Combining Graph-Based Planning and Deep Reinforcement Learning

Travis Hammond
Faculty of Science and Engineering
University of Groningen
Leipzig, Germany
t.hammond@student.rug.nl

Dr. Davide Grossi
Faculty of Science and Engineering
University of Groningen
Groningen, Netherlands
d.grossi@rug.nl

Dr. Matthia Sabatelli
Faculty of Science and Engineering
University of Groningen
Groningen, Netherlands
m.sabatelli@rug.nl

***Abstract*— Historically, planning and reinforcement learning have had similar goals but have lived in a dichotomy of research fields, having complimentary strengths and weaknesses but with no way of combining them in a fundamental way. Motivated by recent attempts to combine the fields of graph-based planning and deep reinforcement learning, we provide an overview of the methods involved and experimentally show how this approach provides a number of quantitative benefits in terms of performance on in-distribution- as well as on out-of-distribution- challenges and qualitative benefits in terms of model interpretability as well as the ability to manually edit this model, allowing to incorporate domain-knowledge into the trained model. To achieve this, we contribute to the Candle deep learning framework and thus to the overall reinforcement learning ecosystem of the Rust programming language.**

***Index terms*—reinforcement learning, machine learning, deep learning, actor-critic, planning, graph theory, rust language**

I. Introduction

Reinforcement learning (RL), especially deep reinforcement learning (DRL), has demonstrated the ability to learn optimal policies from raw, high-dimensional input data such as images without requiring any domain knowledge [1], [2], but it has consistently shown its success to be limited to short-horizon tasks. The further away the goal is, quantified in terms of how many actions and timesteps are required to reach it, the harder it becomes to train RL algorithms to be successful in reaching this goal. This is evident in the performance of RL algorithms over time across the famous ATARI benchmark [1]: the games that required long-distance planning took the longest time to be solved. Furthermore, as is generally the case with neural-networks, these algorithms return black-box policies which are difficult or even impossible to interpret which is a problem that becomes more important the more safety or business critical the deployment of that policy becomes [3].

Classical graph-based planning on the other hand has converged to efficient, correct and provably optimal search algorithms that easily find solutions to long-horizon tasks as early as 1956 with algorithms such as Dijkstra's [4] and later A* [5]. However, these algorithms require the environment to be defined in terms of a graph in which the possibly raw, high-dimensional state space of the real world is abstracted away into a set of nodes and the transition dynamics are abstracted away into a set of edges connecting these nodes. While this graph then lends itself nicely to the analysis and interpretation of resulting plans, it has to be handcrafted with domain-specific knowledge and the quality of this graph has a deciding influence on the quality of the solution. Further, actually executing this plan still requires a controller with the ability to select low-level actions in the real world to transition between these abstract nodes towards the goal.

It seems as though these fields attempt to achieve the same goal and their solutions to the problem complement each other quite nicely in terms of their strengths and weaknesses. Having started from different assumptions, the fields of planning and reinforcement learning have developed their own methodologies and then found each other in a field that become known as model-based reinforcement learning [6] at least as early as 1990 [7]. In this thesis we focus on one particular approach to combining the fields of deep reinforcement learning and graph-based planning by means of hierarchical, model-based RL in which a graph-based representation of the environment is built up over time from past experiences and a non-parametric high-level controller (i.e. a graph-search algorithm) creates plans consisting of short-horizon sub-goals which are delegated to a low-level controller (i.e. a reinforcement learning agent) [8], [9].

We first provide an introduction to the field of deep reinforcement learning in Section II, giving a primer on each of the concepts necessary to understand the final algorithms,

as well as a quick introduction to graphs and the most relevant graph-based planning algorithms and their performance characteristics in Section III. In Section IV we provide an overview of how different works so far have tackled this idea and an in-depth summary of the most important papers that have been published on this particular approach building up to [8] and [9]. We then include an additional section on the role of the Rust programming language in our work in Section V, as we did spend a considerable amount of effort promoting the field of deep reinforcement learning in the ecosystem of this young but promising language, including an open-source contribution to a major machine-learning framework.

Finally, we provide an implementation of this graph-based reinforcement learning approach in Rust, based on the implementations by B. Eysenbach, R. Salakhutdinov, and S. Levine [8] and M. Laskin, S. Emmons, A. Jain, T. Kurutach, P. Abbeel, and D. Pathak [9]. We show that the graph-based approach provides a number of quantitative benefits in terms of performance on increasingly difficult (in terms of longer horizon) environment settings as well as the performance on more difficult settings after training on easier ones, showcasing the ability of zero-shot generalization to- and thus robustness in the face of- slightly different environments. We also show a number of qualitative benefits such as the interpretability of the policy due to the ability to inspect the graph and the current plan, noting that the graph can even be edited by hand if necessary, as well as the flexibility of the trained algorithm to reach any given goal due to the goal-conditioned nature. For this, we introduce our methods in Section VI and show our results in Section VII. We finish with a discussion and a note on future work in particular on aspects we did not cover in this thesis due to time constraints in Section VIII.

The main research questions we wish to answer here are 1) can we combine the strengths of graph-based planning and deep reinforcement learning and 2) does this provide us with quantitative benefits in terms of model performance as well as 3) qualitative benefits in terms of model interpretability?

II. Reinforcement Learning

Reinforcement learning can be loosely defined in terms of a few key components. We define the decision maker as the *agent*, who selects *action*s based on *observation*s and *reward*s provided by the *environment*. The environment in turn, comprising everything outside of the agent (including, for example, its sensors), receives these actions and modifies its underlying state in some way, returning a new set of observations and rewards. It is the job of the agent to learn and adapt its behavior, or *policy*, in such a way as to maximize

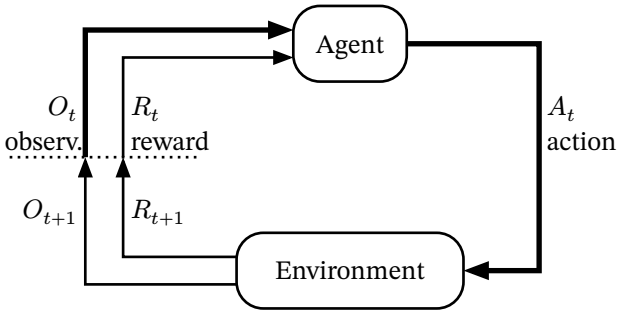

Figure 1: The agent-environment loop. The dotted line indicates where the loop begins, bold lines indicate the initial flow of data. The first observation O_t (maybe from a newly initialized environment) is passed to the agent which responds with an action A_t. The environment then responds with a new observation O_{t+1} and reward R_{t+1} which are passed to the agent, and the process is repeated. After the first timestep, the agent uses the reward R_{t+1} to update its behavior (R_0 does not exist).

the total reward over the full length of an episode (which can possibly continue forever). See Figure 1 for a diagram visualizing this coupled, dynamic process [10].

A. Markov Decision Process

We can more strictly define these concepts in terms of a Markov Decision Process (MDP) [11], a mathematical framework which helps formalize the notion of maximizing reward by sequential decision making in a dynamic environment that is influenced by the actions taken [10].

In this idealized formalization of the MDP we will often talk about *state*s (S_t), which represent the true underlying state of the environment. What the agent in fact receives are *observation*s (O_t) which are not necessarily the same as the true underlying states for many possible reasons, e.g.:

- An embodied agent is in some true state in the real world, but its observations are distorted by imperfect sensors.
- A virtual agent receives an observation from a game showing an empty room, but the true state includes a key located off-screen.

In fact, an MDP is usually defined in terms of the true underlying state because many of its mathematical proofs relies on the state having the *Markov Property*, which says that the state must include information about all aspects of the past agent–environment interaction that make a difference for the future [10]. This obviously does not reflect reality (see the two examples above) and there are certain ways to attempt to reconcile the reality of the partially observable world with the theoretical reliance on the Markov property but these are outside the scope of this thesis and from here on we will use the terms *state* and *observation* interchange-

ably, excused by the fact that in the environments we investigate the observations do in fact coincide with the true states.

We define the agent and environment interactions to take place at discrete time steps ($t = 0, 1, 2, 3, ...$). At each time step t, the agent receives a state S_t which is sampled from the state space $\mathcal{S}$ on the basis of which it chooses an action A_t from the actions space $\mathcal{A}$. This action affects the environment in some way that changes the underlying state and the corresponding numerical reward signal to produce S_{t+1} and $R_{t+1} \in \mathbb{R}$, where the reward is a function of the state-action-next-state triple $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. These interactions produce a *trajectory*:

$$S_0, A_0, R_1, S_1, A_t, R_2, S_2, A_2, R_3, ... \tag{1}$$

which unrolls into a sequence of *transitions*:

$$\begin{aligned} S_0, A_0, R_1, S_1 \qquad & \text{Transition 1} \\ S_1, A_1, R_2, S_2 \qquad & \text{Transition 2} \end{aligned} \tag{2}$$

In a *finite* MDP that satisfies the Markov property, in which the sets of states, actions and rewards ($\mathcal{S}, \mathcal{A}, \text{and } \mathbb{R}$) each contain a finite number of elements, the dynamics of the environment can be fully described by a probability distribution that depends only on the previous state and action:

$$p(s', r|s, a) \stackrel{\text{def}}{=} \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\},$$

for all $s', s \in \mathcal{S}$, $r \in \mathbb{R}$, and $a \in \mathcal{A}$ [10]. This restriction on each state to include any and all information that affects the next state and reward is another way of formulating the Markov property.

B. Episodic vs Continuous Tasks

Episodic, or sometimes called *finite-horizon*, tasks are those that end in a final time step T, where T is a finite, random variable, while *continuous* tasks are those that don't end or at least not in any meaningful timelimit. This makes a difference when reasoning about the behavior of different algorithms because, as we will discuss in Section II.D, we often talk about the agent maximizing the total reward and in an infinite timeframe the total reward can itself easily become infinite. So to simplify the math, or to reason about these situations, we either consider finite horizon tasks or infinite horizon tasks with *discounted* rewards [10].

C. Discrete vs Continuous Action Spaces

Tasks can also be differentiated in terms of the domain of the action space $\mathcal{A}$, which could consist of a discrete, finite set of actions $A_t \in \{A_0, A_1, ...A_n\}$ at each timestep, a continuous range $A_t \in [A_{\min}, A_{\max}]$. As we can imagine, tasks with a continuous action space are typically more difficult to learn and require different kinds of algorithms. In this thesis we will consider the case of continuous action spaces.

D. Returns & Rewards

We can formulate what we wish the agent to achieve by designing a suitable reward function. This reward function is unknown to the agent and therefore part of the environment [10]. The goal of the agent is to maximize the *expected return* which, in the simplest case and for *undiscounted* and *episodic* environments, is simply the *expected value* of the *cumulative sum* of rewards from any given starting state:

$$G_T \stackrel{\text{def}}{=} R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{3}$$

where G_T is the return at timestep T. Note that we omit the expectation operator for simplicity. An episode is a sequence of agent-environment interactions that ends when the environment transitions into a *terminal state*, $S_T \in \mathcal{S}^+$, which is defined as a state which transitions only into itself and which provides no reward. Practically, when the environment reaches a terminal state (regardless of whether this means it was successful or not) it resets and a new episode begins, providing some initial observation O_0 and no reward, from which the agent may again attempt to maximize its return.

In the case of *continuous* environments, we can define the expected return to be the cumulative sum of *discounted* rewards:

$$G_T \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

where $\gamma \in [0, 1)$ is the *discount rate* that determines how much future rewards are discounted, that is, how much less a future reward is worth at the present compared to an immediate reward of the same value. To be exact, a reward R_k received k time steps from the present is now only worth $R_k \cdot \gamma^{k-1}$. For example, when $\gamma = 0$ then only the immediate next reward R_{t+1} is important to the agents goal, so it optimizes its policy greedily to pick actions A_t only such that R_{t+t} is maximized. On the other hand when γ approaches 1, the agent becomes more aware of the value of potential future rewards which might lead to greater total rewards than the greedy perspective.

We can see how this term helps us define the notion of discounted future rewards, but this also helps us deal with any potential infinite rewards: as long as $0 \leq \gamma < 1$, this term has a finite value.

Because we defined the terminal state to be a special state

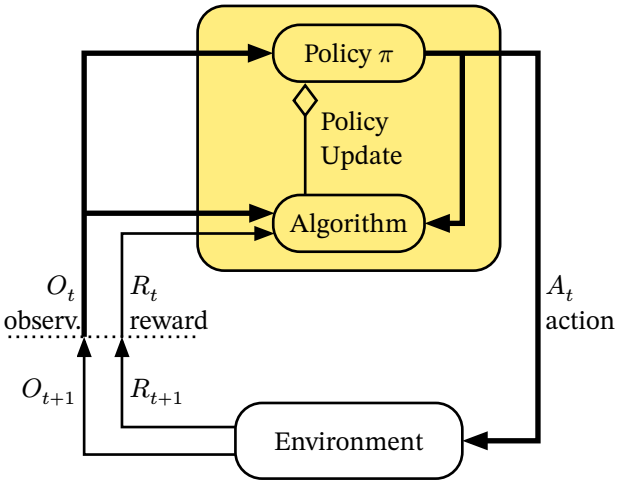

Figure 2: The same agent-environment loop as in Figure 1, but here the agent is expanded to show that it consists of a policy π which defines the actions A_t to take, given an observation O_t and a learning algorithm which defines how the policy is updated as a consequence of the resulting observation O_{t+1} and reward R_{t+1}.

that transitions only into itself and generates only rewards of zero, we can combine both the episodic as well as the continuous tasks into a single notation:

$$G_T \stackrel{\text{def}}{=} \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{5}$$

where either $T = \infty$ or $\gamma = 1$, but not both. The case of both continuous and undiscounted tasks is out of the scope of this thesis.

E. Policies & Value Functions

To achieve the aim of maximizing the expected return, as we defined it in the previous section, the agent will need to adapt its behavior, or *policy*, based on the feedback it receives. A reinforcement learning algorithm specifies how the policy is updated as a result of the experiences collected [10]. Figure 2 shows the distinction between the algorithm and the policy. A policy, denoted π, is defined as a probability density function over the action space for every possible state. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$ [10].

Reinforcement learning algorithms distinguish themselves, among other aspects, from planning methods by learning a *global* solution over all states, not just those states relevant to the solution [6], and there are two common ways to formalize this: As a *value function* $v_\pi(s)$ which is defined over states or as an *action-value function* $q_\pi(s,a)$ which is defined over state-action pairs. For MDPs, these can be formally defined:

$$\begin{aligned} v_\pi(s) &\stackrel{\text{def}}{=} \mathbb{E}_\pi[G_t \mid S_t = s] \\ &\stackrel{\text{def}}{=} \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \end{aligned} \tag{6}$$

$$\begin{aligned} q_\pi(s,a) &\stackrel{\text{def}}{=} \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &\stackrel{\text{def}}{=} \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \end{aligned} \tag{7}$$

for all $s \in \mathcal{S}$.

We can see that for discrete action spaces, the (so-far) optimal policy can be derived e.g. from $q_\pi(s,a)$ by taking the max over the set of possible actions from a given state while for continuous action spaces this is not straightforward and requires a different approach we will discuss in Section VI.

F. Online vs Offline

The concepts of *offline* and *online* algorithms in general, or those that fall somewhere in between, are not specific to reinforcement learning but they help characterize a certain aspect of the learning setting so that it helps to define these terms to fully understand the algorithm we are building towards.

In the case of strictly *online* learning algorithms, the algorithm updates and improves with data as it is made avail-

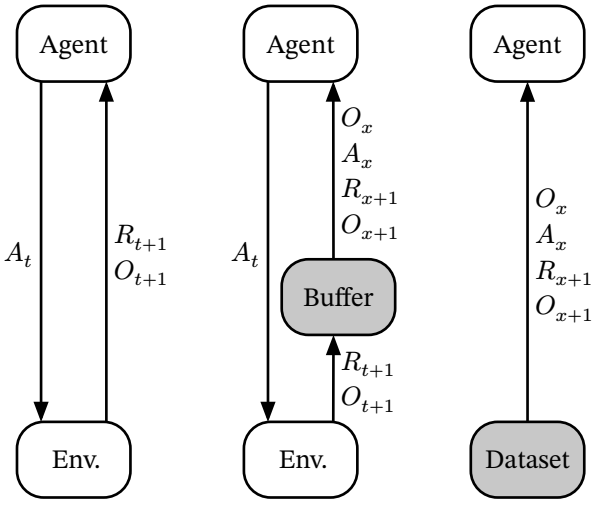

Figure 3: Conceptual differences between purely online and purely offline learning algorithms in the context of reinforcement learning. Here, x denoted a sampled timestep independent of the current timestep t. Purely online algorithms take actions and directly use the resulting observations and rewards from the environment resulting in a tight coupling. Purely offline algorithms learn a policy from a static dataset by sampling transitions. Replay buffer methods sit between the two by interacting with the environment but sampling transitions from a dynamic buffer.

able and then immediately throws away that data. These algorithms have a poor sample efficiency (as of course they use every sample only once), but they tend to handle non-stationary tasks better because they can "forget" about older examples and adapt to newer ones [10].

On the contrast, strictly *offline* learning algorithms operate on the (static) dataset as a whole and would need to be retrained from scratch if any part of that dataset changes. This is an active area of research, as we can imagine the challenges that arise from the agent not being able to explore as it learns but also the benefits of being able to utilize the massive amounts of already collected, static datasets that we have and not have to rely on simulations to train real-world reinforcement learning agents [12].

An algorithm somewhere between the two extremes can be constructed, for example, from an *online* algorithm and a buffered dataset that stores the last N datapoints. This leads to improved stability during the training process, especially for the case of neural networks as this helps the validity of the i.i.d assumption. It also increases the sample efficiency because datapoints are now reused multiple times. Many algorithms are chosen in-between these extremes by incorporating a so-called *replay buffer* [2], which is exactly a buffered dataset of collected transitions, or *experiences*, of a fixed size where the oldest experiences are forgotten as new ones come in when the buffer is full. In the learning step, all datapoints currently in the buffer are sampled from, either uniformly or with some prioritization, and they can even be relabeled as is the case in Hindsight Experience Replay (HER) [13]. This technique in general is often called *experience replay* and this replay buffer will play a central role in our final algorithm.

G. On-policy vs Off-policy

In reinforcement learning, we can optionally differentiate between the policy being optimized and the policy that is being used to collect data. That is, the actions chosen by the agent in the environment don't have to be sampled from the same policy that is being updated in the learning step.

In the simplest example, the agent might always choose a random action, completely disregarding any observations, but still learn, i.e. update an internal policy, from incoming experiences. We might be able to deploy this agent after training and obtain better rewards than if we would deploy the random policy.

To be more concrete, reinforcement learning algorithms differ in terms of whether they are *on-policy* or *off-policy* [10]. In the case of *on-policy* algorithms, they are updating their so-called *target policy*, usually denoted by π, with data collected using that same policy. In the more general case of *off-policy* algorithms, the data collection policy of the agent, also called the behavior policy and denoted by b, during the training process is different to the policy that is being optimized. In our simple example, the behavior policy b is a simple uniform sampling process of the action space and the target policy π is being optimized.

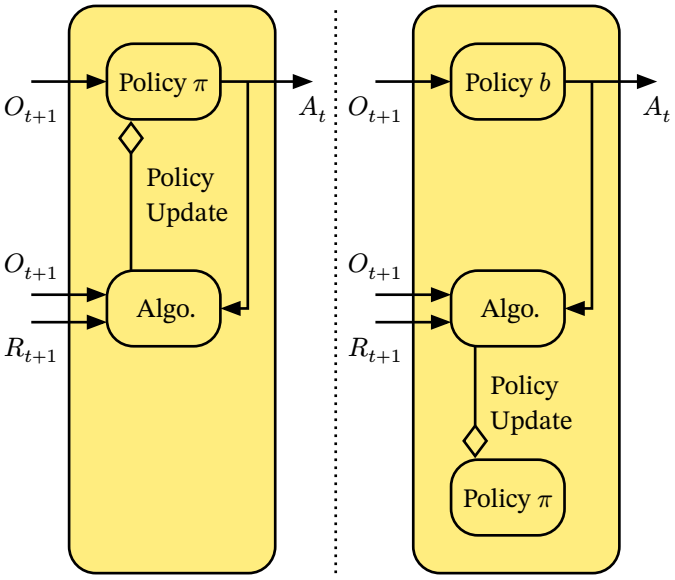

Figure 4: The difference between on-policy (left) and off-policy (right) algorithms.

Off-policy algorithms are a strict generalization of on-policy algorithms as any off-policy algorithm can trivially be made on-policy be setting $b = \pi$. This can also be done to some variable degree as is the case with the ε-greedy policy which samples from the random policy $\varepsilon\%$ of the time and from the target policy otherwise. The use of a replay buffer typically makes an algorithm off-policy because the sampled transitions from which the policy is updated come from older timesteps during which the policy was possibly different, and offline reinforcement learning (Section II.F) is considered to be "fully" or "pure" off-policy [12].

H. Model-Based Reinforcement Learning

Model-based reinforcement learning is an umbrella term meaning to capture all algorithms that learn a model of the environment to which it has *reversible* access [6]. The distinction between *reversible* and *irreversible* access to a model arises from the fact that a model-free reinforcement learning algorithm can, and usually does, implicitly learn a model of the environment (for example by learning the environment dynamics) but it is not able to repeatedly plan forward from the same state like humans might when pondering the different consequences of taking different actions (in the sense that we can mentally reverse back to the original state and take a different action). Instead, it must take an action and observe the consequences. Model-based reinforcement learning algorithms in contrast, have reversible access to the model of the environment and can use that access to repeat-

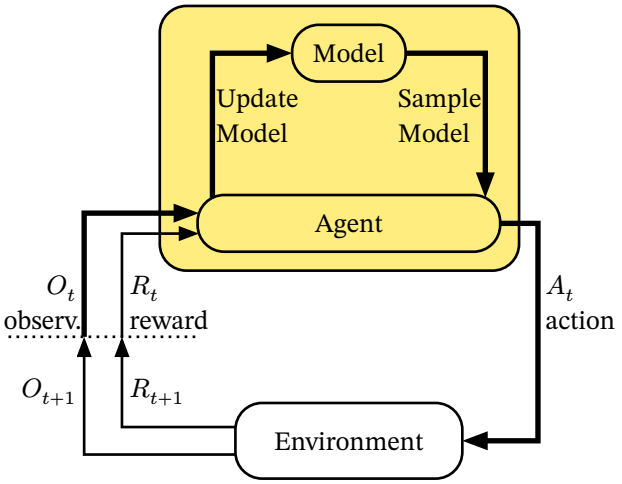

Figure 5: Showcasing an agent with reversible access to a world model. The agent can explicitly update the model based on transitions it takes within the environment, and it can sample (synthetic) transitions from the model by taking repeated "imaginary" actions from any state in the model before actually taking some action A_t in the environment.

edly plan from any state as often as it wishes before actually taking an action.

It is a good idea to remind ourselves that learning can now happen in locations in the algorithm: While learning the policy and while learning the model. This means that we can either learn the model or provide a known model, and we can either learn a policy or provide a rule-based algorithm (See Table 1). The combination of a known model and a rule-based algorithm is referred to as *planning* and will be discussed in the next section. The combination of learning a model and providing a known policy is not strictly considered model-based reinforcement learning [6] since it does not learn a policy (i.e. a global solution), but the other two combinations are.

Model Learned	Policy Learned	Example
+	+	Dyna [10]
-	+	AlphaZero [14]
+	-	Embed2Control [15]
-	-	A* [5]

Table 1: The possible combinations of learning with reversible access to a model. Note that only those methods where the (global) policy is learned are technically considered reinforcement learning. This table is gathered from content provided by T. M. Moerland, J. Broekens, and C. M. Jonker [6].

Having reversible (or even irreversible) access to a previously unknown model can provide a number of benefits such as sample efficiency due to the ability to sample transitions from the model before taking actions (this is especially relevant in real-world situations such as robotics), explicit exploration strategies based on e.g. model uncertainty, and interpretability of the learned policy by being able to inspect the learned model. Model-based methods can also have better transfer performance to slightly different environments by being able to reuse the learned model [16].

But model-based RL also comes with a number of challenges, the biggest of which is model uncertainty making the sampled transitions inaccurate especially while the model is being learned. In practice, this leads to model-based methods where the model is learned usually having a lower asymptotic performance. Another important challenge is that model-based methods usually have a higher number of hyperparameters that need tuning such as when and how often and for how many steps to plan from the model, etc. Hyperparameter tuning is already a challenging aspect of RL in general.

I. *Hierarchical Reinforcement Learning*

In the definition of reinforcement learning so far, especially considering the MDP as the building block upon which the algorithms are built, we typically consider low-level, atomic actions executed at a high-frequency [6]. This highly detailed view of the agent-environment interaction dynamic typically results in long-horizon tasks, where a large number of actions need to be taken until a reward is received and this is makes it hard to correctly assign credit and in consequence, learn the solution. The idea of temporal abstraction over the action space to mitigate this is known as hierarchical reinforcement learning (sometimes considered a subset of model-based RL), and typically distinguishes between low-level controllers that deal with actions on the atomic level but only need to reach short-horizon goals and high-level controllers that deal with actions on a higher

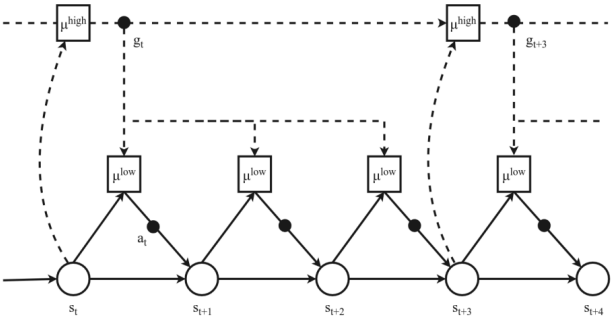

Figure 6: Diagram of Hierarchical RL, copied from [6], where a high-level agent μ^{high} picks high-level actions (goals) g_t for the low-level controller μ^{low} to reach, and the low-level controller picks atomic actions a_t to take which transitions the environment to the next state s_{t+1}.

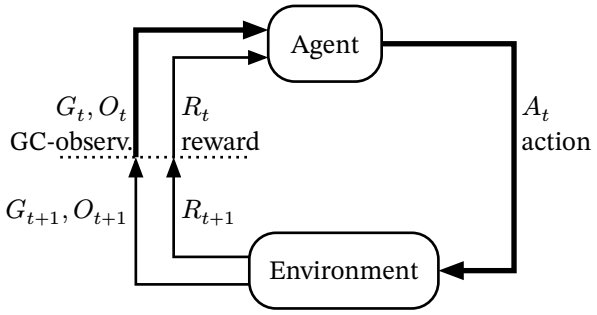

Figure 7: The same agent-environment loop as in Figure 1, but the observation space is augmented by the current goal that should be reached G_t. This simple generalization over the goal space can provide a temporal abstraction and as a consequence, a hierarchical reinforcement learning agent.

level of abstraction and produce plans consisting of series of short-horizon sub-goals.

One popular such attempt, and one we will focus on, are *goal-conditioned value-functions*, also known as universal value function approximators [17] (dating back to Feudal RL [18]) which uses a goal space $\mathcal{G}$ as the abstract action space. The implementation of this is quite straightforward. Consider that we can extend the classic reinforcement learning setting that we introduced in Section II.A to a more general goal-parameterized setting in which the agent is provided not just an observation and a reward signal, but also a goal which it must reach (Figure 7). The reward signal is then also goal-parameterized, producing different values depending on the current goal. The standardized API for this task setting as proposed by [19] is for the environment to recieve an action and to return a 3-tuple of an *achieved goal*, which is the current state of the agent, the *desired goal*, which is the state the agent should aim for as it would produce the successful reward, and the *observation*, which includes the information contained in achieved goal as well as providing additional information about the environment (e.g. walls etc).

This is formalized by extending our definitions of the value function or action-value function from Section II.E by a goal $g \in \mathcal{G}$ resulting $v_\pi(s, g)$ or $q_\pi(s, a, g)$. This is a strictly more general problem, making the agent more flexible by being able to reach different goals without the need for retraining but it also makes the problem more difficult to learn, especially on long-horizon goals. However, the agent now only needs to learn to reach short-horizon goals allowing a higher-level agent to generate suitable sub-goals. Note that this approach generalizes the temporal abstraction of the action space in some sense by how fine-grained the sub-goals are selected, and in the extreme case the next sub-goal is simply the goal. An important question here is how to discover the most relevant sub-goals, i.e. how to define the best level of abstraction. This idea of training goal-conditioned policies as a method of temporal abstraction for long-horizon tasks has been shown to be successful, for example in the case of GoExplore [20].

III. Graph-Based Planning

Planning is the process of modeling a problem and then *searching* for a plan using that model. Historically, the task of modeling the problem has been the task of human domain experts handcrafting models in e.g. an action language, propositional logic, a graph or an MDP. As opposed to reinforcement learning methods, planning methods focus on generating plans from a model that they have *reversible* access to (see Section II.H) and they do not necessarily concern themselves with learning, i.e. in planning we mostly consider the model as known and algorithmically search for good solutions.

Another distinction between planning and reinforcement learning methods, that arises from the difference in assumption of whether the model is known, is that planning methods are typically only concerned with a *local* representation of the solution (i.e. the actions and states between the current state and the goal) and reinforcement learning methods attempt to learn a *global* solution (i.e. a value function or action-value function over all states, as described in Section II.E) [6]. See Table 2 for an overview of the differences between model-based RL, model-free RL, and planning.

	Model	**Global Solution**
Planning	+	-
Model-Based RL	+	+
Model-Free RL	-	+

Table 2: The distinction between planning and model-based or model-free reinforcement learning based on whether the method has *reversible* access to a model (known or learned) and whether the method learns a *local* or a *global* solution. This table is reproduced from T. M. Moerland, J. Broekens, and C. M. Jonker [6].

By being able to assume reversible access to a high-quality model, very efficient and provably correct search algorithms have been devised to find optimal plans. In this thesis, we are particularly concerned with graph-based search.

A. Graphs

As we said in the introduction, in the end we want our agent to make use of a high-level (i.e. more temporally abstract in the hierarchical sense as discussed in Section II.I instead of at the lowest level of atomic actions) graph data structure as a model of the environment that it builds up as it explores and interacts with the world. This graph should have edges which are associated with a certain cost of tra-

versal, as we can imagine that some edges can take longer to traverse than others. It should also be directed, as it might be easy to traverse an edge in one direction and impossible to traverse it in the opposite direction, for example. Let us first define this data structure and then consider some search algorithms that operate on it.

We specifically consider a weighted, directed graph, which is defined by a triple of disjoint sets $G = (V, E, w)$, such that V is the set of *vertices*, also called nodes, E is the set of *edges* which are each an ordered 2-element subset of V defining which two vertices are *joined* by the edge (e.g. $x, y \in V$ joined by $xy \in E$), and w is the *weighting function* which defines the *weight* of an edge (e.g. $w(xy) = 0.5$), which we interpret as the cost of traversing that edge. Because we specifically consider *directed* graphs in this thesis, the ordering of the vertices joined by edges matters, i.e. $xy \in E \neq yx \in E$. An edge can join a vertex with itself $xx \in E$, this is called a *loop*, but we do not consider parallel edges (multiple edges connecting the same vertices in the same direction). This definition has the consequence that the manner in which the graph is drawn is irrelevant, only the information about which vertices exist and how they are joined by edges is important [21].

Two vertices that are joined by an edge are *neighbours*. A vertex with no neighbors is *isolated*. The number of neighbors of a vertex is also called the *degree* of that vertex, and this allows us to consider the *minimum degree* of a graph as well as the *maximum degree* or the *average degree* as the minimum, maximum or average number of neighbors of all of the vertices in V. See Figure 8 for an example of a directed graph.

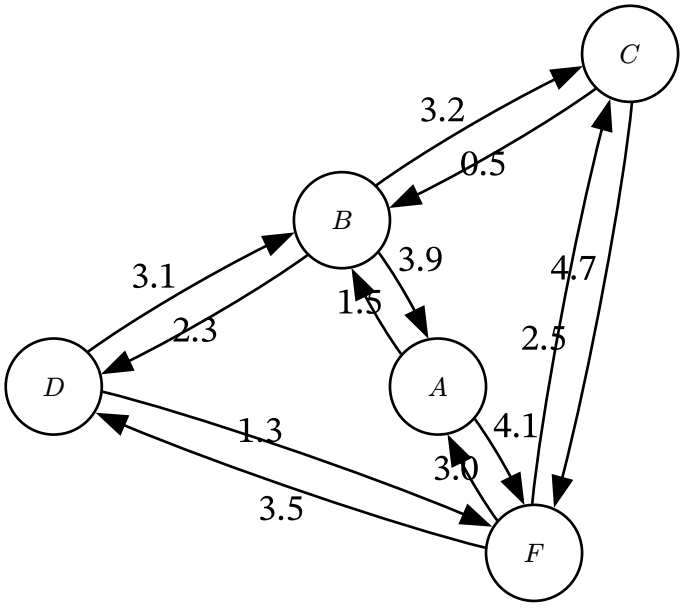

Figure 8: An example graph with 5 nodes.

A path is an ordered list of distinct and sequentially joined vertices, *linking* the first and the last vertices. The number of edges in this path, or equivalently one less than the number of vertices in this path, is called the *length* of the path. The sum of the weights of the edges in the path is called the *distance*- or the *cost* of traversing between- the first and the last vertices. A *cycle* is a cyclical path, that contains an additional edge from the last to the first vertex.

B. Dijkstra's Algorithm

Dijkstra's algorithm [4] is a single-source shortest path algorithm, meaning it returns the shortest paths from a single source vertex $v \in V$ to all vertices $u \in V$. The time complexity of Dijkstra's algorithm depends mainly on the data structure used to represent the set Q (see Listing 1), in particular it depends on the complexities of the *decrease-key* (T_{dk}) and *extract-minimum* (T_{em}) operations of that data structure: $O(|E| * T_{\text{dk}} + |V| * T_{\text{em}})$. The simplest implementation is backed by an array or linked list for vertices and an adjacency list or matrix for edges and has a time complexity of $O(|E| + |V|^2)$. The most efficient implementation is backed by a Fibonacci Heap (which is a priority queue data structure) and has a time complexity of $O(|E| + |V| \log |V|)$.

Algorithm 1: Dijkstra's Algorithm (Priority Queue)

input: data structure: `Graph`, vertex: `source`
output: array: `dist`, array: `prev`

```
1  create vertex priority queue Q
2  create distances array dist
3  create pointer array prev
4  dist[source] ← 0
5  Q.add_with_priority(source, 0)
6  for vertex v ∈ Graph.vertices() do
7     if v ≠ source
8        prev[v] ← UNDEFINED
9        dist[v] ← INFINITY
10       Q.add_with_priority(v, INFINITY)
11 while Q is not empty:
12    u ← Q.extract_min()
13    for v ∈ Graph.neighbors_of(u) do:
14       alt ← dist[u] + Graph.edge_weight(u, v)
15       if alt < dist[v]:
16          prev[v] ← u
17          dist[v] ← alt
18          Q.decrease_priority(v, alt)
19 return dist[], prev[]
```

C. A* Algorithm

The A* algorithm [5] can be considered a more general case of Dijkstra's algorithm that uses a heuristic function to more quickly find the shortest path to the goal. However, A* is a single-source, single-goal algorithm as opposed to a single-source all-goals algorithm as is Dijkstra's algorithm. This is a necessary consequence of using a goal-specific heuristic. This means that A* is more suitable for finding the shortest path to a specific goal, and only when there is an appropriate heuristic available. As long as the heuristic is *admissable*, that means that it never overestimates the true distance to the goal, A* is guaranteed to return a least-cost path from start to goal. If $h(x) = 0, \forall x \in V$, then A* is equivalent to Dijkstra's algorithm.

For both algorithms Listing 1 and Listing 2, Graph is assumed to be a data structure from which it is possible to access all vertices, neighbors of a certain vertex, and the edge weight of a certain ordered pair of connected vertices. `dist` is an array that contains the updated distances from `source` to each other vertex, where the array is indexed by a given vertex to obtain the corresponding distance. `prev` is, for Dijkstra's algorithm, an array that contains pointers to the previous vertices along the shortest paths from `source` to any given vertex where the array is indexed by that vertex to obtain the previous vertex in the path, and for A* it is a HashMap containing pointers to the previous vertex in the single shortest path from the `source` to the the `target`. For both algorithms, the actual path can be reconstructed via `prev`.

Algorithm 2: A* Algorithm (Priority Queue)

input: data structure: `Graph`, vertex: `source`, vertex: `target`, function: `h`
output: array: `dist`, array: `prev`

```
1  create vertex priority queue Q
2  create distances array dist
3  create pointer map prev
4  dist[source] ← 0
5  Q.add_with_priority(source, h(source))
6  for vertex v ∈ Graph.vertices() do
7     if v ≠ source
8        prev[v] ← UNDEFINED
9        dist[v] ← INFINITY
10       Q.add_with_priority(v, INFINITY)
11 while Q is not empty:
12    u ← Q.extract_min()
13    if u = target
14       break
15    for v ∈ Graph.neighbors_of(u) do:
16       alt ← dist[u] + Graph.edge_weight(u, v)
17       if alt < dist[v]:
18          prev[v] ← u
19          dist[v] ← alt
20          Q.decrease_priority(v, alt + h(v))
21 return dist[], prev[]
```

D. Bellman-Ford Algorithm

The Bellman-Ford algorithm [22] is identical to Dijkstra's algorithm apart from a few key differences. Dijkstra's algorithm will fail when there are negative cycles while Bellman-Ford can be used to detect them. Dijkstra is also more efficient with a time complexity of $O(|E| + |V| \log|V|)$ as opposed to Bellman-Fords $O(|E| * |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph. This is because Bellman-Ford performs a check (relaxation step) on each vertex in the graph, while Dijkstra only does this for the one with the best distance calculated so far. While this comparison step to find the best vertex makes it more complicated to implement in a distributed setting, Dijkstra has a superior time complexity compared to Bellman-Ford. If we are sure that negative cycles do not occur within our graph, as we usually are in our case because negative edge weights do not make sense when the the edge weights correspond to a distance metric based on the number of actions necessary to reach the vertex, then Dijkstra's algorithm will be the more efficient choice.

E. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm [23] is an all-pair shortest path algorithm, as opposed to Dijkstra's algorithm or the Bellman-Ford algorithm which are single source shortest path algorithms. The complexity of the Floyd-Warshall algorithm is $O(|V|^3)$, independent of the number of edges. This algorithm can therefore sometimes be favorable with extremely dense graphs but in most cases (provided there are no negative weights) it is still more efficient to simply run Dijkstra repeatedly for each vertex resulting in a time complexity of $O(|E| * |V| + |V|^2 \log|V|)$

F. Johnson's Algorithm

Johnson's algorithm [24] is an all-pair shortest path algorithm, and it works by first running Bellman-Ford on the graph to transform it into a graph with no negative edges, then repeatedly running Dijkstra's algorithm for each vertex. This is basically the same as repeatedly running Dijkstra's algorithm for each vertex but also works when there are negative edge weights and has a similar time complexity

of $O(|E| * |V| + |V|^2 \log|V|)$. For very dense graphs, Floyd-Warshall might have a better performance in practice.

IV. Prior Work

The problem statement of our method of interest, at the intersection of model-based and hierarchical reinforcement learning is quite specific and involves a number of moving parts.

It aims to combine the worlds of graph-based planning and deep reinforcement learning by building a graph-based representation of the environment upon which a high-level, non-parametric graph-search algorithm operates to provide short-horizon sub-goals for a low-level, goal-conditioned RL controller to reach. This approach naturally applies to (visual) navigation tasks, as these are particularly suitable for the graph representation and allow for intuitive visualization.

The main difficulties with this approach involve the question of exploration for how to obtain a collection of well-distributed datapoints from which to build a graph that sufficiently covers the state space, how to define the edge weights of the graph, and how to best sparsify the graph in terms of landmarks or state-space coverage to arrive at a suitable temporal abstraction.

Some works have side-stepped the exploration issue by assuming the ability to spawn uniformly over the state space [8], [9], [25], [26], assuming access to human demonstration data [27], or have mostly ignored the issue [28]. Others use some form of strategy to explore around the frontiers of thus-far explored areas [29]. It is also possible to act in a latent state-space to hallucinate samples for building a graph in a zero-shot manner [30].

To the question of how to connect the nodes of a graph, or how to arrive at a suitable notion of distance between two states or nodes, some works have side-stepped that question by assuming access to domain knowledge [31], some build a graph by connecting trajectories based on transitions and matching (or similar) states [27], [32], while others learn a distance function that can provide these edge weights [8], [9], [25]. If a latent state-space is learned, then some cluster in this space in terms of reachability [26] or use a cross-entropy maximization method to propose subgoals [28].

Graph sparsification is important not just for the exponentially increasing computational complexity of operating on a dense graph, but also for minimizing the number of faulty edges (i.e. wormholes) that a graph-search algorithm will certainly exploit and, as a consequence, return an infeasible plan. Some works have nevertheless ignored this problem [8], while others have devised a Q-irrelevant sparsification algorithm [9] or, similarly, used a learned distance metric together with a cutoff threshold [29]. Some have utilized sampling based approaches [25], [33] to obtain a subset of nodes that still provide a good state-space coverage. While still others have used some form of latent space projection for states to arrive at a suitable set of subgoals [26], [28]. Most methods use a number of additional tricks to reduce the number of faulty edges such as k-nearest-neighbor filtering [9], ensembles of networks [8], distributional RL [8], etc.

The localization of the agent within the internal graph representation can be done in many ways such as by simple pattern-matching with each node in memory [8], by some similarity or distance metric [9], [29], [25] or by training a retrieval network [27].

The locomotion controller is naturally formulated as a goal-conditioned short-horizon reinforcement problem and it has the benefit of being able to be trained in isolation and on nearby goals to speed-up the learning process, because it only needs to learn enough of the dynamics of the environment to be able to select actions to traverse between subgoals.

With the graph representation at a suitable level of abstraction in place, the high-level controller is then trivial, with most of the approaches choosing some variant Dijkstra's algorithm [27], [31], [8], [9], [25] although some also learn a high-level controller [32], [28], [26].

We place a particular focus on [8], [9] in this thesis, so we will provide an in-depth summary of those two papers as well as of [27] and [25] which paved their way.

A. *Semi-Parametric Topological Memory (2018) [27]*

One of the first notable applications of these ideas within the context of deep learning was done in [27] as the Semi-Parametric Topological Memory. Here the authors are inspired by vision-based SLAM (Simultaneous Localization And Mapping) methods. These are methods of building a metric map of the environment from visual sensory data such as from cameras as well as movement data such as from motors that date back to the 80s [34]. However, differently to SLAM methods, they aim to build a topological map instead of a metric one and they do so utilizing deep learning methods.

Incoming observations are used directly as nodes in the internal graph representation, and these nodes are con-

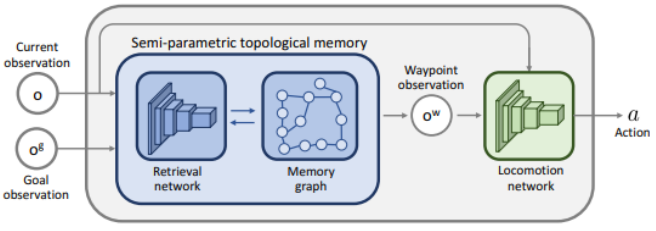

Figure 9: SPTM Algorithm, figure taken from N. Savinov, A. Dosovitskiy, and V. Koltun [27]

nected by edges if they directly follow each other chronologically or if they are sufficiently similar as determined by the retrieval network. Important to note here is that the authors use human demonstration data to bootstrap a qualitative initial dataset.

The retrieval network as well as the locomotion network are trained in a self-supervised fashion on the collected transitions in the buffer, where the buffer is treated simply as a labeled dataset. The retrieval network is trained to estimate the similarity between two observations, where two observations that are temporally close are considered similar. The locomotion network is trained to produce action probabilities, given a current observation and a goal observation (both sampled from the dataset, where the goal observation is separated from the current observation by no more than 20 timesteps). The high-level, non-parametric planner is chosen to be Dijkstra's algorithm.

B. Mapping State Space using Landmarks for Universal Goal Reaching (2019) [25]

In [25], a locomotion controller is pre-trained using a goal-conditioned reinforcement learning agent, or universal value function approximator (UVFA), and training is improved by using hindsight experience replay [13] which increases the sample efficiency by relabeling past transitions during training as if the resulting states had in fact been the correct goal. Because this agent is trained using the simple indicator reward function

$$r_t = R(s_t, a_t, g) = \begin{cases} 0 & \|s'_t - g\| < \delta \\ -1 & \text{otherwise} \end{cases} \tag{8}$$

(where δ is chosen to be some small value) the value-function learned by the resulting local agent will correspond to a (negated) distance function between any two states $d(s, g)$,

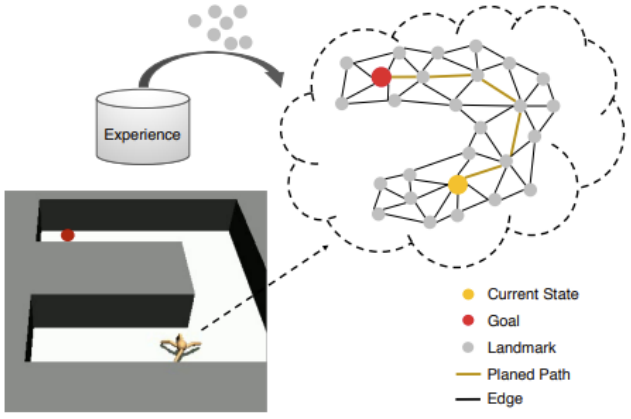

Figure 10: Illustration of the graph-based approach, figure taken from Z. Huang, F. Liu, and H. Su [25]

because the expected return from any state to any goal state is simply the negative of the number of steps required to reach it. The authors also note that this goal-conditioned agent fails at reaching far-away goals but successfully and consistently learns to reach short-horizon goals.

Due to the computational cost of considering every state in the replay buffer, the authors propose a landmark sampling strategy to build a sparse graph representation. They start by uniformly sampling a large number of transitions from the buffer and then perform farthest point sampling [35] on that set with the goal of sufficiently covering the so-far discovered state space. The metric for farthest point sampling can either be the true euclidean distance between two states if known, or the learned distance metric from the pre-trained locomotion controller. They then build a graph from the resulting states using edge weights determined by the learned distance estimator, only connecting states if they are sufficiently close as determined by a hyperparameter: $d(s, g) < \tau$.

The authors re-construct the graph at every timestep t which means they can make sure that the current state is one of the nodes in the graph making retrieval trivial. With the resulting directed, weighted graph based on the estimated distance function they make use of the Bellman Ford algorithm for the high-level planner to find the shortest paths.

C. Search on the Replay Buffer: Bridging Planning and Reinforcement Learning (2019) [8]

As in [25], in [8] a UVFA is pre-trained locally on close-together states and goals using the indicator reward function (8) resulting in a locomotion controller and a learned distance function approximator. Differently to [25], the authors use distributional reinforcement learning [36] as a trick to improve predicted distances where the agent is forced to learn not just the single-valued expected average return but more specifically the discretized distribution of returns where each of N bins corresponds to the number of steps away from the goal (the last bin corresponds to "at least N steps away"). Another trick the authors use to learn better distance estimates is to train an ensemble of estimators and pessimistically aggregate predictions.

Previously seen observations from the replay buffer are used directly as nodes in the internal graph representation, but here the authors construct a dense, almost fully con-

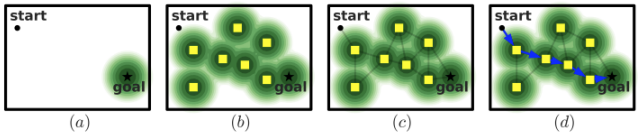

Figure 11: Illustration of the dense graph, figure taken from B. Eysenbach, R. Salakhutdinov, and S. Levine [8]

nected graph where every state in the buffer is connected to every other state by an edge with weight equal to the predicted distance from the pre-trained UVFA unless the distance is larger than some hyperparameter MAXDIST. The graph grows with new states over time and is otherwise fixed, but the shortest paths still need to be calculated anew after every timestep. The authors choose Dijkstra's algorithm to do this, but to amortize this cost they use the Floyd Warshall algorithm to efficiently compute the shortest paths between all points of the graph also reducing the number of necessary calls to the expensive value function.

This approach of building a dense graph using every node in the replay buffer suffers from the quadratic growth of edges with each node and quickly becomes unscalable. A large number of edges also increases the chance of faulty edges representing infeasible transitions which will be exploited by the high-level graph search algorithm.

D. Sparse Graphical Memory for Robust Planning (2020) [9]

The authors in [9] build directly upon [8] and focus on sparsification to reduce faulty edges and computational complexity. They devise an algorithm to dynamically build a sparsified the graph based on a similarity metric they call two-way-consistency (TWC) to merge, or rather ignore, redundant nodes in a τ-approximate, Q-irrelevant manner for the high-level planner such that the quality of the original graph is preserved with respect to high-level plans up to a linear factor. The intuition here is that two states are redundant if they are both interchangeable as goals and interchangeable as starting states according to the goal-conditioned value function.

This sparse graph is built via a single pass through the replay buffer by an online, greedy algorithm which iteratively adds nodes from the buffer to the set of nodes to keep if these nodes are two-way-consistent with respect to the other nodes in the set to keep. If a node is added, edges are created and set to predicted distances unless the distance is larger than some hyperparameter MAXDIST. Further, nodes are limited to be connected to their k-nearest neighbors and during test-time edges are removed if the low-level agent is unable to traverse them. All other aspects are held equal to [8]. More details on the TWC criteria is provided later on in Section VI.D.

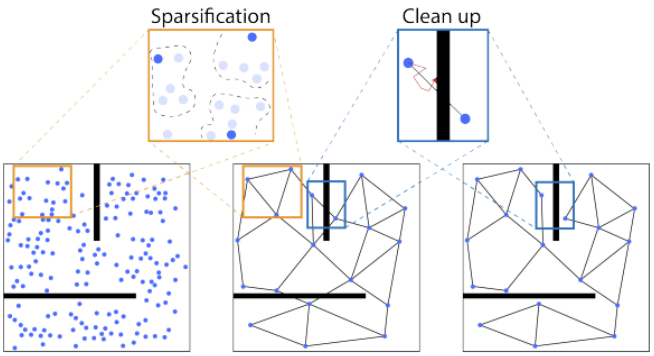

Figure 12: Illustration of the SGM sparsification, figure taken from M. Laskin, S. Emmons, A. Jain, T. Kurutach, P. Abbeel, and D. Pathak [9]

The authors address the shortcomings of the previous approach [8] and show significant improvements in terms of performance and success rates on different environments as well as execution speed due to the induced graph sparsity that is provably error bound.

V. Rust

One of the goals of this thesis was to evaluate and push forward the maturity of the Rust [37] programming language in the field of machine learning and artificial intelligence research. To this end, instead of directly making use of the available code of [8] and [9] it was entirely re-implemented in Rust and an open-source contribution to one of the few major machine-learning frameworks was made where certain tools were missing from the ecosystem.

Rust was created at Mozilla in 2006 by Graydon Hoare, was sponsored by Mozilla in 2010, released it's first stable version `1.0` in 2015, and became independent of Mozilla via the Rust Foundation in 2020. It was the first industry-supported programming language to overcome the longstanding trade-off between the safety guarantees of high-level languages and the performance via memory control of low-level languages [38]. It does this by means of a strong type system, based on the ideas of *ownership* and *borrowing*, by which the compiler, infamously named the "borrow checker", statically manages memory at compile time without the need for a garbage collector and at the same time provably makes entire classes of bugs impossible [37], [38].

A recent analysis of bugs at Microsoft revealed that 70% of vulnerabilities that are assigned a common vulnerabilities and exposure (CVE) tag are due to memory safety bugs [39]. Fortunately, a Rust program that compiles is guaranteed, due to the compiler, not to have any memory safety bugs: use after free, dereferencing a null pointer, double free, buffer overflow and buffer overread. Additionally, it is guaranteed not to have any data races, allowing for lovingly called "fearless concurrency" in Rust.

The language focuses on performance by relying heavily on *zero-cost abstractions* which are high-level abstractions such as data structures and methods that do not come with a runtime cost, only a compile time cost. It is also a compiled and therefore optimized language, and without a garbage collector it matches the performance of C/C++. The lack of a garbage collector due to the borrow-checking mechanism and lack of a required runtime, also comes with a drastically

reduced memory footprint compared to other high-level languages such as Python and Java.

The adoption of Rust has seen a steady increase in the years since 2015 and a recent milestone is the adoption of Rust as the only other language apart from C so far into the Linux Kernel. In the yearly survey by StackOverflow, the latest one to-date being 2023, Rust has been voted the "Most Loved" programming language for 8 years in a row now [40] and the reasons for this, apart from the confidence in program correctness, is also due to the great tooling that comes with the language. The problems of dependency management (e.g. virtual environments in Python), build management (including cross compilation to different operating systems and into statically linked, standalone binaries), testing and benchmarking are solved with Rust's Cargo. Testable documentation is built directly into the code and when publishing a package (crate) it is directly published online on `docs.rs`. Clippy is a code analysis tool that provides helpful tips on how to improve your code. Rustfmt formats your code into a standardized specification.

One of the main, practical benefits to writing a pure-Rust application, is that it compiles down to a single standalone binary with no dependencies which immensely simplifies deployment, an often forgotten but incredibly complex step beyond the research phase.

A. State of Machine Learning in Rust

To the start of this thesis, the reinforcement learning ecosystem in Rust was almost non-existent. There were some examples of deep reinforcement learning using `tch-rs` [41], a library providing Rust bindings to `libtorch` similarly to `PyTorch`, but no examples using one of the three available pure-Rust machine-learning frameworks: `dfdx` [42], `Burn` [43], or `candle` [44]. To the best of our knowledge, deep reinforcement learning, if at all, has only ever been done before in pure Rust as a proof-of-concept curiosity a handful of times.

The machine-learning libraries available were also at a very young stage of maturity. At the time this thesis was started (January 2023), `candle` had not even yet existed and `Burn` had just been announced:

- `dfdx` (first commit: August 2021) is the first pure-Rust machine learning framework. It is maintained by a single author and as a consequence, has only a simple feature set and a slow development speed.
- `burn` (first commit: July 2022) is a promising framework with some development momentum, but after many attempts turned out to be too complicated (due to the heavy use of traits) and opinionated for reinforcement learning research.
- `candle` (first commit: June 2023) is a well designed framework that despite being the youngest, is the most battle-tested due to an incredibly extensive set of state of the art models implemented in the examples. It also enjoys a fast development speed due to being backed by HuggingFace and gaining a large community.

Of the three frameworks, `candle` was considered the most promising because it used a more flexible and usable design in which the library user is not required to handle lifetimes (a quite complicated concept unique to Rust) and because it was backed by HuggingFace, a major open-source AI company, instead of being maintained by only a single developer. While `candle` has an impressive list of state-of-the-art models as examples in the repository, most of them related to large language models and no reinforcement learning examples were available. Our open-source contribution to the `candle` framework was to add the `DDPG` algorithm to the list of examples, providing not just the first pure-Rust deep reinforcement learning example to the ecosystem but also providing a number of helpful reference implementations such as for a replay buffer, the logic for soft-updates between a network and a target network, and a principled way to create and interact with the Python implementation of the popular reinforcement learning benchmarking library OpenAI Gym [19], which is now hosted by the Farama Foundation aiming to standardize and maintain reinforcement learning tools.

VI. Methods

The setting we are interested in specifically is an (impure) off-policy, model-based, hierarchical reinforcement learning agent where the low-level controller is a goal-conditioned reinforcement learning agent and the high-level controller is a non-parametric planning algorithm operating on- and with reversible access to- a learned, graph-based world model. These algorithms have a number of inherent advantages due to the graph-based world model such as being semi-parametric where the high-level controller does not need to be learned nor does it have any hyperparameters that need to be tuned, and they can easily be inspected or even edited providing a huge benefit in terms of interpretability. The graph along with the goal-conditioned nature of the algorithm also provides us with the flexibility of not having to retrain the agent for small changes in the environment or to reach different goals.

We design one such algorithm, and base our implementation on the works on [9] and, as a consequence, [8], where a graph is built directly from previously encountered states. Like [8], [9], we also use an actor-critic architecture, in particular the DDPG algorithm, to train the low-level controller and the distance function simultaneously via the actor and the critic, respectively. Due to time constraints and a lack of suitable hardware to test on, we replaced the learned dis-

tances with the true euclidean distance function as these were significantly faster to compute, reducing the time taken per timestep (adding one node to a decently sized graph) from several minutes to a fraction of a second. We did not have the time to evaluate graph construction from distances learned by the critic.

We implemented the two-way-consistency criterion proposed in [9] to sparsify the graph, and similarly to [9] we introduced a clean-up procedure where we remove faulty edges that take more than SGM $\max$ tries timesteps to reach. Unlike [9], we do this during training as well as during evaluation.

We benchmark a Hierarchical Graph-Based DDPG (HGB-DDPG) algorithm against a Hierarchical DDPG (H-DDPG) without the graph-based model on the same PointEnv environment as in [8] and [9].

A. PointEnv Environment

We focus our experiments on the simple PointEnv environment as in [8] and [9], which is a 2-dimensional navigational environment with a continuous action space and a proprioceptive observation space. The observations are simply the locations of the agent, $s = (x, y) \in \mathbb{R}^2$, and the actions, $a = (dx, dy) \in \mathbb{R}^2$, are vectors within the unit circle around the agent that are added to the agents current location at every timestep unless there is wall blocking the path, in which case the agents location ends up being a small distance away from the collision point.

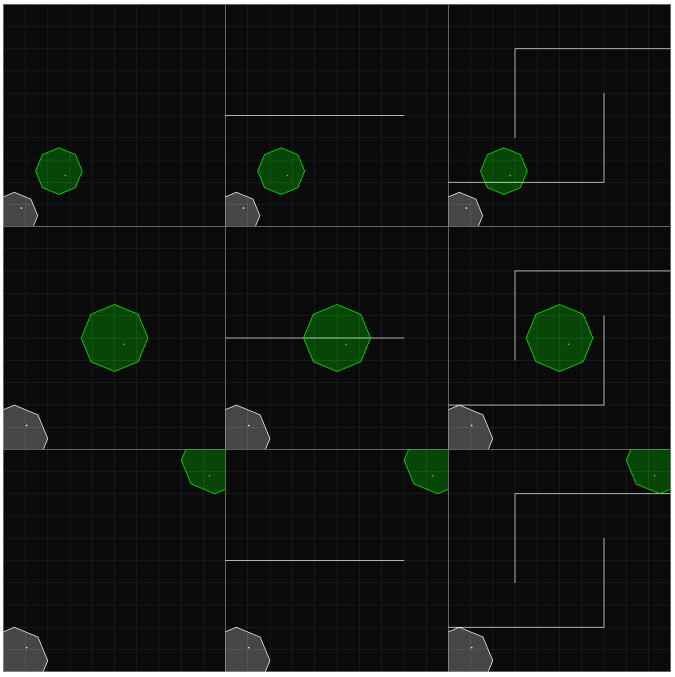

Figure 13: The three environment wall-configurations (PointEnv-Empty, PointEnv-OneLine, and PointEnv-Hooks) across columns combined with the three goal state locations and radii (close, mid, and far) across rows. Spawn states are shown in gray while goal states are shown in green.

All environments are of the same size (10x10), and the differences between the environments are in wall configuration (Empty, OneLine, Hooks) and in the areas in which the spawing states and the goal states are sampled (close, mid, far). The wall configurations, in increasing difficulty, are either the empty environment with no walls, a single horizontal line through the center from the left edge extending towards the right, and two hooks extending from either edge towards each other almost locking arms. The spawing state is fixed to be centered around the point `(0.5, 0.5)` while the goal states and radii vary for different configurations, in increasing difficulty, around the point `(2.5, 2.5)` with radius `1.05` for close, around `(5.0, 5.0)` with radius `1.5` for mid, and around `(9.5, 9.5)` with radius `1.5` for far. These configurations make for a total of 9 different environments visualized in Figure 13. To end the episode successfully, the agent must change its own location to be within a radius of `0.5` of the goal state.

B. Deep Deterministic Policy Gradient (DDPG)

To handle continuous action spaces, we choose the DDPG algorithm [45] which is an actor-critic reinforcement learning method. It uses some of the same "tricks" used by the highly successful DQN [2] algorithm to handle the violations of many of the assumptions that the convergence proofs are based on which make training in practice highly unstable. In particular, a replay buffer is used to retrieve training samples from to somewhat restore the violated i.i.d assumption on the (otherwise highly correlated) samples used to train a neural network, and a target network is used to somewhat restore the independence assumption between the target and the prediction in the loss function.

The actor-critic architecture dates back to [46]. These algorithms make use of an additional network to represent the policy independent of the value function [10]. This policy network is referred to as the actor, because it produces actions based on observations, while the value function network is referred to as the critic, because it critiques the actions selected by the actor by producing a scalar, reward-like output for the current and the following observations.

For each transition, the critic evaluates the quality of the action taken by computing the temporal difference error as:

$$\text{TD-ERROR} = R_{t+1} + \gamma V_\theta(O_{t+1}) - V_\theta(O_t) \qquad (9)$$

Where V_θ is the value function as currently represented by the critic network. If the temporal difference error is positive, the action improved the value of the current state and the actor should be rewarded, otherwise it should not. This

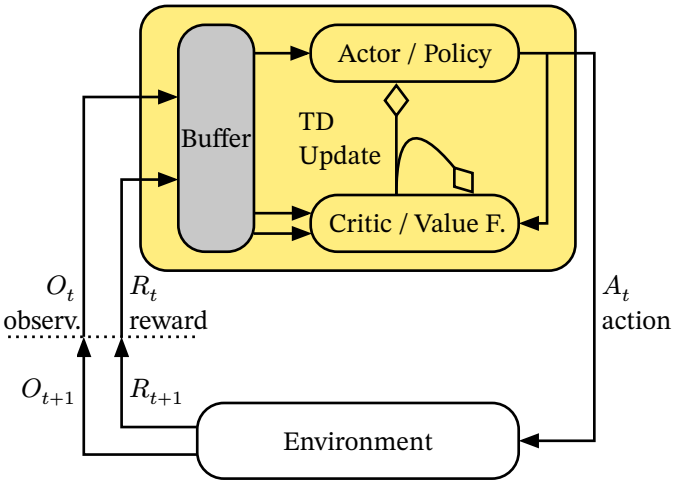

Figure 14: The actor-critic architecture, shown here as an (impure) off-policy implementation with a replay buffer. The critic helps determine the temporal difference (TD) error based on some transition, and this error is used to update both the actor and the critic networks.

error term computed by the critic drives all learning in both the actor and the critic [10]. This general architecture is represented in Figure 14.

In the DDPG algorithm, both the actor network as well as the critic network each have a more slowly updating copy of themselves referred to as the target network. Each target network (represented by the weights θ'), which starts out as an exact copy of the faster-changing learned network (represented by the weights θ), "follows" the learned network as it is updated, but changes more slowly so as to provide more stability during training. As the learned network is updated, at each timestep, the weights of the target network are updated as follows:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{10}$$

where $\tau \ll 1$.

To deal with the problem of exploration, especially at the start of the training process, a number of initial random actions are taken and, as in [45], Ornstein-Uhlenbeck noise [47] is added to the output of the actor network to create the exploration policy π':

$$\pi'_\theta = \pi_\theta(S_t) + \mathcal{N} \tag{11}$$

where $\mathcal{N}$ is random noise generated by the Ornstein-Uhlenbeck process.

C. Hierarchical DDPG (H-DDPG)

We can extend the DDPG algorithm to the goal-conditioned case by simply augmenting the observation space by the current goal as described in Section II.I, and varying that goal over different episodes forcing the agent to learn a global policy that generalizes across any given goal. This simple change to the standard DDPG algorithm will be our baseline algorithm.

D. Sparse Graphical Memory (SGM)

To build up to our HGB-DDPG algorithm, we first describe the SGM algorithm which is used by HGB-DDPG. It makes use of the two-way-consistency (TWC) criteria proposed by [9] to construct a sparse graph. This results in a graph that is far more computationally feasible to perform graph-search on and has the added benefit of reducing the number of invalid edges which pose a problem for graph-search algorithms.

Consider a replay buffer $\mathcal{B}$, containing a number of previously encountered states $\mathcal{S}_\mathcal{B}$, and any asymmetric distance function $d(\cdot,\cdot)$ between two states. Given $s_1 \in \mathcal{S}_\mathcal{B}$ and $s_2 \in \mathcal{S}_\mathcal{B}$, and iterating over all other states $\omega \in \mathcal{S}_\mathcal{B} - \{s_1, s_2\}$, if the maximum absolute difference in distance between starting in either s_1 or in s_2 and trying to reach ω is below some threshold τ

$$C_{\text{out}}(s_1, s_2) = \max_\omega |d(s_1,\omega) - d(s_2,\omega)| \leq \tau \tag{12}$$

then the states s_1 and s_2 are considered interchangeable as starting states. Similarly, if the maximal difference in distance between attempting to reach either s_1 or s_2 while starting in ω is below some threshold τ

$$C_{\text{in}}(s_1, s_2) = \max_\omega |d(\omega,s_1) - d(\omega,s_2)| \leq \tau \tag{13}$$

then the states s_1 and s_2 are considered interchangeable as goal states. If two states are interchangeable both as starting and as goal states, then they are considered redundant and one of these states can be dropped from memory.

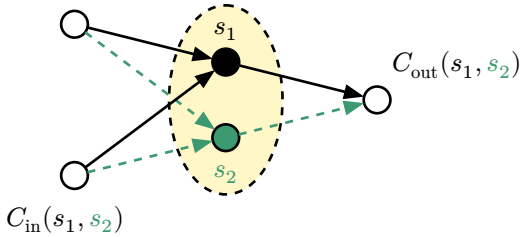

Figure 15: Node merging strategy in the SGM algorithm. This diagram is reconstructed from [9]. Consider s_1 as already contained in the graph, and s_2 as a candidate to be added. For C_{in}, we consider s_2 a potential goal state and compare differences to s_1, while for C_{out} we consider s_2 a potential starting state.

This method of sparsification prioritizes the coverage of the state space in the resulting graph. The authors also provide a formal proof for the following theorem: Given a graph $G_{\mathcal{B}}$ with all the states from a replay buffer $\mathcal{B}$ as nodes and weights from any asymmetric distance function $d(\cdot, \cdot)$, and a graph G_{TWC} that is constructed by aggregating states in $\mathcal{B}$ according to TWC (where a state is not added to the graph if $C_{\text{out}} \leq \tau$ and $C_{\text{in}} \leq \tau$), then considering a shortest path $P_{\mathcal{B}}$ with k edges between any two states in $G_{\mathcal{B}}$ and the corresponding path P_{TWC} connecting the same two states in G_{TWC}:

- The weighted path length of P_{TWC} minus the weighted path length of $P_{\mathcal{B}}$ is no more than $2k\tau$
- If $d(\cdot, \cdot)$ has error at most ε, the weighted path length of P_{TWC} is within $k\varepsilon + 2k\tau$ of the true weighted distance along $P_{\mathcal{B}}$

In the case where a non-parametric high-level controller, with an action space defined by the states $\omega \in \mathcal{B}$ (i.e. it chooses waypoints ω for the low-level controller to reach based on the current goal), we can apply the TWC criteria as follows, for a τ-approximate, Q-irrelevent abstraction over ω for the high-level controller:

$$\max_{\omega} \left| Q^{\text{hl}}(s_1, a = \omega | g) - Q^{\text{hl}}(s_2, a = \omega | g) \right| \leq \tau \quad (14)$$

$$\max_{\omega} \left| Q^{\text{hl}}(\omega, a = s_1 | g) - Q^{\text{hl}}(\omega, a = s_2 | g) \right| \leq \tau \quad (15)$$

The graph is constructed in an online fashion, where a newly observed state is either added to the graph if it satisfies TWC with respect to the nodes already in the graph or it is not added to the graph. This way, the algorithm has a time complexity of $O(|V|^2)$ for each node to be added, where $|V|$ is the number of nodes in the sparse graph at that moment.

E. Hierarchical Graph-Based DDPG (HGB-DDPG)

When combining the ideas of goal-conditioned RL, off-policy RL via a replay buffer, and the actor-critic architecture, we have the opportunity to build an explicit graph-based model using previous observations collected from the transitions in the buffer as nodes and the learned value function of the goal-conditioned agent to determine the edge weights. While learning a goal-conditioned agent is much harder than a simple fixed-goal agent, especially for longer-horizon tasks, we only require it to reach short-horizon subgoals while the graph-based model allows us to use a non-parametric graph-search algorithm as a high-level planner.

The HGB-DDPG algorithm is in essence a high-level agent which is a wrapper around a low-level H-DDPG agent, it uses the standard H-DDPG algorithm to reach short-horizon goals and graph-based methods to make long-horizon

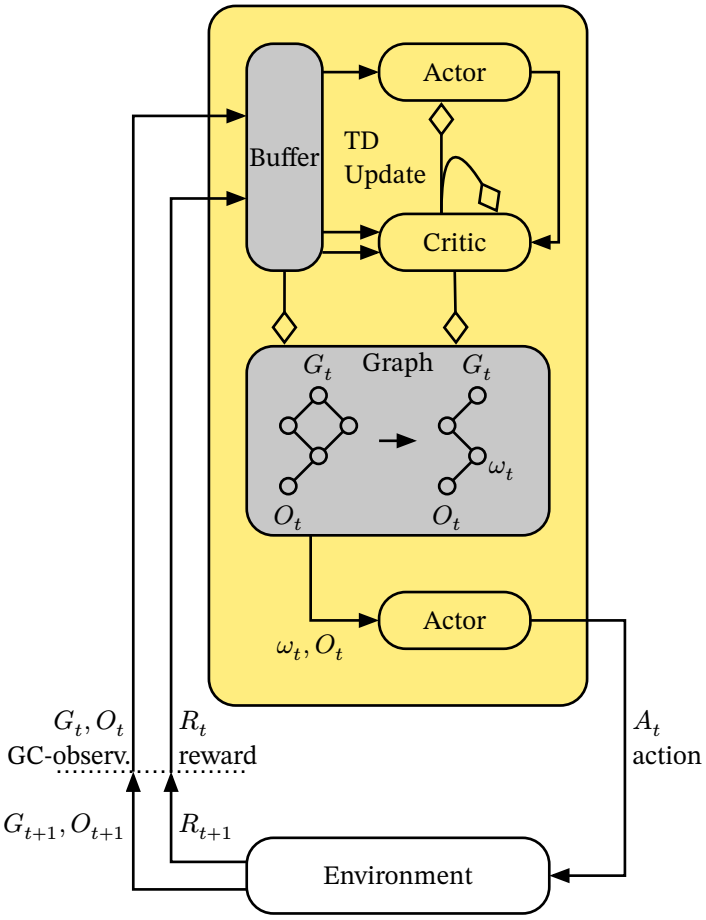

Figure 16: The Hierarchical Graph-Based DDPG (HGB-DDPG) algorithm. There top of the diagram shows the same actor-critic diagram as in Figure 14, and the goal-conditioned interaction with the environment is also identical to the diagrams we have discussed before, e.g. in Figure 7. The difference here is the addition of a graph component that is capable of building a graph from the replay buffer and the critic network and generating plans from that graph.

plans. At the start of training, it initializes a H-DDPG algorithm as well as a directed graph data structure $G = (V, E, w)$. It also makes a choice of whether to use a static distance function $d = d(s_{\text{from}}, s_{\text{to}})$ if one is provided, or to use the goal-conditioned critic-network of the H-DDPG as a distance function by setting the goal to be equal to one of the states, e.g. $d = \text{Critic}(s_{\text{from}}, s_{\text{to}}, s_{\text{to}})$ where $s_{\text{from}}, s_{\text{to}} \in \mathcal{S}$.

Note that the critic is an asymmetric distance function (i.e. $\text{Critic}(s_{\text{from}}, s_{\text{to}}, g_{\text{current}}) \neq \text{Critic}(s_{\text{to}}, s_{\text{from}}, g_{\text{current}})$), while the provided distance function does not have to be (i.e. the euclidean distance). The TWC criterion does not make an assumption here and works for both cases. The combination of goal-conditioned and asymmetric does however require some care when implementing, for example to remember setting the goal accordingly $g_{\text{current}} = s_{\text{to}}$ or $g_{\text{current}} = s_{\text{from}}$.

At each timestep t the incoming state $s_t \in \mathcal{S}$ is added to the replay buffer of the H-DDPG agent and evaluated via the TWC sparsification criterion with respect to the distance function d and states $s_n \in V \subset \mathcal{S}$ currently in the graph, which decides whether the state will be added to the graph.

If the state satisfies the criterion, then it is added to the graph and edges are added in both directions between the new state s_t and each other state s_n with an edge weight equal to the output of the distance function of the corresponding direction (i.e. $w(s_t s_n) = d(s_t, s_n)$ and $w(s_n s_t) = d(s_n, s_t)$). If the edge weight is smaller than MAXDIST, then no edge is added:

$$
\begin{aligned}
w(xy) &= d(x, y) \\
V &= V \cup \{s_t\} \\
E &= E \cup \{s_t s_n \mid s_n \in V \wedge w(s_t, s_n) < \text{MAXDIST}\} \\
E &= E \cup \{s_n s_t \mid s_n \in V \wedge w(s_n, s_t) < \text{MAXDIST}\}
\end{aligned} \tag{16}
$$

This graph grows over time while remaining sparse in such a way as to preserve the state coverage of the environment. To choose an action, the high-level agent first checks if the current goal is reachable via some waypoint in the graph by comparing the distance from each waypoint to the goal $d(w_{\text{from}}, g_{\text{current}}, g_{\text{current}})$ where $w_{\text{from}} \in V$ and checking whether it is below some threshold.

If one or more candidates are found, a plan is generated by using Dijkstra's algorithm to find the shortest path between the the closest waypoint to the current state and the closest waypoint to the goal state. The high-level agent then provides the low-level H-DDPG agent with the current state and the next waypoint in the plan as the goal to get the next action. If no candidates were found and thus no plan could be generated, the HGB-DDPG policy defaults to standard H-DDPG policy by passing the current state and the current goal to get the next action.

The high-level agent generates a plan either at the beginning of each episode or if no plan was able to be generated it attempts to do so again at each timestep. It allows the low-level agent to attempt to reach the waypoint for a fixed number of timesteps after which it considers the transition infeasible and removes the edge from the graph and generates a new plan. This cleanup strategy is paramount to the success of the algorithm as otherwise any infeasible transition in a plan would result in a guaranteed failure to reach the goal, however it should be noted that the initial poor performance of the low-level agent while it is still learning will result in many feasible edges being incorrectly removed. This can be partially mitigated by reconstructing the graph anew from the states in the replay buffer every few episodes and letting the cleanup procedure continue, however the states in the buffer will at some points no longer be normally distributed over the state space as old states are dropped from memory resulting in the new graph no longer sufficiently covering the state space. This is potentially the most crucial problem to be solved in future work.

At the end of each episode, the H-DDPG agent is trained for a number of iterations on the transitions collected in the replay buffer. Transitions in which the low-level agent was reaching a waypoint are saved in the replay buffer as such, with the current goal replaced by the waypoint and an internal reward is provided when a waypoint was reached. This self-supervised mechanism speeds-up the learning process and is similar to goal relabeling strategies such as [13].

F. Experiment Setup

There are two main challenges with our approach. For one, while we are still learning the value function the distances will be very inaccurate and any invalid edge in which the distance between two nodes is underestimated will be exploited by the high-level planner resulting in infeasible plans.

Another challenge is how to deal with the exploration issue because the high-level planner can only find paths to any goal if the graph sufficiently covers the state-space and while the Orenstein-Uhlenbeck process helps in exploration, it by far does not result in a uniform exploration of the state space when the spawing states are (relatively) fixed.

To deal with the first of these two problems, [8], [9], [25] have proposed a number of tricks to improve distances learned by the critic and to reduce the number of invalid edges, although all of these works have sidestepped the second problem by assuming the ability to spawn uniformly over the state space. We will sidestep both of these issues by assuming 1) access to the true euclidean distance function between states and 2) the ability to spawn uniformly over the state space (during an initial data collection phase).

The approach we will take is to first generate an initial set of uniformly distributed and hopefully valuable transitions during a data collection phase. During this phase, for a total of 3000 episodes, the agent is spawned uniformly over the state space and a goal is sampled somewhere within a radius of larger than the step-size of the agent but smaller than 1.5 times the step-size. The agent then transitions for a maximum of 5 timesteps (or until the goal is reached) with the random policy and these transitions are saved in the replay buffer. The agent is not trained during this phase. This starts the agent off with some uniformly distributed data and gives it, with a high probability, a few successful transitions as demonstration data at the same time to kickstart learning.

After this initial data collection period, the agent is trained normally on the environment with the spawing states and goal states restricted to certain areas. One training run con-

sists of 300 episodes. At the end of each episode the agent is trained for 200 training iterations where at each iteration the agent is trained on a batch of 64 randomly sampled transitions from the replay buffer.

The hyperparameters of the agent are held fixed over the different environments, and the hyperparameters of the H-DDPG are held equal to the hyperparameters of the DDPG component of the HGB-DDPG (See Table 3, Table 5, and Table 4). Similarly, the parameters of the environments are held equal with the exception of their explicit differences described in Section VI.A.

We will run both H-DDPG and HGB-DDPG on the 9 different variants of the PointEnv environment with the same hyperparameters and training configuration and setup, which we consider as in-distribution challenges, and observe the differences in performance. We will then investigate the performance of HGB-DDPG over H-DDPG on a select number of out-of-distribution challenges where an agent pretrained on an easier environment is evaluated on a harder environment. Finally, we will take a closer look at the internal graph model in different situations to make some conclusions about the behaviour of the agent and take the opportunity to showcase the qualitative benefits of the graph model.

Parameter	Value	Comments
learning rate	0.0003	Same for actor and critic
gamma / discount	0.99	
tau / soft-update rate	0.005	Soft update is done at every timestep
hidden layer size #1	256	Same network params for actor and critic
hidden layer size #2	256	Same network params for actor and critic
replay buffer size	10,000	
batch size	64	

Table 3: Both the H-DDPG and HGB-DDPG algorithms share these same DDPG-related hyperparameters.

Parameter	Value	Comments
theta	0.0	The mean to which the process returns
kappa	0.15	The speed with which the process returns to the mean
sigma	0.2	The volatility of the process

Table 4: Hyperparameters for the Ornstein-Uhlenbeck noise process. Both the H-DDPG and HGB-DDPG algorithms use these same parameters.

Parameter	Value	Comments
distance mode	True	Use the True distance function or the learned distances from the critic
graph reconstruct freq	50	Number of timesteps after which to reconstruct the graph from the buffer
max tries	5	Number of timesteps to try reaching the waypoint before removing the edge
close enough	0.5	The threshold distance for considering the waypoint to be reached (chosen to be equal to the environment)
waypoint reward	0.0	The reward given for reaching a waypoint (chosen to be equal to the environment)
max distance	1.0	The maximum distance allowed for two states to be connected by an edge
τ	0.4	Parameter τ for graph sparsification (see Section IV & Section VI.D)

Table 5: Hyperparameters exclusive to HGB-DDPG.

VII. Results

We can see that apart from on environments in which the goal state is very close to the spawing state, the HGB-DDPG algorithm consistently scores higher on in-distribution challenges and has a very clear advantage on out-of-distrbution challenges. The interpretability advantages of the graph module is showcased on a model which was early-stopped while training on the hardest environment, a typical case of cherry picking the model with the best performance only here we can be more certain of the true quality of the model, and on an out-of-distribution challenge where the agent was pretrained on the easiest- and then evaluated on the hardest environment. All plots are shown with the mean performance in terms of success-rate over 300 episodes, aggregated over 50 identical runs, visualized by a bold line and a shaded region is highlighted above and below that line visualizing the standard error of the mean. The color orange

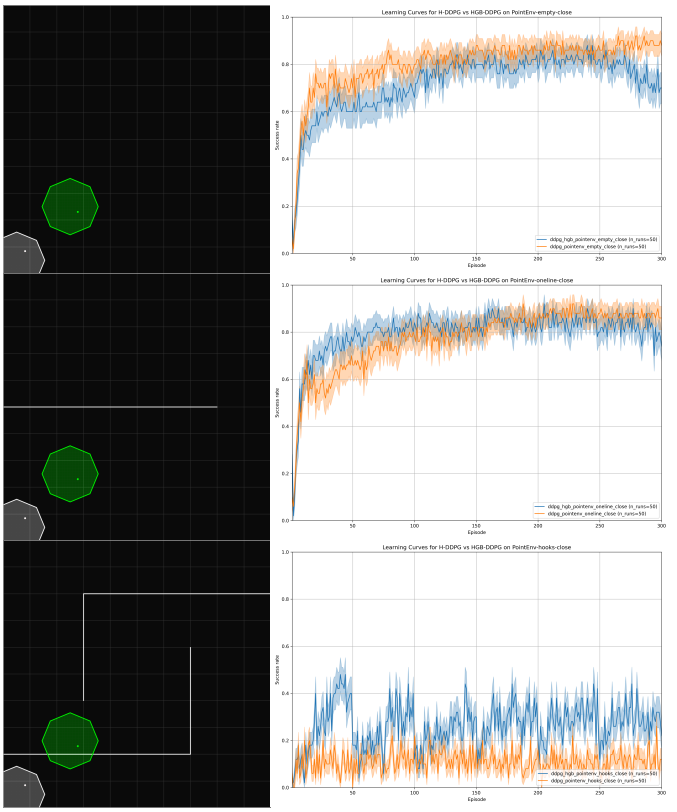

Figure 17: The performance of the H-DDPG baseline (orange) vs our HGB-DDPG algorithm (blue) on the PointEnv environments with distance "close".

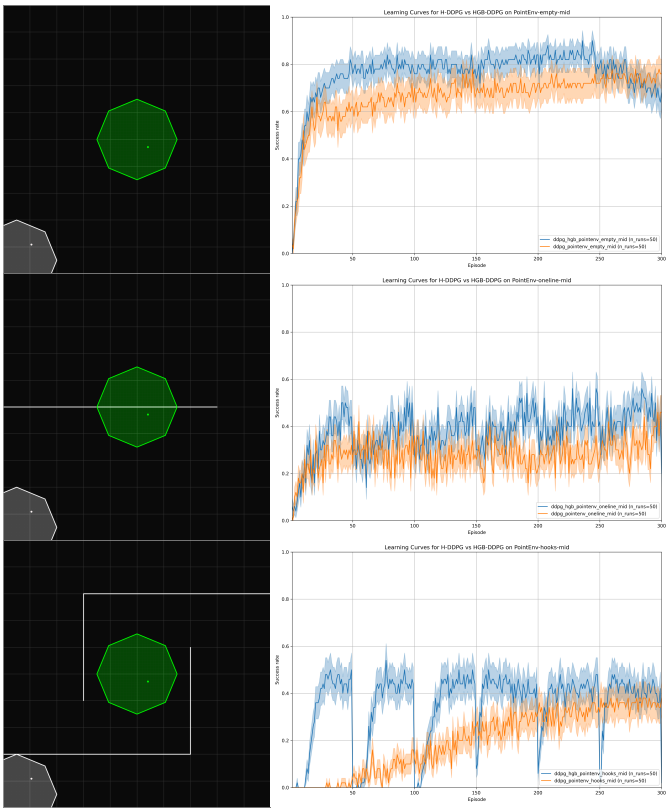

Figure 18: The performance of the H-DDPG baseline (orange) vs our HGB-DDPG algorithm (blue) on the PointEnv environments with distance "mid".

is consistently chosen to represent the baseline H-DDPG algorithm while blue shows our HGB-DDPG algorithm.

A. *In-Distribution Challenges*

We can see in Figure 17 how the asymptotic performance of our algorithm for close distances (top and middle plots) is worse than the standard H-DDPG algorithm without the graph module, this is because the overhead of learning to reach waypoints and managing the internal graph representation is too high a cost when just reaching the nearby state is a very achievable goal. Indeed this effect can be seen again on in Figure 18 on the top plot where given enough training time, the model-free H-DDPG algorithm will surpass the performance of the model-based algorithm.

However, the benefits of our algorithm are more obvious when 1) we are resource contrained with respect to training time and 2) the environment is more complex with respect to sparse rewards and long-horizon goals. We can see this in Figure 17 (bottom), Figure 18 (middle, bottom) and in Figure 19 (middle). These are long-horizon challenges, environments in which the agent must sometimes (or always) plan around obstacles to reach the goal, which are of course possible for the model-free H-DDPG algorithm to learn, in theory, but would require much more training time (at a rate increasing with the distance to the goal, i.e. the difficulty of the problem).

On the hardest difficulty, Figure 19 (bottom), even our HGB-DDPG does not perform well. This is due to the difficulty of training the low-level controller from scratch while simultaneously removing seemlingly infeasible edges, and it becomes a challenge of balancing the maximum number of tries before removing an edge with the graph reconstruction frequency (a testament to the more general problem of hyperparameter tuning in reinforcement learning). This parameter would need to be adjusted for this specific environment for better performance, which is unfortunate.

The dips in performance for the HGB-DDPG agent, visible in Figure 17 (bottom), Figure 18 (middle, bottom), and Figure 19 (middle, bottom), are due to the graph reconstruction at every 50 episodes resulting in the return of many previously removed invalid transitions but these are quickly removed again after a few episodes.

B. *Out-of-Distribution Challenges*

The out-of-distribution challenges are where the HGB-DDPG vastly outperforms the H-DDPG. In this case, the agent is pretrained on a PointEnv-Empty-close environment (the easiest) and then the agent is evaluated on a different

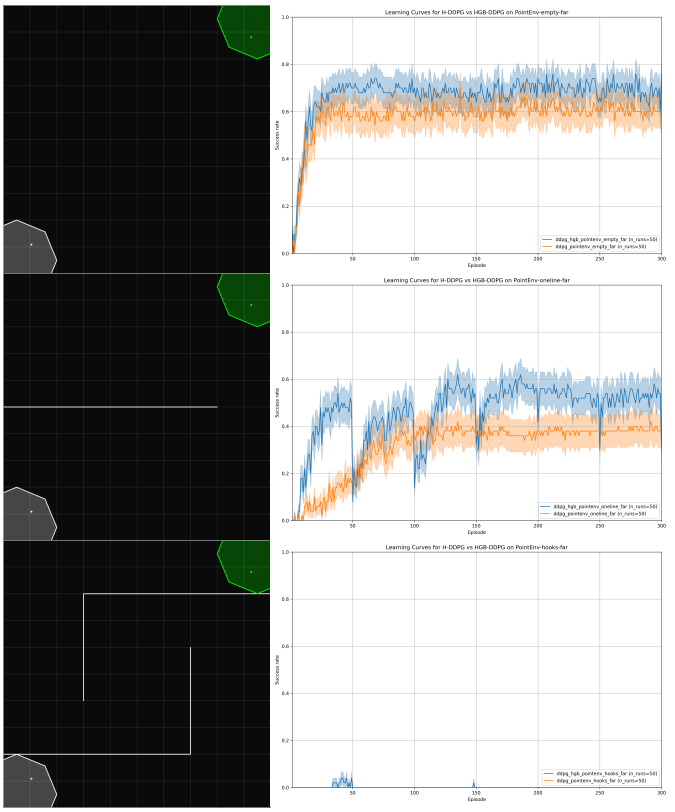

Figure 19: The performance of the H-DDPG baseline (orange) vs our HGB-DDPG algorithm (blue) on the PointEnv environments with distance “far”.

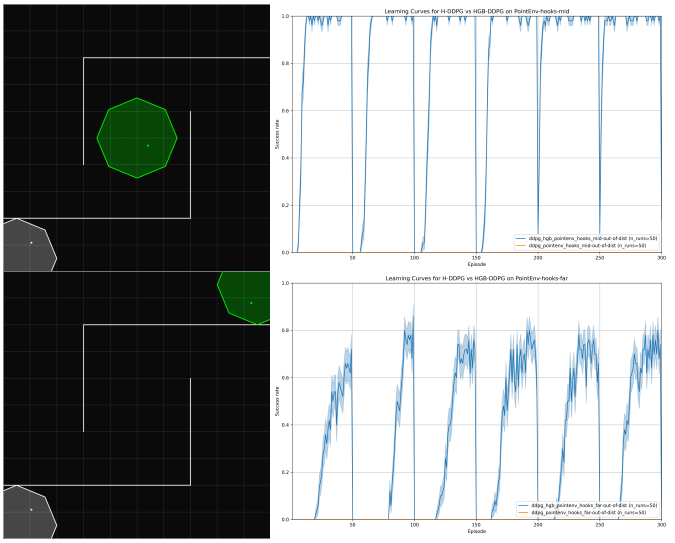

Figure 20: The performance of H-DDPG vs HGB-DDPG when the algorithm is pretrained on PointEnv-Empty-close and then evalutated on PointEnv-Hooks-mid and PointEnv-Hooks-far. The HGB-DDPG algorithm does quite well, often achieving 100% success rate between graph reconstructs, while the H-DDPG algorithm consistently fails with a 0% success rate.

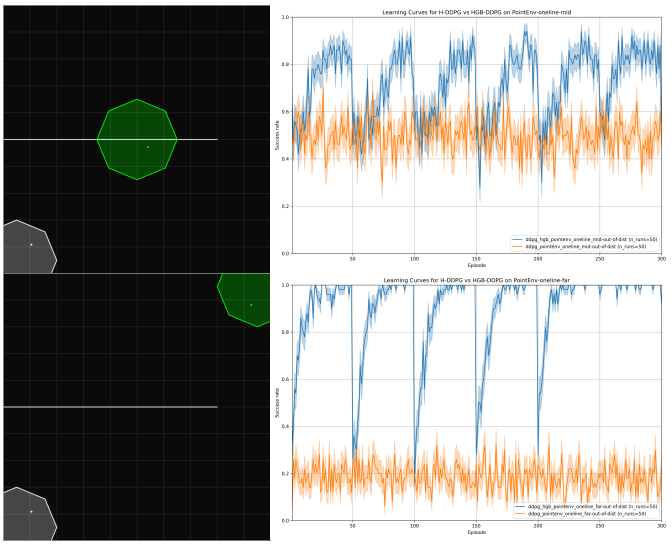

Figure 21: The performance of H-DDPG vs HGB-DDPG when the algorithm is pretrained on PointEnv-Empty-close and then evalutated on PointEnv-OneLine-mid and PointEnv-OneLine-far. The HGB-DDPG algorithm does quite well, often achieving 100% success rate between graph reconstructs, while the H-DDPG algorithm consistently achieves 50% success for PointEnv-OneLine-mid and 20% for PointEnv-OneLine-far.

(harder) environment in a zero-shot fashion, without any further training.

Here, the advantages of the graph module start to become clear as we can see in Figure 20, where we focus on the two hardest difficulties (PointEnv-Hooks-far and PointEnv-Hooks-mid), as well as Figure 21, with two easier difficulties but both with a potential obstacle in the way (PointEnv-OneLine-far and PointEnv-OneLine-mid). The H-DDPG agent consistently fails if there is any obstruction between the agent and the goal that was not present during training, while the HGB-DDPG agent can plan around these new obstacles with ease. Pretraining on an easy environment further benefits the HGB-DDPG agent such that it is now able to perform very well on the hardest environment which was hard-to-impossible to do previously (Compare Figure 20 (bottom) with Figure 19 (bottom)).

Indeed the fact that we are pretraining the low-level controller on an easier task allows the high-level agent to produce a better model by not removing transitions it considers infeasible when they are just not reached due to the performance of the low-level controller. This also results in a better distribution of incoming datapoints as the agent reaches the goal quite quickly after the initial data collection phase. There is potential for improvement here, we note that one of the main problems (indeed a central problem in reinforcement learning) is keeping the set of datapoints in the buffer well distributed over the state space.

Walls	Distance	H-DDPG (μ)	H-DDPG (SEM)	HGB-DDPG (μ)	HGB-DDPG (SEM)
empty	close	**0.8**	0.05	0.72	0.06
empty	mid	0.67	0.07	**0.76**	0.06
empty	far	0.58	0.07	**0.67**	0.07
oneline	close	0.78	0.06	**0.8**	0.06
oneline	mid	0.28	0.06	**0.38**	0.07
oneline	far	0.31	0.06	**0.46**	0.07
hooks	close	0.11	0.04	**0.26**	0.06
hooks	mid	0.2	0.05	**0.35**	0.06
hooks	far	**0.0**	0.0	**0.0**	0.0

Table 6: The mean in-distribution performance in terms of success rate, as well as the standard error of the mean, of the H-DDPG and HGB-DDPG algorithms on the various environments.

Walls	Distance	H-DDPG (μ)	H-DDPG (SEM)	HGB-DDPG (μ)	HGB-DDPG (SEM)
empty	close	**1.0**	0.0	**1.0**	0.0
empty	mid	**1.0**	0.0	**1.0**	0.0
empty	far	0.99	0.01	**1.0**	0.0
oneline	close	**1.0**	0.0	**1.0**	0.0
oneline	mid	0.5	0.07	**0.72**	0.06
oneline	far	0.19	0.05	**0.9**	0.02
hooks	close	0.2	0.06	**0.63**	0.06
hooks	mid	0.0	0.0	**0.85**	0.01
hooks	far	0.0	0.0	**0.33**	0.04

Table 7: The mean out-of-distribution performance in terms of success rate, as well as the standard error of the mean, of the H-DDPG and HGB-DDPG algorithms on the various environments.

C. Interpretability Benefits

We can also inspect the interal graph representation to gain insights into the behaviour of the agent, explain the reason for its choices, identify potential flaws in its model, and even manually correct these flaws if necessary. To this end, we have developed a graphical interface along with the Rust implementation that helps visualize this graph and observe the behaviour of the agent as it acts and learns.

As an example to showcase this we have cherry-picked the best model on the hardest environment (Figure 19 (right) at timestep 49 on a favorable run), something which can be problematic due to the fact that this outlier of great performance could be caused by the model cheating or overfitting or otherwise not actually having learnt it is supposed to have. In this case, however, we can take a look inside the blackbox and make sure. We may already have a theory about how a solution would look like for the PointEnv-Hooks-far environment, but if we plot the agents graph representation on top of the environment (Figure 22), and highlight the plan (yellow) that the agent has generated to reach the goal, we can see exactly that this is a valid solution. We could even edit this graph directly if we see an invalid edge present in the graph, or even a valid edge not present in the graph.

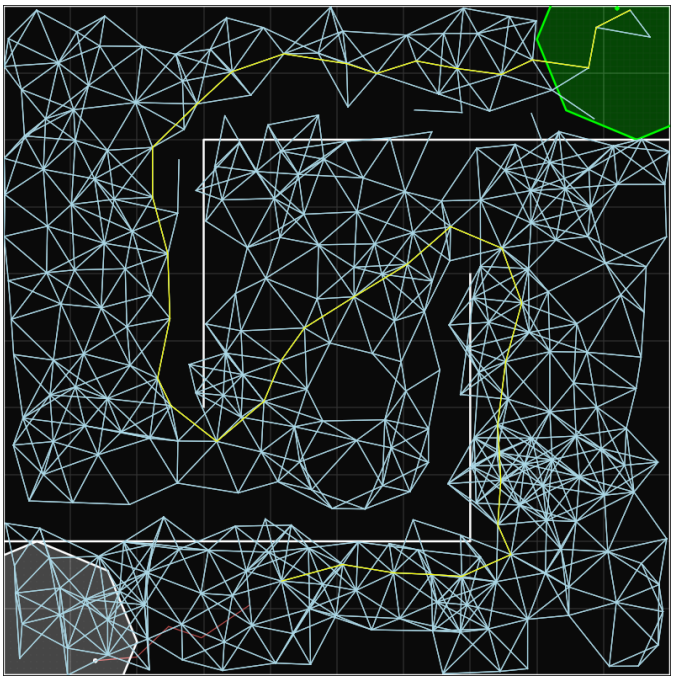

Figure 22: A cherry picked model with great performance visualized such as to showcase the interpretability benefits of the graph module. The sparse graph shows the agents internal representation of the environment, and the yellow path shows the current plan in terms of successive waypoints that the agent is attempting to reach to reach the goal.

We can also observe the high-level agent adapt to an out-of-distribution challenge by visualizing the graph as it changes. Consider Figure 23 where we visualize a HGB-DDPG agent that has been pretrained on PointEnv-Empty and that has just been deployed on PointEnv-Hooks. We can see that it has a well-distributed buffer of states (bottom left) due to our data-collection phase and we can see that the graph is sparsified in a way that still preserves this state-coverage (bottom right), but we can clearly see that the agent is suggesting an infeasible plan (top right) that would have been possible in the environment it was trained on but not in this new environment. Now consider Figure 24, where we can see what happens after letting the agent run for 50 episodes. The plan is now ideal (top right), and we can also

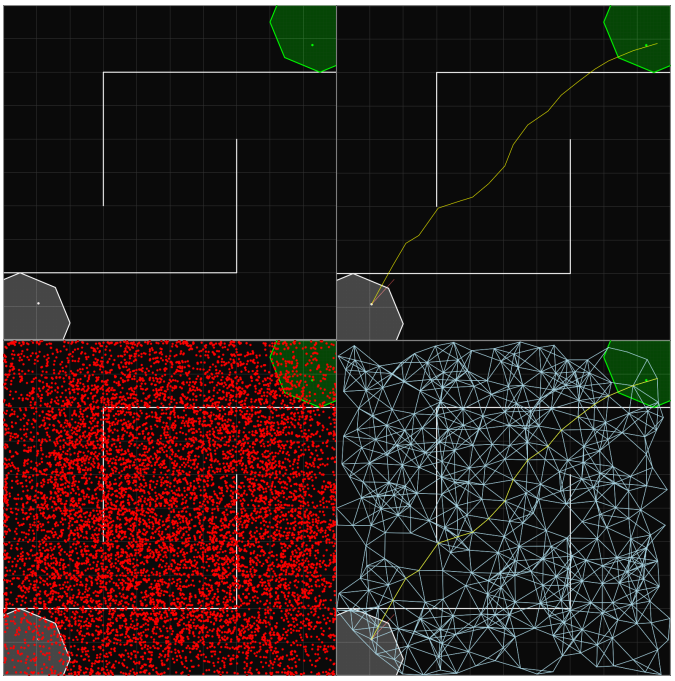

Figure 23: The state of the HGB-DDPG agent, pretrained on PointEnv-Empty-close, just deployed on PointEnv-Hooks-far and visualized in the environment (top left), with the generated plan (top right), the state of the replay buffer (bottom left), and the internal graph resentation including the plan (bottom right).

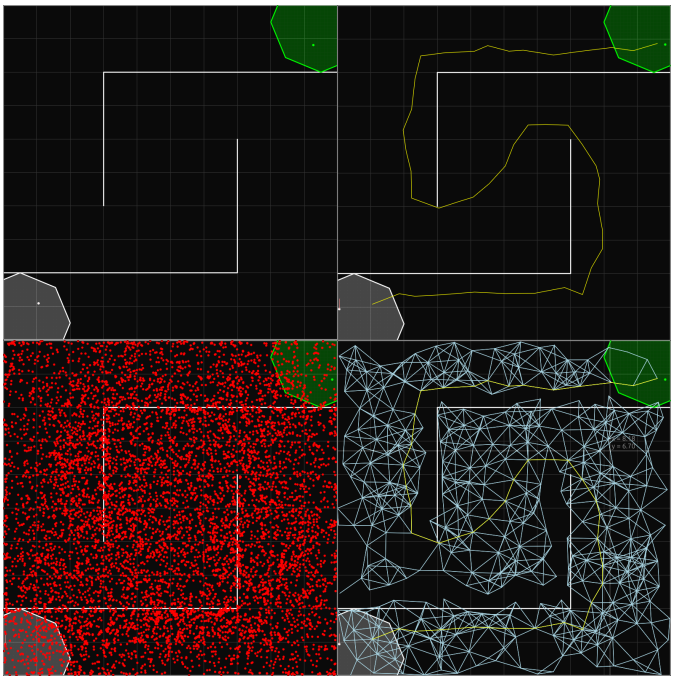

Figure 24: The state of the HGB-DDPG agent, pretrained on PointEnv-Empty-close, deployed on PointEnv-Hooks-far and visualized in the environment (top left), with the generated plan (top right), the state of the replay buffer (bottom left), and the internal graph resentation including the plan (bottom right). The different to Figure 23 is that this agent has run for 50 episodes instead of 0.

see *why*: The agent has removed all those infeasible edges from the graph where it tried and failed to transition over, leaving gaps exactly there where the walls block the path and allowing for a feasible plan to be constructed and executed. At this point, the agent reaches the goal with a very high success rate.

VIII. Discussion & Future Work

We have demonstrated that this method, combining graph-based planning and deep reinforcement learning, is indeed a promising approach providing a number of benefits in certain situations. The quantitative performance benefits on in-distribution challenges are marginal, only present if training time is a constraint and the goal is sufficiently far away, which is to be expected for a model-based vs a model-free method. However, when the low-level controller is pretrained, our method shows superior performance on all challanges without retraining and demonstrates a greater level of generalization and long-horizon planning. Furthermore, the model is interpretable and even editable which are major advantages for certain use-cases.

Due to the difficulties of operating within a young language without a mature ecosystem and the time constraints of this thesis, there are a number of easy improvements that we did not implement. For example, to improve training speed and stability, hindsight experience replay [13] can be implemented as well as distributional RL [36] as proposed in [8].

With better hardware and thus longer training times, a more in-depth hyperparameter analysis can be done and harder and more diverse environments could be tackled. Interesting topics might include how the interpretability of the graph transfers to non-navigational environments (i.e. where there is no notion of euclidean distance) and how to deal with the problem of perceptual aliasing in environments with high-dimensional observations (i.e. where similar observations might be temporally far apart).

The most interesting topics for future work certainly include addressing the two problematic assumptions we make on the environment in our thesis: 1) That we have access to the true underlying distance function between any two states, and 2) that we assume the ability to spawn the agent in locations randomly distributed over the state-space. For the former, the next logical step is to make use of the learned distance function in the critic network and make it work using, for example, the tricks proposed by [8] and [9] such as pessimistically aggregating over ensembles of critic networks, using distributional RL, and normalizing observa-

tions correctly. For the latter, there is a lot of potential for improved exploration strategies that can be implemented to make use of the graph such as explicitly visiting nodes at the frontiers of the exploration horizon according to the graph and then randomly exploring from there or keeping track of information on the frequency of finding novel states nearby the nodes in the graph and then sampling high-potential exploration sub-goals intelligently with e.g. Thompson sampling.

IX. Acknowledgments

I would like to thank my supervisor, Dr. Matthia Sabatelli, for his guidance and support along the way, and to thank the University of Groningen for providing me with an excellent education. This thesis is especially dedicated to the memory of Dr. Marco Wiering, who was a friend and a mentor to me.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *CoRR*, 2012, [Online]. Available: http://arxiv.org/abs/1207.4708

[2] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] C. Glanois *et al.*, "A Survey on Interpretable Reinforcement Learning." 2022.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[5] 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, doi: 10.1109/TSSC.1968.300136.

[6] T. M. Moerland, J. Broekens, and C. M. Jonker, "Model-based Reinforcement Learning: A Survey," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2006.16712

[7] R. E. Korf, "Real-time heuristic search," *Artificial intelligence*, vol. 42, no. 2–3, pp. 189–211, 1990.

[8] B. Eysenbach, R. Salakhutdinov, and S. Levine, "Search on the Replay Buffer: Bridging Planning and Reinforcement Learning," *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1906.05253

[9] M. Laskin, S. Emmons, A. Jain, T. Kurutach, P. Abbeel, and D. Pathak, "Sparse Graphical Memory for Robust Planning," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2003.06417

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] M. L. Puterman, "Markov Decision Processes: Discrete Stochastic Dynamic Programming." John Wiley & Sons, Inc., 1994.

[12] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," *arXiv preprint arXiv:2005.01643*, 2020.

[13] M. Andrychowicz *et al.*, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[14] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[15] M. Watter, J. T. Springenberg, J. Boedecker, and M. A. Riedmiller, "Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images," *CoRR*, 2015, [Online]. Available: http://arxiv.org/abs/1506.07365

[16] R. Sasso, M. Sabatelli, and M. A. Wiering, "Multi-Source Transfer Learning for Deep Model-Based Reinforcement Learning," *Transactions on Machine Learning Research*, 2023, [Online]. Available: https://openreview.net/forum?id=1nhTDzxxMA

[17] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal Value Function Approximators," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., in Proceedings of Machine Learning Research, vol. 37. Lille, France: PMLR, 2015, pp. 1312–1320. [Online]. Available: https://proceedings.mlr.press/v37/schaul15.html

[18] P. Dayan and G. E. Hinton, "Feudal Reinforcement Learning," in *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles, Eds., Morgan-Kaufmann, 1992, p. . [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1992/file/d14220ee66aeec73c49038385428ec4c-Paper.pdf

[19] M. Towers *et al.*, "Gymnasium." [Online]. Available: https://github.com/Farama-Foundation/Gymnasium

[20] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-Explore: a New Approach for Hard-Exploration Problems." 2021.

[21] R. Diestel, *Graph Theory*, 5th ed. Springer Publishing Company, Incorporated, 2017.

[22] A. Shimbel, "Structure in communication nets," in *Proceedings of the symposium on information networks*, 1954, pp. 199–203.

[23] B. Roy and others, "Transitivité et connexité," *CR Acad. Sci. Paris*, vol. 249, no. 216–218, p. 182–183, 1959.

[24] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[25] Z. Huang, F. Liu, and H. Su, "Mapping State Space using Landmarks for Universal Goal Reaching," *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1908.05451

[26] L. Zhang, G. Yang, and B. C. Stadie, "World Model as a Graph: Learning Latent Landmarks for Planning," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2011.12491

[27] N. Savinov, A. Dosovitskiy, and V. Koltun, "Semi-parametric Topological Memory for Navigation," *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1803.00653

[28] S. Nasiriany, V. H. Pong, S. Lin, and S. Levine, "Planning with Goal-Conditioned Policies," *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1911.08453

[29] C. Hoang, S. Sohn, J. Choi, W. Carvalho, and H. Lee, "Successor feature landmarks for long-horizon goal-conditioned reinforcement learning," *Advances in neural information processing systems*, vol. 34, pp. 26963–26975, 2021.

[30] K. Liu, T. Kurutach, C. Tung, P. Abbeel, and A. Tamar, "Hallucinative topological memory for zero-shot visual planning," in *International Conference on Machine Learning*, 2020, pp. 6259–6270.

[31] D. S. Chaplot, R. Salakhutdinov, A. Gupta, and S. Gupta, "Neural topological slam for visual navigation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 12875–12884.

[32] W. Shang, A. Trott, S. Zheng, C. Xiong, and R. Socher, "Learning world graphs to accelerate hierarchical reinforcement learning," *arXiv preprint arXiv:1907.00664*, 2019.

[33] J. Kim, Y. Seo, and J. Shin, "Landmark-guided subgoal generation in hierarchical reinforcement learning," *Advances in neural information processing systems*, vol. 34, pp. 28336–28349, 2021.

[34] C. Cadena *et al.*, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[35] Y. Eldar, M. Lindenbaum, M. Porat, and Y. Zeevi, "The farthest point strategy for progressive image sampling," *IEEE Transactions on Image Processing*, vol. 6, no. 9, pp. 1305–1315, 1997, doi: 10.1109/83.623193.

[36] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," *CoRR*, 2017, [Online]. Available: http://arxiv.org/abs/1707.06887

[37] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[38] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in Rust," *Communications of the ACM*, vol. 64, no. 4, pp. 144–152, 2021.

[39] G. Thomas, [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[40] Stack Overflow, "Stack Overflow Developer Survey 2023." Accessed: Jun. 13, 2023. [Online]. Available: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages

[41] L. Mazare, "tch-rs." [Online]. Available: https://https//https://github.com/LaurentMazare/tch-rs

[42] C. Lowman, "dfdx." [Online]. Available: https://https//https://github.com/coreylowman/dfdx

[43] N. Simard, "burn." [Online]. Available: https://https//https://github.com/tracel-ai/burn

[44] L. Mazare, "candle." [Online]. Available: https://https//github.com/huggingface/candle

[45] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1509.02971

[46] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 5, pp. 834–846, 1983, doi: 10.1109/TSMC.1983.6313077.

[47] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the Brownian motion," *Physical review*, vol. 36, no. 5, p. 823–824, 1930.