Part I

Appendix

Table of Contents

A Formal Analysis of Proposition 3.1 1

B Reward Labeling 1

C Building the Topological Graph 2

D Extended Experiments/Baselines 3

E Miscellaneous Implementation Details 4

F Code Release and Experiment Videos 5

G Environments 5

A Formal Analysis of Proposition 3.1

Proposition 3.1 *If we recover the optimal value function $V^*(s, s')$ for short-horizon goals s' (relative to s), and $\mathcal{G} = \mathcal{S}$ (all states exist in the graph), and the MDP is deterministic with $\gamma = 1$, then finding the minimum-cost path in the graph $\mathcal{G}$ with edge-weights $-V^*(s, s')$ recovers the optimal path. .*

Proof: Let $A(s)$ and $A^h(s)$ define a set of all nodes adjacent to node s and within a short horizon from a node s correspondingly.

The Bellman equation can be used to write the cost of the minimal-cost path, $J^*(s, g)$, in the graph with rewards defined via edge-weights $r(s, a, s') = -V^*(s, s')$:

$$J^*(s, g) = \min_{s' \in A^h(s)} [-V^*(s, s') + J^*(s', g)] = -\max_{s' \in A^h(s)} [V^*(s, s') - J^*(s', g)].$$

We can expand the recursion:

$$J^*(s, g) = -\max_{s' \in A^h(s), s'' \in A^h(s'), \ldots, g \in A^h(s^{(n)})} [V^*(s, s') + V^*(s', s'') + \ldots + V^*(s^{(n)}, g)]. \quad (2)$$

We can further expand each $V^*(\cdot, \cdot)$ term as

$$V^*(s^{(n-k)}, s^{(n-k+1)}) = \max_{\substack{s_1 \in A(s^{(n-k)}) \\ s_2 \in A(s_1) \\ \ldots \\ s^{(n-k+1)} \in A(s_t) \\ t \in \mathbb{N}}} [-C(s^{(n-k)}, s_1) - C(s_1, s_2) - \ldots - C(s_t, s^{(n-k+1)})]. \quad (3)$$

If we expand every term in Equation 2 with 3 it becomes exactly the optimization objective for the shortest path problem with the original edge-weights. One can see $V^*(s, s')$ as a solution to the shortest path problems in the subgraphs of $\mathcal{G}$ induced by $A^h(s)$.

B Reward Labeling

For the base task of goal-reaching, we use a simple reward scheme with a *survival penalty* that incentivizes the robot to take the shortest path to the goal:

$$R_{\text{dist}}(s_t, a_t, g) = \begin{cases} -1 & \forall s_t \neq g \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

For more complex utilities, such as incentivizing driving in the sun (e.g., for a solar robot), we discount the survival penalty by a factor of 4.

$$R_{\text{grass}}(s_t, a_t, g) = \begin{cases} -1 + 0.75 * \mathbb{1}_{\text{grass}}\{s_t\} & \forall s_t \neq g \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

$$R_{\text{sun}}(s_t, a_t, g) = \begin{cases} -1 + 0.75 * \mathbb{1}_{\text{sun}}\{s_t\} & \forall s_t \neq g \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

An interesting implication of the above reward scheme is to view the negative penalty as a proxy for the amount of work a robot needs to do — a solar robot may use 1 unit of energy per time step to navigate in an environment, but it may also create 0.75 units of energy by exposing itself to the sun, effectively discounting the navigation cost in sunny regions. This reward scheme trades-off the choice of the shortest path to the goal with maximizing the user-specified utility function.

For our experiments, we use three different mechanisms to generate these labels:

1. **Fully Autonomous:** In several cases, the reward signal can be easily expressed as a linear/heuristic function of the visual observations. For instance, to obtain labels for "sunny" or "grassy", we process the egocentric images from the robot by thresholding in the HSV colorspace. We process the bottom center crop of the image by thresholding it, and declare event $\mathbb{1}_{\text{sun}}$ or $\mathbb{1}_{\text{sun}}$ if a majority of the pixels satisfy the thresholds.
2. **Manual Labeling:** For more abstract tasks, generating reward labels may require careful hand-labeling at the level of each observation, or each frame. We generate labels for "on-sidewalk" by manually labeling trajectories that are driving on the sidewalk/pavement — this was only feasible because the number of such trajectories was relatively small.
3. **Learned Reward Classifiers:** A desirable hybrid of the above approaches can be constructed where manual labels are queried for a small portion of the training dataset, which can be used to train a simple image classification model. This model can be used to obtain reward labels, albeit noisy, for the remainder of the dataset in a *semi-autonomous* way. We follow this process for obtaining high-quality labels for the "grassy" and "on-pavement" tasks. So long as the reward signal is fully contained by the visual observation, which loosely relates to the Markovian assumption for RL, this method gives us a scalable way to learn a predictive model of rewards.

We note that while the above choice of reward function may seem arbitrary, the overall utility function (or the "relative weight" between the two objectives) would be application-dependent. For instance, a solar-powered robot may be able to recoup 20% of its navigation energy when driving in the sun, and its effective reward could be $(-1 + 0.2 * \mathbb{1}_{\text{sun}})$. We ran experiments to test ReViND's sensitivity to this trade-off and found that it performs expectedly for a wide range of reward functions (see Figure 5). Practically, this would be a hyperparameter set empirically by the user based on the desired level of trade-off between the goal-reaching and utility maximization objectives.

C Building the Topological Graph

As discussed in Section 3.3, we combine the value function learned via offline RL with a topological graph of the environment. This section outlines the finer details regarding how this graph is constructed. We use a combination of learned value function (from Q-learning), spatial proximity (from GPS), and temporal proximity (during data collection), to deduce edge connectivity. If the corresponding timestamps of two nodes are close ($< 2s$), suggesting that they were captured in

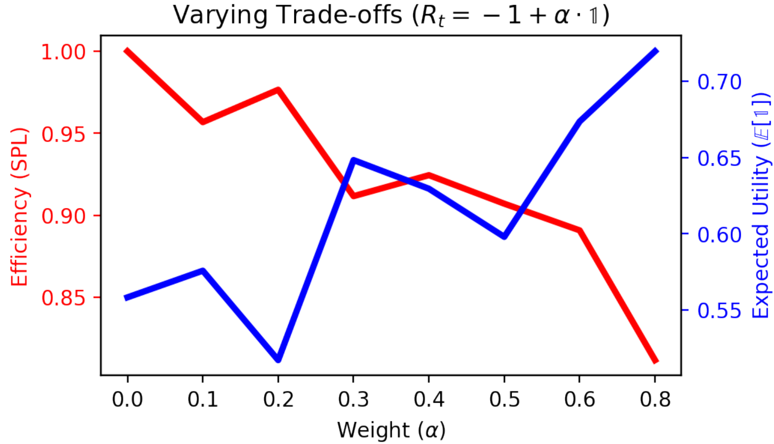

Figure 5: ReViND can support a wide range of reward functions and performs as expected for varying levels of trade-offs between the goal-reaching and utility maximization objectives.

quick succession, then the corresponding nodes are connected — adding edges that were physically traversed. If the distance estimates (or, negative value) between two nodes are small, suggesting that they are *close*, then the corresponding nodes are also connected — adding edges between distant nodes along the same route, and giving us a mechanism to connect nodes that were collected in different trajectories or at different times of day but correspond to the nearby locations. To avoid cases of underestimated distances by the model due to aliased observations, e.g. green open fields or a white wall, we filter out prospective edges that are significantly further away as per their GPS estimates — thus, if two nodes are nearby as per their GPS, e.g. nodes on different sides of a wall, they may not be disconnected if the values do not estimate a small distance; but two similar looking nodes 100s of meters away, that may be facing a white wall, may have a small distance estimate but are not added to the graph in order to avoid *wormholes*. Algorithm 0 summarizes this process — the timestamp threshold ϵ is 1 second, the learned distance threshold τ is 50 time steps (corresponding to $\sim$ 12 meters), and the spatial threshold η is 100 meters.

Algorithm 3 Graph Building

1: **function** GETEDGE(i, j)
2: **Input**: Nodes $n_i, n_j \in \mathcal{G}$; value function V_ψ; hyperparameters $\{\tau, \epsilon, \eta\}$
3: **Output**: Boolean e_{ij} corresponding to the existence of edge in $\mathcal{G}$, and its weight
4: goal = GetRelative(n_i, n_j) ▷ using GPS and compass
5: $D_{ij} = -V_\psi(n_i, \text{goal})$ ▷ learned distance estimate
6: $T_{ij} = |n_i[\text{'timestamp'}] - n_j[\text{'timestamp'}]|$ ▷ timestamp distance
7: $X_{ij} = \|n_i[\text{'GPS'}] - n_j[\text{'GPS'}])\|$ ▷ spatial distance
8: **if** ($T_{ij} < \epsilon$) **then** return $\{$*True*, $D_{ij}\}$
9: **else if** ($D_{ij} < \tau$) AND ($X_{ij} < \eta$) **then** return $\{$*True*, $D_{ij}\}$
10: **else** return *False*

Since a graph obtained by such an analysis may be quite dense, we perform a *transitive reduction* operation on the graph to remove redundant edges.

D Extended Experiments/Baselines

This section presents a detailed breakdown of the quantitative results discussed in Section 4.3. We evaluate ReViND against four baselines in 15 environments with varying levels of complexity, in terms of environment organization, obstacles, and scale. Tables 2 and 3 summarize the performance of the different methods for the task of maximizing the R_{grass} and R_{sun} utilities, respectively.

Method	Easy (<50m)		Medium (50–150m)		Hard (150–500m)	
	Success	$\mathbb{E}\mathbb{1}_{\text{sun}}$	Success	$\mathbb{E}\mathbb{1}_{\text{sun}}$	Success	$\mathbb{E}\mathbb{1}_{\text{sun}}$
Behavior Cloning	1/5	0.58	0/5	0.32	0/5	0.29
Filtered BC	3/5	0.51	0/5	0.31	0/5	0.32
IQL [8]	3/5	0.54	2/5	0.42	0/5	0.34
ViNG [7]	**5/5**	**0.63**	**4/5**	0.58	3/5	0.63
ReViND (Ours)	**5/5**	0.61	3/5	**0.75**	**4/5**	**0.74**

Table 3: Success rates and utility maximization for the task of navigation in sunny regions (R_{sun}).

We see that ReViND is able to consistently outperform the baselines, both in terms of success as well as its ability to maximize the utilities $\mathbb{1}_{\cdot}$. In particular, we see that ReViND's performance closely matches that of IQL in the easier environments, where the system does not need to rely excessively on the graph. However, the real prominence of ReViND is evident in the more challenging environments, where it is consistency successful while also maximizing the chosen utility. As the horizon of the task increases, the search algorithm on the graph returns more desirable paths that may stray from the direct, shortest path to the goal, but are highly effective in maximizing the utility. We also note that ViNG, which uses a similar topological graph, is statistically similar to ReViND in terms of its goal-reaching ability; however, since it does not support a mechanism to customize the behavior of the learned policy, it suffers in the other performance metrics. BC, fBC and IQL consistently fail to reach goals beyond 15-20m away, due to challenges in learning a useful policy from offline data — these "flat" policies often demonstrate *bee-lining* behavior, driving straight to the goal, which leads to collisions in all but the easiest experiments.

E Miscellaneous Implementation Details

Table 4 presents the neural network architectures used by our system. We provide the important hyperparameters for training our system in Table 5. The underlying learning algorithm in ReViND is based on IQL [8], and we encourage the reader to check out the IQL paper for more implementation details.

Layer	Input Shape	Output Shape	Layer details
1	[3, 64, 48]	[1536]	Impala Encoder [48]
2	[1536]	[50]	Dense Layer
3	[50]	[50]	`tanh` (LayerNorm)
4	[50], [3]	[53]	Concat. image & goal
	Policy Network $a_t \sim \pi(s_t)$		
5	[53]	[256]	Dense Layer
6	[256]	[256]	Dense Layer
7	[256]	[10]	Dense Layer
	Q Network $Q(s_t, a_t)$		
5	[53], [10]	[256]	Dense Layer
6	[256]	[256]	Dense Layer
7	[256]	[1]	Dense Layer
	Value Network $V(s_t)$		
5	[53]	[256]	Dense Layer
6	[256]	[256]	Dense Layer
7	[256]	[1]	Dense Layer

Table 4: Architectures of the various neural networks used by ReViND.

Hyperparameter	Value	Meaning
τ	0.9	IQL Expectile
A	0.1	Policy weight
γ	0.99	Discount factor
η	0.005	Soft Target Update
$\alpha_{\text{actor}}, \alpha_{\text{critic}}, \alpha_{\text{value}}$	$3e-4$	Learning rates

Table 5: Hyperparameters used during training ReViND from offline data.

Figure 6: Example egocentric observations from the training dataset [9] (top) and the deployment environments (bottom), including the predicted labels for the "sunny" reward.

F Code Release and Experiment Videos

We are sharing the code corresponding to our offline learning algorithm, labeling, and evaluation scripts, as well as experiment videos of ReViND deployed on a Clearpath Jackal mobile robotic platform. Please check out the project page for further information: sites.google.com/view/revind.

G Environments

We train ReViND using 30 hours of publicly available robot trajectories collected using a randomized data collection procedure in an office park [9]. We conduct evaluation experiments in a variety of novel environments with similar visual structure and composition as the training environments — i.e. suburban environments with some traversals on the grass, around trees of a certain kind, and on roads. While it may be extremely challenging to get generalization capabilities that work for *all* scenarios, we demonstrate that ReViND can learn behaviors from a small offline dataset and generalize to a variety of previously unseen, *visually similar* environments including grasslands, forests and suburban neighborhoods. Figure 6 shows some example environments from the training and deployment environments, along with their corresponding labels (automatically generated).