Supplementary Material

Anonymous submission

A. Environment Details

The environments of AntMaze (U-shape), AntMaze (S-shape), AntMaze (W-shape), AntFourRooms and HalfCheetahHurdle are shown in Figure 1. The rewards are sparse and binary in all environments. For Ant tasks, when the L2 distance between the *xy* coordinates of the simulated ant robot and the desired goal is less than 1.5, the ant robot achieves a "success" and gets reward 1; otherwise, the reward is 0. For HalfCheetahHurdle task, when the L2 distance between the *xy* coordinates of the simulated cheetah robot and the desired goal is less than 0.5, the cheetah robot achieves a "success" and gets reward 1; otherwise, the reward is 0. The starting *xy* position is fixed to $(0, 0)$ for all the environments. For Ant tasks, in the training phase, the goal position is randomly sampled inside the environment, while in the evaluation phase, the goal position is fixed to different points for different environments. For the HalfCheetah task, the goal position is fixed both in the training and evaluation phase.

- AntMaze (U-shape): The edge length of the U-shaped maze is 12. In the evaluation phase, the goal position is fixed to the farthest point $(0, 8)$. The maximum timestep for each episode is set to 500 for both training and evaluation.
- AntMaze (S-shape): The edge length of the S-shaped maze is 20. In the evaluation phase, the goal position is fixed to the farthest point $(16, 16)$. The maximum timestep for each episode is set to 1000 for both training and evaluation.
- AntMaze (W-shape): The edge length of the W-shaped maze is 20. In the evaluation phase, the goal position is fixed to the farthest point $(0, 8)$. The maximum timestep for each episode is also set to 1000 for both training and evaluation.
- AntFourRooms: The edge length of the environment is 18. In the evaluation phase, the goal position is fixed to the farthest point $(14, 14)$. The maximum timestep for each episode is set to 1000 for both training and evaluation.
- HalfCheetahHurdle: The height of the hurdle is 0.35. The goal position is fixed to $(4, 0.2)$ for both training and evaluation. The maximum timestep for each episode is set to 1000.

B. Implementation Details

B.1 Network Structure

The actor and critic networks for both high-level and low-level policies are implemented as multi-layer perceptrons. Each network has two hidden layers of dimension 256, and ReLU activations are used in all these networks. For high-level and low-level actor networks, the output is scaled to $[-1, 1]$ by the Tanh function and then scaled to the range of the corresponding action space using linear interval transformation. The representation network ϕ is implemented as a multi-layer perceptron with one hidden layer of dimension 100. ReLU activations are also applied in ϕ.

We implement our code based on Python 3.9.18 and the PyTorch framework of version "2.2.1+cu118". We train all neural networks using the Adam optimizer. All experiments are processed on a single NVIDIA TITAN Xp GPU. The Linux version is "5.15.0-136-generic", and the model name of the CPUs is "Intel(R) Xeon(R) CPU E5-2643 v4 @ 3.40GHz". The total memory is 256 GiB.

B.2 Hyperparameters

We list the main hyperparameters and their ranges considered in the experiments in Table 1 and 2. For Ant tasks, the initial probability p of frontier-reaching assignments is 0.2, and it is gradually increased to 0.4 as the training processes. For the HalfCheetahHurdle task, this probability is fixed at 0.1.

C. Pseudo-code of ASI

We provide the pseudo-code of ASI algorithm, as shown in Algorithm 1.

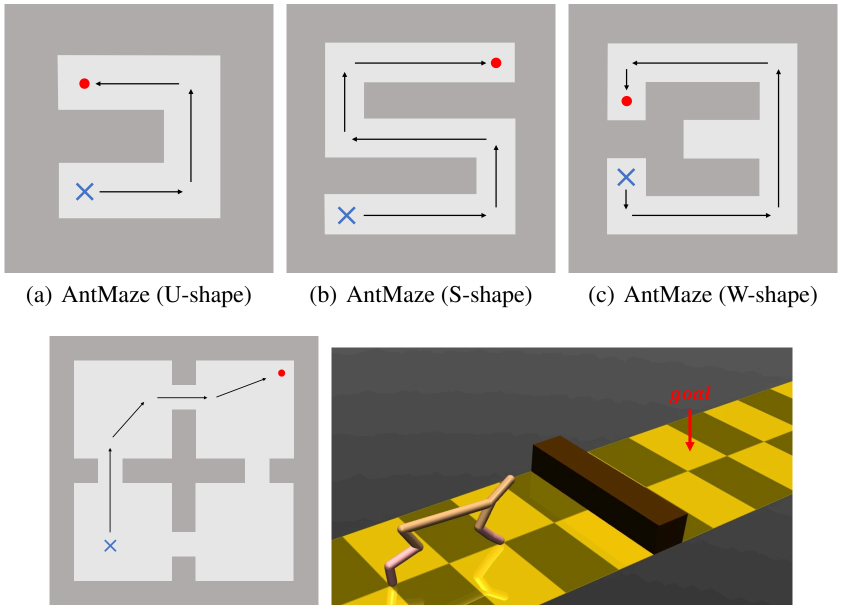
D. Code Implementations of Core Modules

For easier comprehension, we provide partial code implementations, including the FR subgoal selection strategy (as shown in Listing 1) and the hierarchical self-imitation learning method (as shown in Listing 2 and 3).

E. Additional Experimental Results

E.1 Visualization of Coverage Ratio

We have displayed coverage ratio curves in the S-shaped AntMaze environment in the ablation study section. To show the effect of ASI more intuitively, we additionally provide a visualized result of coverage ratio in the U-shaped AntMaze environment in Figure 2. It can be seen that ASI explores faster than HESS, demonstrating its effectiveness.



(a) AntMaze (U-shape) (b) AntMaze (S-shape) (c) AntMaze (W-shape)

(d) AntFourRooms (e) HalfCheetahHurdle

Figure 1: Environments used in the experiments.

Table 1: Hyper-parameters across all environments.

Hyper-parameters	Values	Ranges
Representation dimension	2	
High-level action interval c	20	
Replay buffer size	5e6	
Learning rate for both levels of policies	0.0002	
Discount factor for both levels γ	0.99	
Learning rate for subgoal representation	0.0001	
Batch size for representation learning	100	
Maximum number of nodes within the latent graph	5000	$\{2000,3000,5000\}$
Batch size of self-imitation learning for both levels	128	
Number of low-level self-imitation targets $\|u\|$	3	
Balancing coefficient for high-level self-imitation α	0.01	$\{0.001,0.01,0.1,1\}$
Balancing coefficient for low-level self-imitation β	0.001	$\{0,0.001,0.01,0.1\}$

Table 2: Hyper-parameters that differ across the environments.

Hyperparameters	AntMaze(U)	AntMaze(S)	AntMaze(W)	AntFourRooms	HalfCheetahHurdle
Grid size for representation graph	2	4	4	4	2
Range of k selection	[3,6]	[2,5]	[2,5]	[2,5]	[2,5]
Probability of FR episode p	$0.2 \rightarrow 0.4$	$0.2 \rightarrow 0.4$	$0.2 \rightarrow 0.4$	$0.2 \rightarrow 0.4$	0.1

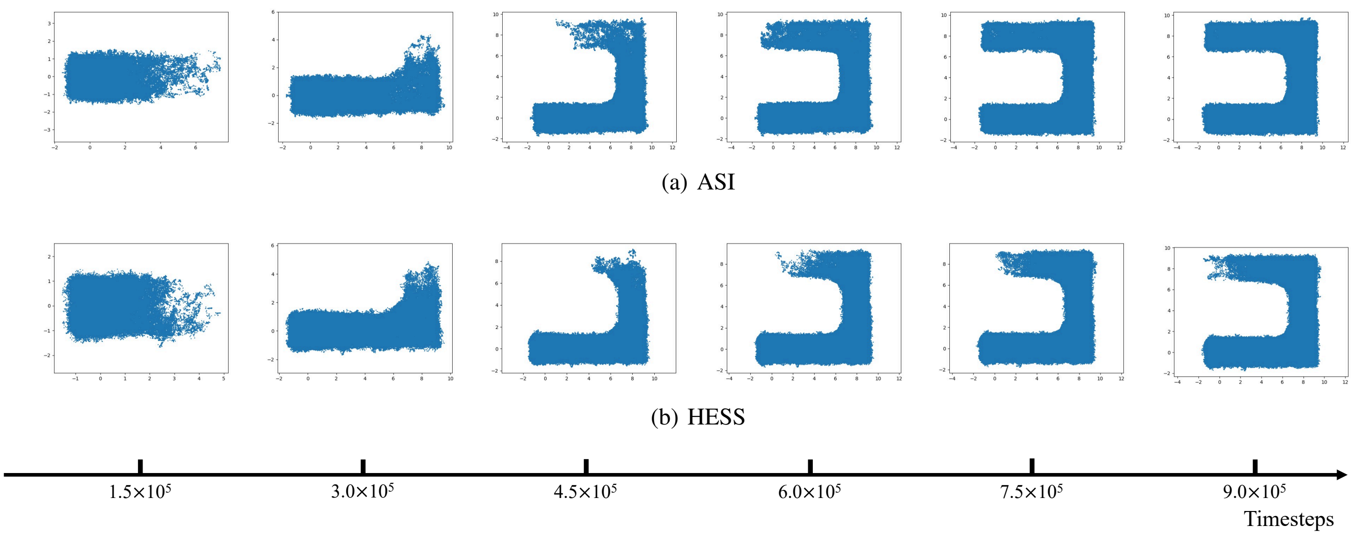


Figure 2: Visualization of coverage ratio in *xy* space. With the frontier-reaching module and the assistance of self-imitation learning, ASI explores faster than HESS.

Algorithm 1: GCHRL with ASI algorithm

Input: High-level horizon c, total training episodes $\mathcal{N}$, graph updating frequency f.
Initialize: High- and low- level policy $\pi_{hi}(sg|s, g; \theta_{hi})$ and $\pi_{lo}(a|s, sg; \theta_{lo})$; representation network $\phi(s)$; high- and low- level replay buffer $\mathcal{B}_{hi}$ and $\mathcal{B}_{lo}$; self-imitation buffer $\mathcal{B}_{imi}$; adjacency graph $\mathcal{M}$.
for $i = 1..\mathcal{N}$ **do**
  Sample episode goal g.
  With a probability of p, set frontier-reaching flag $fr\_flag = True$.
  if fr_flag **then**
    Search frontier g_F and route $(l_1, ..., l_f)$ in $\mathcal{M}$.
  end if
  for $t = 0, 1, ...T - 1$ **do**
    if $t \equiv 0 (\mathrm{mod}\ c)$ **then**
      if fr_flag **then**
        if $\|\phi(s_t) - g_F\|_2 < \epsilon$ **then**
          Start exploration using visit counts.
        else
          Select sg_t according to the FR section.
        end if
      else
        Select subgoal sg_t with any existing method.
      end if
      Collect high-level transition into $\mathcal{B}_{hi}$.
    else
      Determine subgoal sg_t by subgoal transition $sg_t = h(sg_{t-1}, s_{t-1}, s_t)$.
    end if
    Execute $a_t \sim \pi_{lo}(s_t, sg_t; \theta_{lo})$, obtain r_t, s_{t+1}, and collect low-level transition into $\mathcal{B}_{lo}$.
    Collect (s_t, g) into $\mathcal{B}_{imi}$.
  end for
  Update π_{hi} with off-policy DRL algorithms and self-imitation according to Eq. (13).
  Update π_{lo} with off-policy DRL algorithms and self-imitation according to Eq. (14).
  if $i \equiv 0 (\mathrm{mod}\ f)$ **then**
    Update ϕ with Eq. (7), and update $\mathcal{M}$ with Eq. (8).
  end if
end for
Return: π_{hi}, π_{lo}, ϕ.

Listing 1: FR subgoal selection

```
# defined in an agent class
# 'cur' is the current latent state
# 'start' and 'end' is the starting point and the final goal in the latent space
def select_waypoint(self, cur, start, end):
    if start is None or end is None:
        raise ValueError("illegal start or end point")
    # maintain a 'route' attribute within each episode
    if self.route is None:
        # search in the latent graph
        self.route = self.dist_hash.query_route(start, end)
    if self.route.shape[0] == 0:
        return None
    # calculate distances to waypoints
    dist_to_route = np.linalg.norm(self.route - cur, axis=-1)
    # find the nearest
    nearest_idx = np.argmin(dist_to_route)
    # select within a range and add noise
    wp = self.route[min(nearest_idx + random.randint(self.c // 6, self.c // 3), self.route.shape[0]-1)] + np.random.normal(loc=np.array([0, 0]), scale=SELF_IMITATION_NOISE_SCALE*np.array([1, 1]))
    return wp
```

Listing 2: Self-imitation learning (high-level)

```
# defined in an agent class
def update_hi_self_imitation_batch(self, epoch):
    starts_batch, obs_batch, goal_batch = self.self_imitation_buffer.sample(self.args.batch_size)
    # get representations
    goal_repr = self.representation(torch.Tensor(goal_batch).to(self.device)).detach().cpu().numpy()
    obs_repr = self.representation(torch.Tensor(obs_batch).to(self.device)).detach().cpu().numpy()
    starts_repr = self.representation(torch.Tensor(starts_batch).to(self.device)).detach().cpu().numpy()
    # search routes in the graph
    routes = self.dist_hash.query_route_parallel(obs_repr, goal_repr)
    # filter out illegal data
    valid_idx = [i for i in range(len(routes)) if routes[i].shape[0] > 1]
    valid_obs_batch, valid_goal_batch = obs_batch[valid_idx], goal_batch[valid_idx]
    valid_obs_repr = obs_repr[valid_idx]
    # __select_target chooses imitation targets from routes
    valid_targets = np.array([self.__select_target(r)[1] for r in routes if r.shape[0] > 1]) - valid_obs_repr
    # add some noise
    _target_noise = np.random.normal(loc=np.zeros(valid_targets.shape), scale=SELF_IMITATION_NOISE_SCALE*np.ones(valid_targets.shape))
    valid_targets += _target_noise
    target_tensor = torch.tensor(valid_targets).float().to(self.device)
    hi_input_tensor = torch.tensor(np.concatenate([valid_obs_batch, valid_goal_batch], axis=-1)).float().to(self.device)
    # sample actions
    hi_action_batch, _, _ = self.hi_agent.policy.sample(hi_input_tensor)
    hi_imitation_loss = 0.01 * ((hi_action_batch - target_tensor) ** 2).mean()
    return hi_imitation_loss
```

Listing 3: Self-imitation learning (low-level)

```
def update_low_self_imitation_batch(self, epoch):
    num_refs_per_sample = int(self.c // 6)  # number of references
    ''' get obs_repr, goal_repr, starts_repr, routes same as high-level, omitted for layout '''
    valid_idx = [i for i in range(len(routes)) if routes[i].shape[0] > 2]
    valid_obs_batch, valid_goal_batch = obs_batch[valid_idx], goal_batch[valid_idx]
    valid_obs_repr = obs_repr[valid_idx]
    select_info = [self.__select_target(r) for r in routes if r.shape[0] > 2]
    valid_hi_action_for_low = np.array([s[1] for s in select_info])
    # actions to be aligned
    action_to_align = self.low_actor_network(torch.Tensor(valid_obs_batch).to(self.device), torch.Tensor(valid_hi_action_for_low).to(self.device))
    action_to_align = action_to_align.unsqueeze(1).repeat(1, num_refs_per_sample, 1)
    valid_hi_action_refs = np.empty((len(valid_idx), num_refs_per_sample, self.real_goal_dim))
    _cur = 0
    for i in range(len(routes)):
        r = routes[i]
        if r.shape[0] <= 2: continue
        refs_idx = np.linspace(1, max(1, int(min((len(r)-1), select_info[_cur][0]) // 2)), num_refs_per_sample).astype(int)
        valid_hi_action_refs[_cur, :, :] = r[refs_idx, :]
        _cur += 1
    # actions selected under references
    with torch.no_grad():
        valid_obs_for_low_tensor = torch.Tensor(valid_obs_batch).float().unsqueeze(1).repeat(1, num_refs_per_sample, 1).to(self.device)
        valid_hi_action_refs_tensor = torch.Tensor(valid_hi_action_refs).float().to(self.device)
        target_actions = self.low_actor_network(valid_obs_for_low_tensor, valid_hi_action_refs_tensor)
    lo_imitation_loss = 0.001 * ((action_to_align - target_actions) ** 2).mean()
    return lo_imitation_loss
```