitrary, we heuristically pick it by examining the average d_ϕ distance between subsequent states in the replay buffer, obtained from a preliminary f_ϕ backbone trained without L_Q.

2.3 Perceptual Experience Retrieval (PER)

How do we solve the optimization problem in equation 1 from the main paper: Our experiments use $-\mathcal{R}(\tau) = len(\tau)$. We first compute the perceptual representations z for all states in the replay buffer, and stack them into a tensor block of shape $(dataset\_size, D)$. Given s_c and s_g, we search this tensor with vectorized masking operations to retrieve a set of neighboring states $N(s_c, d_p)$ and $N(s_g, d_p)$ (i.e., sets of states within a perceptual distance threshold d_p of s_c and s_g), to address cons.4. We sort the resulting pairs of states $(s_i, s_j) \in N(s_c, d_p) \times N(s_g, d_p)$ in terms of $j - i$, and filter these pairs using cons.5. We then pick the first (i.e., closest) pair, and return all the states with indices between i, j from the replay buffer as the resulting trajectory $\tau_{\mathcal{M}(s_c,s_g)}$. The important thing to emphasize about all of these operations is that they can be trivially vectorized, and therefore the optimization problem in eq.3-5 can be solved in less time than a forward pass of f_ϕ.

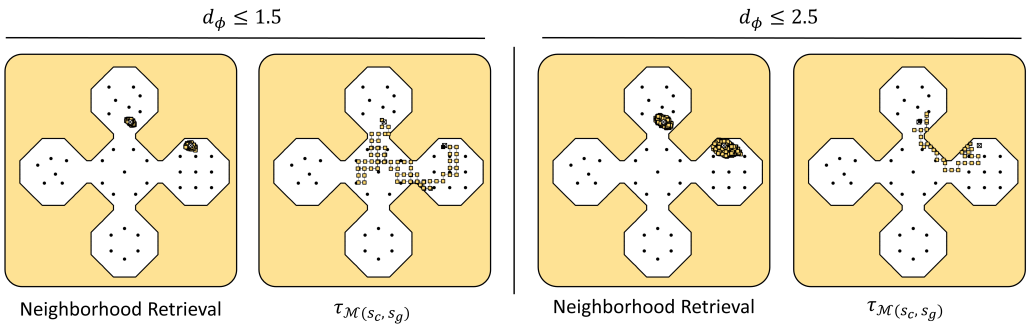

Figure 3: Visualization of retrieved trajectories, with different perceptual distance threshold. Query states s_c and s_g are represented as white squares with a cross in the center, while the start and end points of the retrieved trajectory $\tau_{\mathcal{M}(s_c,s_g)}$ are denoted with a black square and a yellow square with a diagonal dash respectively.

What do the retrieved trajectories $\tau_{\mathcal{M}(s_c,s_g)}$ look like: As the perceptual distance threshold for d_ϕ increases, the physical radii spanned by the nearest neighbor sets $N(s_c, d_p)$ and $N(s_g, d_p)$ increase, as shown in Fig.3. As a result, constraint 4 in the PER equation gets looser, more trajectories satisfy constraints 4 and 5 (because their start and end points are allowed to deviate further from the query pair s_c, s_g), and therefore the optimization in equation 3 returns a shorter trajectory $\tau_{\mathcal{M}(s_c,s_g)}$.

2.4 Long-Horizon Planning Through Stitching Trajectory Segments

Algorithm 1 Classic PRM (Roadmap Construction)

$V \leftarrow \{SampleFree_i\}_{i=1,\dots,num\_vertices};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for each $s_i \in V$ **do**
  $U \leftarrow Near(V, s_i, r) \setminus \{s_i\}$
  for each $s_j \in U$ **do** ▷ Draw lines as edges
    if $CollisionFree(s_i, s_j)$ **then** $E \leftarrow E \cup \{(s_i, s_j), (s_j, s_i)\}$
return $G = (V, E)$

Algorithm 2 Classic PRM (Shortest-Path Queries Over the Roadmap)

Input: $s_c, s_g, G = (V, E), \mathcal{R}(\tau), f_\phi, \mathcal{M}$
for each $s_i \in V$ **do** ▷ Insert s_c and s_g into the PRM graph
  if $CollisionFree(s_c, s_i)$ **then**
    $E \leftarrow E \cup \{(s_c, s_i), (s_i, s_c)\}$
  if $CollisionFree(s_i, s_g)$ **then**
    $E \leftarrow E \cup \{(s_i, s_g), (s_g, s_i)\}$
return $\{s_j\} \leftarrow ShortestPath(s_c, s_g, G)$ ▷ Shortest path through graph search

Algorithm 3 R-PRM (Roadmap Construction)

Input: $f_\phi, \mathcal{M}$
$V \leftarrow \{SampleFree_i\}_{i=1,\dots,num\_vertices};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for each $s_i \in V$ **do**
  $U \leftarrow Near(V, s_i, r) \setminus \{s_i\}$
  for each $s_j \in U$ **do** ▷ Place PER trajectories in edges
    $E \leftarrow E \cup \{(s_i, s_j) :\ \tau_{edge} = \tau_{\mathcal{M}(s_i, s_j)},\ d_{edge} = -\mathcal{R}(\tau_{\mathcal{M}(s_i, s_j)})\}$
return $G = (V, E)$

How does R-PRM work in detail: Alg.1, 2 give step-by-step descriptions for the classical PRM algorithm (adapted from [**?**]), while Alg.3, 4 describe our new definitions for R-PRM. It can be seen that there are two main differences:

- In R-PRM, whenever an edge is created, a trajectory $\tau_{\mathcal{M}(s_c, s_i)}$ is retrieved through perceptual experience retrieval and stored in a field τ_{edge}, while its reward $-\mathcal{R}(\tau_{\mathcal{M}(s_c, s_i)})$ is stored in a different field d_{edge}.
- In PRM the length and cost of a line segment are the same (i.e., euclidian distance), whereas in R-PRM the length of a trajectory $len(\tau_{\mathcal{M}(s_c, s_i)})$ and its reward $-\mathcal{R}(\tau_{\mathcal{M}(s_c, s_i)})$ are decoupled. This means that a shortest path query in R-PRM returns a sequence of nodes and edges that optimize the reward function $\mathcal{R}$. An additional step in R-PRM is that at the end of the shortest-path query, all trajectories $\tau_{edge} = \tau_{\mathcal{M}(s_{i-1}, s_i)}$ stored in the returned edges are concatenated into a single trajectory $\tau_{\mathcal{M}^*(s_c, s_g)} = \tau_{stitched}$.

How does R-RRT work in detail: Alg.5 gives a step-by-step description for the classical RRT algorithm (adapted from [**?**]), while Alg.6 describes our new definition for R-RRT. In addition to the two previous differences between PRM and R-PRM, there is one additional difference between RRT and R-RRT. In RRT, there is a steering sub-routine that draws a line segment of length r starting from $s_{nearest}$ and extending towards s_{rand}, to create a new vertex s_{new}. In R-RRT, this is replaced by retrieving a trajectory $\tau_{\mathcal{M}(s_{nearest}, s_{rand})}$ starting from $s_{nearest}$ and ending at s_{rand}, and its r 'th state is used to create the new vertex s_{new}.

How does R-RRT work in detail:* Alg.7 gives a step-by-step description for the classical RRT* algorithm (adapted from [**?**]), while Alg.8 describes our new definition for R-RRT*. R-RRT* almost exactly maintains the tree rewiring machinery employed in RRT*, the only difference being that the

Algorithm 4 R-PRM (Trajectory Restitching Given the Constructed Roadmap)

Input: $s_c, s_g, G = (V, E), \mathcal{R}(\tau), f_\phi, \mathcal{M}$
for each $s_i \in V$ **do** ▷ Insert s_c and s_g into the PRM graph
  if $len(\tau_{\mathcal{M}(s_c,s_i)}) \leq r$ **then** ▷ Place PER trajectories in edges
    $E \leftarrow E \cup \{(s_c, s_i) : \tau_{edge} = \tau_{\mathcal{M}(s_c,s_i)},\ d_{edge} = -\mathcal{R}(\tau_{\mathcal{M}(s_c,s_i)})\}$
  if $len(\tau_{\mathcal{M}(s_i,s_g)}) \leq r$ **then**
    $E \leftarrow E \cup \{(s_i, s_g) : \tau_{edge} = \tau_{\mathcal{M}(s_i,s_g)},\ d_{edge} = -\mathcal{R}(\tau_{\mathcal{M}(s_i,s_g)})\}$

$\tau_{stitched} \leftarrow \emptyset$
$\{s_j\} \leftarrow ShortestPath(s_c, s_g, G, \mathcal{R}(\tau))$ ▷ Trajectory stitching by dynamic programming
for $0 < i < |\{s_j\}|$ **do** ▷ Concatenate PER trajectories along the shortest path
  $\tau_{stitched} \leftarrow \tau_{stitched} \circ \tau_{\mathcal{M}(s_{i-1},s_i)}$
return $\tau_{\mathcal{M}^*(s_c,s_g)} = \tau_{stitched}$

Algorithm 5 Classic RRT

$V \leftarrow \{s_{init}\};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for $i = 1, ..., n$ **do**
  $s_{rand} \leftarrow SampleFree_i$ ▷ Sample random vertex
  $s_{nearest} \leftarrow Nearest(V, s_{rand})$ ▷ Find the nearest vertex in V
  $s_{new} \leftarrow Steer(s_{nearest}, s_{rand}, r)$ ▷ Draw a line segment of length r
  if $CollisionFree(s_{nearest}, s_{new})$ **then**
    $V \leftarrow V \cup \{s_{new}\}\ ;\ E \leftarrow E \cup \{(s_{nearest}, s_{new}), (s_{new}, s_{nearest})\}$
return $G = (V, E)$

line costs (i.e., euclidian distance) are replaced with $-\mathcal{R}(\tau_{\mathcal{M}(s_i,s_j)})$ (i.e., in addition to the previous three differences from R-PRM and R-RRT).

What does optimizing the Bellman error on the roadmap mean: Dynamic programming based graph-search algorithms like Dijkstra or (discrete) value-iteration generally employ a cost caching mechanism to iteratively update cost-to-come values, and these updates reduce Bellman error (i.e., the difference between the cost-to-come values before and after the update) during forward-search [**?**]. R-PRM uses Dijkstra for shortest-path search, while R-RRT* employs the same dynamic programming based tree-rewiring mechanism as the original RRT*, therefore both algorithms are optimizing the Bellman error between the vertices of their graphs through dynamic programming.

What does restitching transitions at arbitrary resolution mean: The retrospective-planning algorithms R-PRM, R-RRT, R-RRT* are sampling based. This means that if two consequent states s_t and s_{t+1} are retrieved during their $SampleFree_i$ routines, then these algorithms will set an edge using $\tau_{\mathcal{M}(s_t,s_{t+1})}$, which is simply the single transition (s_t, a_t, s_{t+1}). Therefore, given enough samples, these algorithms can restitch trajectories down to the level of individual transitions.

2.5 Refining Memory Contents via Forming and Executing Plans

What does optimizing memory contents mean: A replay buffer $\mathcal{M}$ is a collection of trajectories. Optimizing its contents means adding trajectories to $\mathcal{M}$ that achieve higher total reward.

How does the perception-action loop in PALMER work in detail: Alg.9 gives a step-by-step description of the overall perception-action loop implemented in PALMER. **What PALMER does is essentially bridging any auxiliary exploration method with any auxiliary exploitation method**, by reorganizing exploration experience in $\mathcal{M}$ into $\tau_{\mathcal{M}^*(s_c,s_g)}$ that can be used for exploitation. The particular way in which the trajectories $\tau_{\mathcal{M}^*(s_c,s_g)}$ can be executed for exploitation has a great deal of flexibility, for example: all actions in $\tau_{\mathcal{M}^*(s_c,s_g)}$ can be executed sequentially in an open-loop manner, the first actions of $\tau_{\mathcal{M}^*(s_c,s_g)}$ generated at each timestep can be executed in a model predictive control (MPC) manner, states in $\tau_{\mathcal{M}^*(s_c,s_g)}$ can be tracked by an auxiliary local feedback controller, or the entirety of $\tau_{\mathcal{M}^*(s_c,s_g)}$ can be used to initialize a separate trajectory optimization method.

Algorithm 6 R-RRT

$V \leftarrow \{s_{init}\};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for $i = 1, ..., n$ **do**
  $s_{rand} \leftarrow SampleFree_i$ ▷ Sample random vertex
  $s_{nearest} \leftarrow Nearest(V, s_{rand})$ ▷ Find the nearest vertex in V
  $s_{new} \leftarrow \tau_{\mathcal{M}(s_{nearest}, s_{rand}), r}$ ▷ Get the r 'th state in $\tau_{\mathcal{M}(s_{nearest}, s_{rand})}$
  if $len(\tau_{\mathcal{M}(s_{nearest}, s_{new})}) \leq r$ **then**
    $V \leftarrow V \cup \{s_{new}\}$
    $E \leftarrow E \cup \{(s_{nearest}, s_{new}) :\ \tau_{edge} = \tau_{\mathcal{M}(s_{nearest}, s_{new})},$
      $d_{edge} = -\mathcal{R}(\tau_{\mathcal{M}(s_{nearest}, s_{new})})\}$
return $G = (V, E)$

Algorithm 7 Classic RRT*

$V \leftarrow \{s_{init}\};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for $i = 1, ..., n$ **do**
  $s_{rand} \leftarrow SampleFree_i$ ▷ Sample random vertex
  $s_{nearest} \leftarrow Nearest(V, s_{rand})$ ▷ Find the nearest vertex in V
  $s_{new} \leftarrow Steer(s_{nearest}, s_{rand}, r)$ ▷ Draw a line segment of length r

  if $CollisionFree(s_{nearest}, s_{new})$ **then**
    $S_{near} \leftarrow Near(V, s_{new}, r)$
    $V \leftarrow V \cup \{s_{new}\}$
    $s_{min} \leftarrow s_{nearest}$
    $c_{min} \leftarrow Cost(s_{nearest}) + Cost(Line(s_{nearest}, s_{new}))$

    for each $s_{near} \in X_{near}$ **do** ▷ Connect along a minimum-cost path
      $c_{near} \leftarrow Cost(s_{near}) + Cost(Line(s_{near}, s_{new}))$
      if $CollisionFree(s_{near}, s_{new})$ and $c_{near} \leq c_{min}$ **then**
        $s_{min} \leftarrow s_{near}\ ;\ c_{min} \leftarrow c_{near}$
    $E \leftarrow E \cup \{(s_{min}, s_{new})\}$

    for each $s_{near} \in X_{near}$ **do** ▷ Rewire the tree
      $c_{near,new} \leftarrow Cost(s_{new}) + Cost(Line(s_{new}, s_{near}))$
      if $CollisionFree(s_{new}, s_{near})$ and $c_{near,new} \leq c_{near}$ **then**
        $s_{parent} \leftarrow Parent(s_{near})$
        $E \leftarrow (E \backslash \{(s_{parent}, s_{near})\} \cup \{s_{new}, s_{near}\})$

return $G = (V, E)$

3 Related Work

What is the main reason why memory-based reasoning over actually-observed transitions is necessary? Why are methods like SPTM or SoRB that solely rely on learning-based distance estimates are inherently prone to false predictions: By definition, states that are far apart from each other in-terms of physical reachability are rarely observed together in an experiential learning framework (e.g., within the same RL episode, or within close by time-steps during random exploration). Therefore, for any learning-based prediction model that is conditioned on current-goal state pairs (e.g., $Q(s_t, a, s_g),\ \pi_{inv}(a'|s_t, s_g),\ p_t(T'|s_t, s_g)$), if it is trained solely on the observed distribution of experiential data, far apart states will be out of distribution (i.e., they have a low probability of being sampled from the experiential data distribution, therefore they are underrepresented in the replay buffer). This means that predictions for such far apart states will inevitably be inaccurate, unless a hard-negative sampling mechanism is implemented to explicitly push their reachability-estimates lower. The problem with this is that there is no inherent signal solely contained in perceptual input (e.g., images) that can guide the resampling process in a way that is generally applicable to all tasks.

Algorithm 8 R-RRT*

$V \leftarrow \{s_{init}\};\ E \leftarrow \emptyset$ ▷ Initialize vertices and edges
for $i = 1, ..., n$ **do**
  $s_{rand} \leftarrow SampleFree_i$ ▷ Sample random vertex
  $s_{nearest} \leftarrow Nearest(V, s_{rand})$ ▷ Find the nearest vertex in V
  $s_{new} \leftarrow \tau_{\mathcal{M}(s_{nearest}, s_{rand}), r}$ ▷ Get the r 'th state in $\tau_{\mathcal{M}(s_{nearest}, s_{rand})}$

  if $len(\tau_{\mathcal{M}(s_{nearest}, s_{new})}) \leq r$ **then**
    $S_{near} \leftarrow Near(V, s_{new}, r)$
    $V \leftarrow V \cup \{s_{new}\}$
    $s_{min} \leftarrow s_{nearest}$
    $c_{min} \leftarrow Cost(s_{nearest}) + -\mathcal{R}(\tau_{\mathcal{M}(s_{nearest}, s_{new})})$

    for each $s_{near} \in X_{near}$ **do** ▷ Connect along a minimum-cost path
      $c_{near} \leftarrow Cost(s_{near}) + -\mathcal{R}(\tau_{\mathcal{M}(s_{near}, s_{new})})$
      if $len(\tau_{\mathcal{M}(s_{near}, s_{new})}) \leq r$ and $c_{near} \leq c_{min}$ **then**
        $s_{min} \leftarrow s_{near}\ ;\ c_{min} \leftarrow c_{near}$
    $E \leftarrow E \cup \{(s_{min}, s_{new}) :\ \tau_{edge} = \tau_{\mathcal{M}(s_{min}, s_{new})}, d_{edge} = -\mathcal{R}(\tau_{\mathcal{M}(s_{min}, s_{new})})\}$

    for each $s_{near} \in X_{near}$ **do** ▷ Rewire the tree
      $c_{near,new} \leftarrow Cost(s_{new}) + -\mathcal{R}(\tau_{\mathcal{M}(s_{new}, s_{near})})$
      if $len(\tau_{\mathcal{M}(s_{new}, s_{near})}) \leq r$ and $c_{near,new} \leq c_{near}$ **then**
        $s_{parent} \leftarrow Parent(s_{near})$
        $E \leftarrow (E \backslash \{(s_{parent}, s_{near})\} \cup \{s_{new}, s_{near}\})$

return $G = (V, E)$

For example, [**?**] employs a hard-negative sampling mechanism for a manipulation task using joint pose labels for guidance, but such a mechanism has two bottlenecks: i) it is specific to their particular manipulation task, ii) it assumes auxiliary labels. Similarly, the temporally consistent localization mechanism used in SPTM is essentially a heuristic fix specific to navigation. For the case of SoRB, there is no inherent mechanism in distributional RL that addresses this out-of-distribution issue either. While employing an ensemble of Q-functions could potentially capture the epistemic uncertainty for out-of-distribution pairs to directly address this problem, we used an ensemble of Q-functions in our implementation of SoRB and empirically observed that it was insufficient. A similar conclusion can be drawn from the Fig.8 of the original SoRB paper [**?**], as the bulk of the performance increase appears to be due to distributional RL, and ensembles only provide a moderate benefit.

All of these considerations highlight the importance of memory as a robustification mechanism. To summarize the discussions from above, there are two main reasons that cause false reachability predictions: i) there is no robust and general signal solely contained in the isolated instances of perceptual input (s_c, s_g) (i.e., without the trajectory of states in between that connect them) that can identify whether two states are physically far apart, ii) both physically close and far apart states can occur with a long temporal distance in between. Therefore, an agent needs to rely on memory: in order to identify whether two states are close or not, it should try to remember if it ever actually observed those two states close-by in a segment of past experience.

What are the details for our implementations of SoRB and SPTM: Tha main difference of our SPTM implementation is that it doesn't employ temporally consistent localization and adaptive waypoint selection. The main differences of our SoRB implementation are: i) we use an ensemble of Q-functions, but they are trained with DDQN rather than distributional RL, ii) we train the Q-function on offlline random-walk data, rather than an online episodic training setup with resets and a reward oracle as employed in the original paper. We acknowledge and emphasize that for SoRB, these differences are the most likely reason for the lower performance level we observed in our evaluations compared to the original paper, as they inevitably reduce the accuracy of Q-values. We however note that our method also employs the same Q-values, and generally these implementation differences in the baselines were chosen to facilitate a clear understanding of our approach without confounders,

Algorithm 9 PALMER: Perception-Action Loop with Memory Retrieval

Input: $\mathcal{R}(\tau)$
$f_\phi.init(), Q.init(), \mathcal{M} \leftarrow \emptyset$ ▷ Initialize policy parameters
while $t \leq max\_timestep$ **do**

  while $i \leq num\_exploration\_steps$ **do** ▷ Exploration
    i) Using [*any suitable method*]: explore the environment to obtain an exploration trajectory τ_{new}
    ii) Using [τ_{new}]: **update** $\mathcal{M}$ ▷ Memory expansion

  while $i \leq num\_updates$ **do**
    Using [$\mathcal{M}$]: **update** $Q(s_t, a_t, s_g)$ ▷ Train value function
    Using [$\mathcal{M}$ and $Q(s_t, a_t, s_g)$]: **update** $f_\phi(s_t, s_g)$ ▷ Train perception model

  while $i \leq num\_exploitation\_steps$ **do** ▷ Exploitation
    i) Using [$\mathcal{M}$]: sample a random goal $s_g \sim \mathcal{M}$
    ii) Using [$f_\phi(s_c, s_g)$ and $\mathcal{R}(\tau)$]: generate $\tau_{\mathcal{M}^*(s_c,s_g)}$
    iii) Using [*any suitable method*]: execute $\tau_{\mathcal{M}^*(s_c,s_g)}$ to obtain a real trajectory τ_{real}
    iv) Using [τ_{real}]: **update** $\mathcal{M}$ ▷ Memory optimization

because: i) they do not directly address the root cause of the false prediction problem (as discussed above), ii) one of the main benefits of our method is that it operates over arbitrary offline data without any resets or reward oracles (and it uses DDQN to train the related Q-function).

4 Experiments

Setup

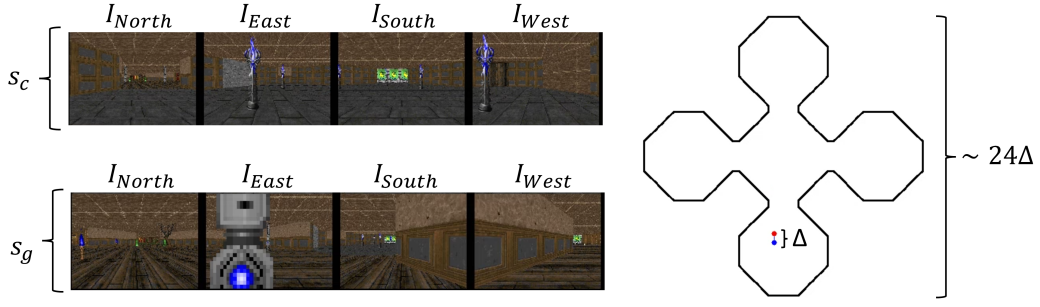

Figure 4: Visualization of the map used in VizDoom experiments. **Further video visuals of image-based navigation can be found in the folder vizdoom_nav provided in the supplementary alongside this document**.

What are the details for the experimental setup in VizDoom: As shown in Fig.4, states solely consist of four images $I_{North/East/South/West}$ that form a panorama (i.e., $4 \times 3 \times 160 \times 120$ dimensions), and actions move the agent North/South/East/West by a fixed distance Δ. The geodesic distances scale approximately by a factor of $\times 3$ compared to euclidian distances (e.g., If a goal has an euclidian distance of 14Δ, it takes approximately 42 timesteps for an optimal policy to reach it). The map contains many long-thin column-like obstructions (e.g., torches, pillars, trees), as we found that image-based navigation policies are prone to getting stuck in such obstacles. These obstacles have dynamically changing appearances (e.g., flickering flames on torches, glowing lights on pillars), and can completely block the field of view of the agent after a collision (as shown in I_{East} in Fig.4). The replay buffer $\mathcal{M}$ consists of 300k images obtained from a uniform random walk exploring the map in a single continuous sequence of actions, without resets and rewards.

Validating Perceptual Experience Retrieval (PER)

What is the exact evaluation process that produced Fig.4 in the main paper: For every integer value $n \in [0, 14]$, we randomly sample 1000 pairs of start and goal-states in a way that the euclidian distance between them lies within $n \times \Delta$ and $(n+1) \times \Delta$ through rejection sampling, and a policy is considered successful if it can get within Δ proximity of the goal-state.

Robust Distances

What is the exact evaluation process for the right panel of Fig.5 in the main paper: To produce the roadmap visualizaitons, we randomly sample 250 states from the replay buffer and set the edges between them by thresholding the distance estimates from all methods. Thresholds were calibrated individually and by hand for each baseline, by picking the threshold with the lowest number of false edges until a further reduction in the threshold resulted in splitting the roadmap into a large number of isolated subgraphs (i.e., therefore making it impossible to use it for global planning).

Proposed Planning Algorithms

Why does the policy $\pi_{\mathcal{M}^*}$ use R-PRM for planning: PRM and R-PRM are multi-query methods [**?**], meaning that the full roadmap only needs to be constructed once. For any query pair of current-goal states (s_c, s_g), the same roadmap can be reutilized by inserting (s_c, s_g) in the roadmap and performing a shortest path query. In contrast, RRT and RRT* require recreating a full roadmap for every new query pair. Since $\pi_{\mathcal{M}^*}$ replans at each timestep with a different current state s_t, PRM based approaches are computationally much cheaper. We also note that planning graphs for all methods in this experiment contain 500 vertices.

What are the details for the π_{mpc} baseline: The π_{mpc} policy uses the p_{fwd} model to obtain simulated rollouts, and uses the p_t model to rank those rollouts in terms of how close they get to the goal. This allows an MPC optimization loop that picks and implements the first action from the most successful simulated rollout. As previously discussed above, the main bottleneck for SPTM and SoRB is the difficulty of estimating reachability metrics solely using the two states s_c, s_g without any consideration of the states in between. Simulated rollouts in π_{mpc} naturally address this problem by generating and evaluating entire trajectories. The main bottleneck for π_{mpc} is that the accuracy of state predictions in simulated rollouts from p_{fwd} degrade with the rollout length.

Experiments in Habitat

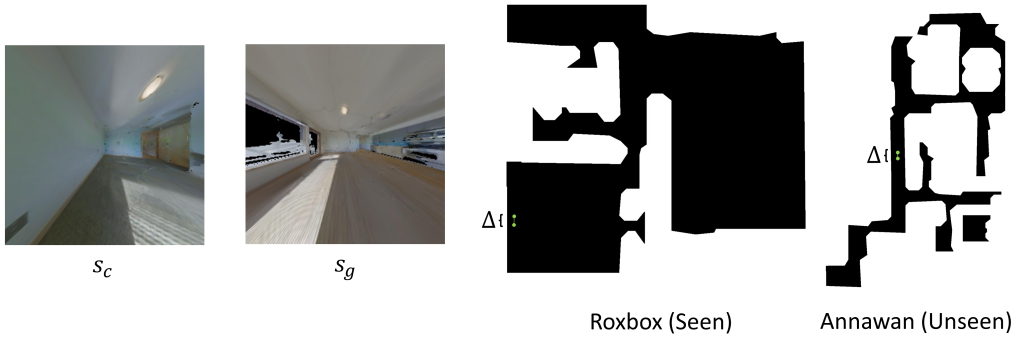

Figure 5: Visualization of the apartments used in Habitat experiments. **Further video visuals of image-based navigation can be found in the folders habitat_roxbox_nav and habitat_annawan_nav provided in the supplementary alongside this document**.

What are the details for the experimental setup in Habitat: As shown in Fig.5, states solely consist of a single 150 FOV image (i.e., $3 \times 256 \times 256$ dimensions). There are 3 actions: $\{turn\_left\_30\_deg, turn\_right\_30\_deg, move\_forward\_\Delta\}$. We run evaluations on two randomly picked apartments: Roxbox, and Annawan. In both apartments, we collect a replay buffer $\mathcal{M}$ that consists of 150k images obtained from a uniform random walk exploring the map in a single

continuous sequence of actions, without resets and rewards. We use only the memory buffer from Roxbox to train the perception model f_ϕ, and use this same model to do perceptual experience retrieval and trajectory stitching on replay buffers from both apartments. We have observed that the latent distances from f_ϕ generalize well, and can directly allow perceptual experience retrieval and trajectory stitching without any fine-tuning on the images from the new test apartment.

Why does the agent occasionally take random-looking actions in the habitat navigation trials: This is due to a combination of two main factors. First, our MPC loop replans from scratch at each timestep using Algorithm.4. This frequent replanning has a destabilizing effect on the control loop, similar to employing a large derivative action in a PID controller (i.e., a strong anticipatory term causes frequent switches in the actions). This first factor is exacerbated by the second main factor: the poor performance of $argmax_a\ Q(s_t, a, s_g)$. This is most likely due to the difficulty of offline RL training with hindsight relabelling over random-walk data of only 150k timesteps obtained with a much more challenging non-cartesian action space $\{turn\_left\_30\_deg, turn\_right\_30\_deg, move\_forward\_\Delta\}$. The restitched trajectories $\tau_{\mathcal{M}^*(s_t,s_g),s,1})$ produced by R-PRM at each time-step are converted to actions by following their first state $\tau_{\mathcal{M}^*(s_t,s_g),s,1}$ using $argmax_a\ Q(s_t, a, \tau_{\mathcal{M}^*(s_t,s_g),s,1})$, and inaccurate Q-values occasionally cause random-looking actions.

There are multiple ways to counter this. The most direct way is to train a better Q-function. Our current Q-function is trained in a particularly challenging setting, as: i) it is trained on entirely offline data with hindsight goal relabelling, ii) this data is collected through uniformly random actions, and iii) it consists of only 150k environment steps. A second way is to instead counter the large derivative action by reducing the replanning rate and introducing a momentum mechanism to the controller. For example, we can replan through R-PRM only every n'th timestep (i.e., hence reducing the replanning rate), and act according to $argmax_a\ Q(s_t, a, \tau_{\mathcal{M}^*(s_t,s_g),s,n})$ for the timesteps inbetween (i.e., hence introducing momentum to the controls). We couldn't get this fix to work well, because the Q-values are only accurate up to states that are $\sim 2 \times \Delta$ distance away (hence acting according to $argmax_a\ Q(s_t, a, \tau_{\mathcal{M}^*(s_t,s_g),s,n})$ isn't possible for $n \geq 2$ in our case). We note however that our method can still navigate from any point to any point in a challenging 3D reconstruction of a real-world apartment in Habitat using poor Q-value estimates, highlighting the robustness introduced by memory-based planning.

5 Discussion and Future Directions

How is PALMER related to the "Options Framework (Sutton et al.)" and "Skill-Chaining (Konidaris et al.)": The idea of restitching (i.e., chaining) transition sequences from a replay buffer has direct connections to the options framework [**?**] and skill-chaining [**?**]. Essentially, PALMER can be thought of as a framework for converting every possible sequence of transitions $\tau \in \mathcal{M}$ in memory into an option $o = \{\pi_o, I_o, \beta_o\}$, where the option policy π_o is implemented by simply executing all the actions in τ in an open-loop manner, the initiation set is $I_o = \{s \in \mathcal{S} : d_\phi(s, \tau_{s,0}) \leq d_p)\}$, and the termination condition is $\beta_o = \{s \in \mathcal{S} : d_\phi(s, \tau_{s,-1}) \leq d_p)\}$. Therefore, PALMER can be interpreted as a skill-chaining algorithm that converts unstructured transitions in a replay buffer into a set of executable options to be chained.

How is PALMER related to "LQR-Trees (Tedrake et al.)": A central idea in PALMER is repurposing the edge creation subroutines of sampling-based planning algorithms so that whenever an edge is created some additional processing is done to connect the endpoints (i.e., particularly, perceptual experience retrieval in our case). This approach is directly inspired by the method of LQR-Trees [**?**], which instead creates a trajectory stabilizing LQR controller to connect the endpoints of each edge. This results in a roadmap of local controllers, rather then a roadmap of memories as in PALMER.