Stop! Planner Time: Metareasoning for Probabilistic Planning Using Learned Performance Profiles

Matthew Budd, Bruno Lacerda, Nick Hawes

Oxford Robotics Institute, University of Oxford
{mbudd, bruno, nickh}@robots.ox.ac.uk

Abstract

The metareasoning framework aims to enable autonomous agents to factor in planning costs when making decisions. In this work, we develop the first non-myopic metareasoning algorithm for planning with Markov decision processes. Our method learns the behaviour of anytime probabilistic planning algorithms from performance data. Specifically, we propose a novel model for metareasoning, based on contextual performance profiles that predict the value of the planner's current solution given the time spent planning, the state of the planning algorithm's internal parameters, and the difficulty of the planning problem being solved. This model removes the need to assume that the current solution quality is always known, broadening the class of metareasoning problems that can be addressed. We then employ deep reinforcement learning to learn a policy that decides, at each timestep, whether to continue planning or start executing the current plan, and how to set hyperparameters of the planner to enhance its performance. We demonstrate our algorithm's ability to perform effective metareasoning in two domains.

Introduction

Rational agents should reason about the consequences of their actions. However, this reasoning process itself has its own consequences: thinking is a physical process that requires time and energy. In many systems this issue can be ignored, in particular where the costs of any computation are negligible compared to the costs of actions. In other systems, the real-world effects of reasoning cannot be discounted. As an example, for mobile robots thinking and acting are often closely coupled. An autonomous vehicle can use its finite battery capacity on its on-board computer or its motors. To what extent is the energy saved by executing a better motion plan offset by the energy cost of additional computation? This scenario is illustrated in Figure 1. Of course, this assumes that the agent is reasoning *online*. If it is able to pre-plan all of its tasks offline, planning effort is negligible at runtime and no metareasoning tradeoff exists.

Over the last 40 years, the field of rational metareasoning has developed bounded optimal algorithms for reasoning about these tradeoffs (Russell and Wefald 1991; Cox and Raja 2011; Hay et al. 2012). Recent problem settings have included motion planning and heuristic search (Burns, Ruml, and Do 2013; Sung, Kaelbling, and Lozano-Pérez 2021; Bhatia et al. 2022). Optimal exact metareasoning is polynomially harder than reasoning (Lin et al. 2015), so all practical algorithms carry out approximate metareasoning. A common simplification is to only consider the immediate effects of a single reasoning step: this is known as the *meta-greedy* or *meta-myopic* assumption (Russell and Wefald 1989, 1991).

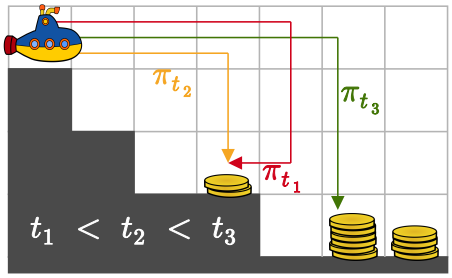

Figure 1: A deep sea treasure problem. Each policy represents the results of planning for a longer duration, and achieves reward indicated by the number of coins. π_{t_2} would be optimal overall if the extra reward gained by π_{t_3} is less than the cost (duration) of the additional planning time.

Under the umbrella of metareasoning problems, we focus specifically on metalevel control of anytime algorithms (Cox and Raja 2011). In this setting, reasoning is carried out by an *object-level* process, which is an anytime interruptible algorithm. This class of algorithms provides its current solution at any time during computation, but the longer the algorithm runs the better the solution will be. The object-level process is supervised by a *meta-level* process, which observes the object-level algorithm's state. Metalevel actions consist of allowing object-level computation to continue, or terminating the algorithm and acting using its current solution.

We wish to apply metareasoning to probabilistic planning with Markov decision processes (MDPs). Recent works have used deep reinforcement learning (RL) to learn non-myopic metalevel policies (Sung, Kaelbling, and Lozano-Pérez 2021; Bhatia et al. 2022). However, these works apply only to object-level algorithms where the current solution quality is known exactly, for example the current minimum path length in RRT*. Relaxing the solution quality observability

assumption is one of the main contributions of this paper and is crucial for the class of probabilistic planning algorithms which we consider, since they often maintain only an approximation of the solution quality. We go on to develop a novel contextual metalevel MDP formulation which enables an RL agent to learn to infer solution quality from limited information about the planner and problem instance.

Compared to existing MDP planning metareasoning works, we avoid meta-greedy approximations and instead directly learn the value function of the metalevel MDP. Our method is agnostic to the choice of object-level algorithm, rather than relying on properties of one specific algorithm as Lin et al. (2015) do. Finally, we use the hyperparameter tuning method proposed by Bhatia et al. (2022) to give the metalevel agent more control over the object-level algorithm. This enables switching between parameters that emphasise fast but low-quality solutions vs slow and high-quality solutions, alongside deciding when to execute the current solution.

Related Work

The metalevel control problem is commonly posed as a metalevel Markov decision process (Hay et al. 2012). Closest to our setting, Lin et al. (2015) formulate online stochastic shortest path (SSP) MDP planning metareasoning as a metalevel MDP. In common with many metareasoning algorithms, they are unable to exactly solve this model and must use a meta-greedy approximation (Russell and Wefald 1991). This is the simplifying assumption that the reasoning agent can only choose between executing now, or executing after a single additional reasoning step. The approximation ignores the value of further computation, but can be shown to be optimal in the case of diminishing returns and non-decreasing computation cost (Callaway et al. 2018). However, solution quality plots from real algorithms and problems rarely show smoothly diminishing returns. One example from motion planning is the sharp change caused by a change of homotopy class.

One way to achieve non-myopic metalevel behaviour is to build a probabilistic model of the algorithm's solution quality evolution from a dataset, and use this *performance profile* to determine the optimal execution point (Hansen and Zilberstein 2001). As long-horizon model-based planning is computationally expensive, the approach is limited to short horizons to avoid excessive metareasoning overhead. Recent works improve on this approach by learning the performance profiles, and metalevel control policies, from experience (Bhatia et al. 2022; Sung, Kaelbling, and Lozano-Pérez 2021). These works focus on deterministic search problems, and make metalevel decisions based on a fixed utility function that combines the object-level solution quality and computation time. This contrasts with the setting of Lin et al. (2015), where the effects and costs of reasoning depend on the current object-level MDP state. By defining utility as a function of object-level solution quality, these methods assume that the current solution quality is always known. In this work we relax that assumption in order to apply deep RL-based metareasoning to probabilistic planning.

Callaway et al. (2018) present a method that estimates the value of computation using a weighted combination of myopic and full-knowledge value of information (VOI) features. These weights are learned from an object-level problem distribution using Bayesian optimisation. Their algorithm outperforms those that depend only on myopic VOI features. However, the method assumes perfect knowledge of the metalevel MDP dynamics, which is infeasible for a practical planning algorithm. It also scales poorly with increasing complexity of the object-level problem, due to the online calculations required to compute the values of VOI features.

Indeed, Callaway et al. note that metareasoning is only useful when object-level reasoning is significantly more expensive than metareasoning. It is counterproductive to spend more time deciding whether to think than actually thinking. Other work which explicitly analyses the overhead of metareasoning (Milli, Lieder, and Griffiths 2017) concurs with this point. This explains the strong simplifying assumptions commonly used in metareasoning algorithms, and justifies using deep RL-trained policies for these problems: the cost of querying actions from a trained policy is minimal.

A different but related problem setting is situated temporal planning, which aims to maximise the probability of finding a valid deterministic plan within a deadline (Cashmore et al. 2018). Rather than a metalevel agent deciding when to execute, metareasoning takes place on the level of deciding which nodes to expand in a search tree. Some more recent extensions also attempt to minimise the cost-to-goal of the chosen plan (Shperberg et al. 2020). However, these methods do not directly minimise the combination of planning cost and execution cost as ours does.

Preliminaries

Markov Decision Processes. A stochastic shortest path (SSP) MDP is a tuple $\mathcal{M} = \langle S, \mathit{init}, A, T, C, G \rangle$, where S is a finite set of states; $\mathit{init} : S \to [0, 1]$ is an initial state distribution; A is a finite set of actions; $T : S \times A \times S \to [0, 1]$ is a probabilistic transition function; $C : S \times A \to \mathbb{R}_{\geq 0}$ is a cost function; and $G \subset S$ is a set of absorbing, zero-cost goal states.

When the initial state is deterministic, we will replace *init* with s_0 in the definition of $\mathcal{M}$. Actions are selected using a stationary policy $\pi : S \to A$. A policy is *proper* in state s if the probability of reaching a goal state when starting from s is 1. For an SSP MDP there must exist a proper policy from the initial state. All improper policies must have infinite expected cumulative cost from states that they are improper in. It can be shown that there exists a minimum cost proper policy (Mausam and Kolobov 2012). A policy is *complete* if it provides an action for all states $s \in S$. Alternatively, a *partial* policy π has a domain $S_\pi \subset S$ and is complete in S_π. For a proper policy π, let $V^\pi_\mathcal{M} = \mathbb{E}_{\mathcal{M}, s_0 \sim \mathit{init}} [\sum_{i=0}^{\infty} C(s_i, \pi(s_i))]$, where s_i is a random variable representing the state visited at the i-th timestep, denote the expected cumulative cost of executing policy π.

Problem Formulation

In this section, we pose our problem as minimising the expected cost of a metalevel SSP MDP $\mathcal{M}^M$ that observes and controls the behaviour of an object-level probabilistic planner. Similarly to previous work (Milli, Lieder, and Griffiths 2017;

Sung, Kaelbling, and Lozano-Pérez 2021; Bhatia et al. 2022), we assume that object-level problems are drawn from a distribution $p(\mathcal{M})$ over a set of decision problems D. Specifically, the planner operates on an object-level SSP MDP instance $\mathcal{M} = \langle S, s_0, A, T, C, G \rangle$, where $\mathcal{M}$ is sampled according to $p(\mathcal{M})$. The object-level planner has a set of hyperparameters $\{\Delta_i\}_{i=1}^{N_\Delta}$, which alter its behaviour. Each hyperparameter has a set of valid values $\Delta_n = \{\delta_1, \delta_2, \ldots, \delta_{k_n}\}$.

The metalevel agent has no direct control over the actions taken in the object-level MDP. At each metalevel timestep, the metalevel agent observes the object-level planner's configuration (its internal state) χ and chooses a metalevel action. Let π_χ be the object-level planner's current best solution, represented within its current configuration χ. Metalevel actions are to a) continue planning for another timestep, altering hyperparameter values if desired, or b) stop planning and execute the current best solution π_χ. Continuing planning for one timestep runs the planner for time τ (which may correspond to many object-level planning iterations), and incurs a fixed instance-dependent planning cost $\lambda(\mathcal{M})$.

The object-level planner's current best policy π_χ may not be complete for all states reachable under that policy from s_0. We assume that π_χ is combined with a *default policy* which is complete and proper in state s_0. This new complete and proper policy, $\tilde{\pi}_\chi$, follows π_χ when it is defined, and the default policy when outside of π_χ's support.

We now define our metalevel reasoning model.

Definition 1 *Given a stochastic shortest path object-level MDP $\mathcal{M} = \langle S, s_0, A, T, C, G \rangle$, the metalevel MDP is an SSP MDP $\mathcal{M}^M = \langle S^M, s_0^M, A^M, T^M, C^M, G^M \rangle$, where:*

- $S^M = X \cup \{\text{DONE}\}$ *where X is the set of all possible configurations of the object-level planner and* DONE *is a terminal state;*
- $s_0^M = \chi_0$ *where χ_0 is the initial configuration of the object-level planner;*
- $A^M = (\{\text{PLAN}\} \times \Delta_1 \times \ldots \times \Delta_{N_\Delta}) \cup \{\text{EXEC}\}$*, i.e. the agent may choose to let the planner run for another metalevel timestep (τ time in planner time) using the specified hyperparameter values, or execute the current best solution;*
- $T^M : S^M \times A^M \times S^M \to [0, 1]$ *is defined as follows:*

$$T^M(\chi, a, \chi') = \begin{cases} 1 & \text{if } \chi \in X, \\ & \chi' = \text{DONE and} \\ & a = \text{EXEC} \\ p(\chi' \mid \mathcal{M}, \chi, a) & \text{if } \chi, \chi' \in X \text{ and} \\ & a \neq \text{EXEC} \\ 0 & \text{otherwise}, \end{cases}$$

where $p(\chi' \mid \mathcal{M}, \chi, a)$ is the probability of the object-level planner transitioning to configuration χ', given that it was in configuration χ and ran on the object-level MDP $\mathcal{M}$ for one metalevel timestep, using the hyperparameters specified by a;

- $C^M : S \times A \to \mathbb{R}_{\geq 0}$ *is defined as follows:*

$$C^M(\chi, a) = \begin{cases} \lambda(\mathcal{M}) & \text{if } a = \text{PLAN} \\ V_{\mathcal{M}}^{\tilde{\pi}_\chi} & \text{if } a = \text{EXEC}, \end{cases}$$

where $\lambda(\mathcal{M})$ is the cost of planning for one metalevel timestep in MDP $\mathcal{M}$ and π_χ is the object-level planner's current policy; and

- $G^M = \{\text{DONE}\}$.

A cost-minimising metalevel policy $\pi^M : S^M \to A^M$ for $\mathcal{M}^M$ minimises the sum of expected total planning cost and expected object-level policy execution cumulative cost. In the definition of $\mathcal{M}^M$, we assume that X contains all the internal features necessary to build Markovian internal state dynamics. These dynamics might still be probabilistic because the object-level algorithm itself will typically have a probabilistic nature (e.g. successor sampling in trial-based search). Furthermore, to ensure that the metalevel cost minimisation objective is meaningful, we assume that the cost of thinking $\lambda(\mathcal{M})$ uses the same unit as the object-level MDP cost. For example, the metalevel problem might be minimising total (planning + execution) time or energy spent.

Note that $V_{\mathcal{M}}^{\tilde{\pi}_\chi}$ can be interpreted as the *quality profile* of the anytime planner, i.e., given a planner configuration, it provides a measure of the quality of the solution. With full knowledge of $p(\chi' \mid \mathcal{M}, \chi, a)$ and $V_{\mathcal{M}}^{\tilde{\pi}_\chi}$, the metalevel MDP could be solved to find an optimal metalevel policy. However, two key factors make this infeasible.

Firstly, for practical planners, the configuration space X is prohibitively large, making offline solution impractical. Online solution raises the spectre of metareasoning overhead. Even more critically, the configuration dynamics depend on the MDP instance $\mathcal{M}$. Evaluating the transition dynamics of X on $\mathcal{M}$ is at least as hard as solving $\mathcal{M}$ itself, negating the value of performing metareasoning (Lin et al. 2015).

Secondly, although the configuration includes a representation of the current policy $\tilde{\pi}_\chi$, the expected cost $V_{\mathcal{M}}^{\tilde{\pi}_\chi}$ of this solution is not necessarily readily accessible. Many MDP solution algorithms simultaneously carry out policy improvement and policy value estimation. They will only precisely calculate the expected cost of the optimal policy at convergence. Of course, the current policy can be evaluated on the current MDP instance, but this must be done sparingly to avoid metareasoning overhead. Next, we propose an abstraction of the metalevel MDP which addresses these issues.

Method

Abstracting the Metalevel MDP

Inspired by Bhatia et al. (2022), we begin by abstracting X to Ω, where $\Phi_X : X \to \Omega$ provides a smaller representation of *algorithm features*. Features $\Phi_X(\chi)$ are hand-designed and algorithm specific, and aim to act as surrogates for the full configuration χ. Examples include the total number of trials or state expansions carried out, the depth of a Monte-Carlo search tree, statistical properties of the lengths of sampled trajectories, or value function estimates.

To learn a policy that is good in expectation across the space of all MDPs in the domain D of object-level problems, we extend the state-representation of the abstracted metalevel MDP to include a *context vector* representing the object-level instance $\mathcal{M}$ being solved. By learning the correlations between context vector values and algorithm performance, we

can approximate the effects of the object-level MDP instance on $p(\chi' \mid \mathcal{M}, \chi, a)$. The information contained in the context vector is based on what the system could be reasonably expected to know about the problem it is solving, and thus is problem-dependent. For example, in our race track domain experiments, the context vector contains the maximum speed and probability of acceleration failure for the agent's car. We denote the context set as Ψ, and define the function $\Phi_D : D \to \Psi$ which maps an MDP $\mathcal{M}$ to its context vector $\Phi_D(\mathcal{M})$. The cost of thinking $\lambda(\mathcal{M})$ is also included in the context vector, as it is also a constant property of the object-level MDP instance. The context vector can be treated as a part of the state of the abstract metalevel MDP which is set in the initial state according to $p(\mathcal{M})$, and remains fixed during execution.

We can now define our abstract metalevel MDP:

Definition 2 *The abstract metalevel MDP is defined as* $\hat{\mathcal{M}}^M = \langle \hat{S}^M, \hat{init}_0^M, A^M, \hat{T}^M, \hat{C}^M, G^M \rangle$*, where:*

- $\hat{S}^M = \Omega \times \Psi \cup \{\text{DONE}\}$*, i.e., a state is either of the form* (ω, ψ) *where* ω *is the current value of the algorithm features and* ψ *is a context vector representing the object-level MDP being considered; or is the final state* DONE *which represents sending the current object-level policy for execution;*
- *For* $s = (\omega, \psi) \in \hat{S}^M$*:*

$$\hat{init}_0(s) = \begin{cases} \int_{\{\mathcal{M} \mid \Phi_D(\mathcal{M}) = \psi\}} p(\mathcal{M}) d\mathcal{M} & \text{if } \omega = \omega_0 \\ 0 & \text{otherwise,} \end{cases}$$

  where ω_0 *are the planner initial feature values (i.e. the initial context vector value distribution is calculated via marginalisation over* $p(\mathcal{M})$*);*
- *As with the metalevel MDP,* $\hat{T}^M$ *moves* $\mathcal{M}$ *to state* DONE *with probability 1 when action* EXEC *is selected. For* $s = (\omega, \psi), s' = (\omega', \psi) \in \hat{S}$ *and* $a \in A^M$*,* $\hat{T}^M(s, a, s')$ *is the probability of the planner moving to a configuration* χ' *such that* $\Phi_X(\chi') = \omega'$*, given that its algorithm features were* ω *and it ran for one metalevel timestep, using the hyperparameters specified by* a*;*
- *The cost function* $\hat{C}^M$ *is only action-dependent and defined as* C^M*.*

Note that the action space and goal state of the abstract metalevel MDP are the same as the metalevel MDP in Def. 1. Its behaviour is similar to the metalevel MDP, but it (i) operates over a small set of algorithm features rather than its full internal configuration, to achieve scalability; and (ii) considers all possible MDPs in D using a context vector which is fixed during execution and distributed in the initial state according to $p(\mathcal{M})$. We do not have access to a closed-form definition of $\hat{T}^M$. Hence, next we propose a general deep RL algorithm that learns the value function and optimal policy for $\hat{\mathcal{M}}$. This approach assumes that the transition function remains Markovian when defined over a state-space based on abstraction Φ_X. In future work, we will investigate how to relax this assumption by allowing the agent to reason using a history of its past actions and observations, e.g., using a recurrent, LSTM or transformer architecture.

Algorithm 1: General deep RL on the abstract metalevel MDP

Input: Problem distribution $p(\mathcal{M})$, object-level algorithm PLANNERALG, default policy $\tilde{\pi}$, number of training episodes M_e, object-level planning time per metalevel timestep τ
Output: Trained policy π_θ^M.

Initialise π_θ^M randomly
for episode = $1, \ldots, M_e$ **do**
  Sample $\mathcal{M}$ from D according to $p(\mathcal{M})$
  $\psi \leftarrow$ calculate MDP context $\Phi_D(\mathcal{M})$
  $\chi \leftarrow$ initial configuration of PLANNERALG for $\mathcal{M}$
  $\omega \leftarrow$ calculate features $\Phi_X(\chi)$
  $\hat{s} \leftarrow (\omega, \psi)$
  repeat {run episode}
    $\hat{a} \leftarrow$ select action using $\pi_\theta^M(\hat{s})$
    if $\hat{a}$ = EXEC **then**
      $\tilde{\pi}_\chi \leftarrow$ best solution from current configuration χ, completed by $\tilde{\pi}$
      $\hat{c} \leftarrow$ estimate $V_{\mathcal{M}}^{\tilde{\pi}_\chi}$ using Monte-Carlo simulation
      $\hat{s}' \leftarrow$ DONE
    else $\{\hat{a} = (\text{PLAN}, \delta_1, \ldots, \delta_{N_\Delta})\}$
      $\chi \leftarrow$ update hyperparameters to $\delta_1, \ldots, \delta_{N_\Delta}$
      $\chi \leftarrow$ run planner for time τ
      $\hat{c} \leftarrow \lambda(\mathcal{M})$
      $\omega \leftarrow$ calculate new features $\Phi_X(\chi)$
      $\hat{s}' \leftarrow (\omega, \psi)$
    end if
    Store $(\hat{s}, \hat{a}, \hat{c}, \hat{s}')$
    $\hat{s} \leftarrow \hat{s}'$
  until $\hat{s} = \text{DONE}$
  Update parameters of π_θ^M using stored episode data
end for
return π_θ^M

Unlike previous metalevel MDP formulations (Sung, Kaelbling, and Lozano-Pérez 2021; Bhatia et al. 2022), the state of our abstract metalevel MDP does not include the performance of the current policy. Thus, we cannot use a closed-form cost function based on a utility function that combines the current time and solution quality, as those methods do. Instead, we must learn the cost function from experience, using evaluations of the final policy cost at execution. This means that we are implicitly learning contextual and feature-based performance profiles for the planner algorithm.

Deep RL for the Abstract Metalevel MDP

An algorithm for training the RL metareasoning agent is given in Algorithm 1. To simplify the presentation, we abstract away the specifics of the exact RL algorithm used, although the algorithm structure is based on that proposed by Bhatia et al. (2022), which is similar to DQN (Mnih et al. 2015). Adapting our approach to an actor-critic framework is straightforward. Specifically, replacing DQN with an algorithm that supports a hybrid action space, e.g. Hybrid SAC (Delalleau et al. 2019), would enable continuous hyperparameter tuning alongside the discrete plan/act actions.

In Algorithm 1, each training episode corresponds to solv-

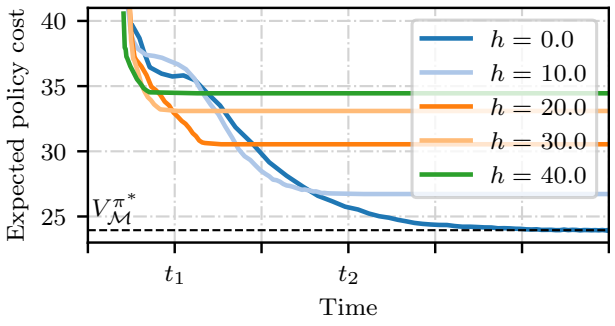

Figure 2: The effect of differing heuristic weights on solution convergence for WBRTDP, on a racetrack problem instance.

ing and then executing a policy in a single problem instance sampled from the problem distribution $p(\mathcal{M})$ (line 3). A new object-level planner is initialised for each problem instance (line 5). In line 7 the initial state contains the initial values of the planner features plus the context vector $\Phi_D(\mathcal{M})$ for the sampled problem, which does not change during the episode. At each timestep, the metalevel agent can choose to execute the current best object-level policy (completed by the default policy) $\tilde{\pi}_\chi$, (line 10) or to continue planning (line 14).

If the agent chooses to execute, we estimate the expected cost of the completed policy $\tilde{\pi}_\chi$, by sampling N trajectories of $\mathcal{M}$ under $\tilde{\pi}_\chi$, and taking their average cumulative cost (line 12). With this final cost, the metalevel MDP transitions to its final absorbing state DONE (line 13) and the episode ends (line 23). If the agent chooses to continue planning, the action specifies the object-level planner hyperparameters to use for the next timestep (line 15). These are used by the object-level planner to plan for a single metalevel timestep (line 16) which, we recall, can correspond to many iterations of the object-level planner. The thinking step costs $\lambda(\mathcal{M})$ (line 17), and causes the features $\Phi_X(\chi)$ to evolve (line 18). This is reflected in the updated state (line 19).

Experienced state transitions are used to update the learned policy (line 24), depending on the specific RL algorithm. As the RL agent chooses when to terminate deliberation, the metalevel episode length is not fixed. In practice, we set a maximum number of timesteps per episode, and force the action choice to be EXEC after that number. The policy-dependent episode length has the advantage that the algorithm performance data collected is only that which is relevant to the metareasoning task. By comparison, sequence prediction methods such as that proposed by Sung, Kaelbling, and Lozano-Pérez (2021) require training on performance profiles that are run to full convergence. This can take a long time due to diminishing returns as the algorithm converges.

Experiments

Object-Level Algorithm: Weighted BRTDP

Automatic hyperparameter tuning is generally useful to improve algorithms' performance (Falkner, Klein, and Hutter 2018), and one would expect a hyperparameter-tuning learning agent to learn the optimal hyperparameters to use for a problem distribution $p(\mathcal{M})$. However, we wish to demonstrate the *metareasoning-specific benefits* of hyperparameter tuning. To do so, we introduce a hyperparameterised version of BRTDP (McMahan, Likhachev, and Gordon 2005), which we call Weighted BRTDP (WBRTDP). By allowing the metalevel agent to change the weighting hyperparameter, we can give it more control over the object-level algorithm's behaviour. Specifically, we expect it to learn to use weights that lead to fast, imperfect solutions when the cost of planning λ is high relative to the cost of object-level policy execution.

The algorithm is inspired by variants of A* and AO* (Hansen and Zhou 2007; Bonet and Geffner 2012) which use inadmissible heuristics to bias the algorithms' convergence behaviour. WBRTDP allows switching between different lower-bound heuristics during the search process, with minimal overhead. Figure 2 shows WBRTDP running on a single instance of the racetrack problem (described below), using differing fixed heuristic values for each line. With the admissible ($h_l = 0.0$) fixed heuristic for all states, the algorithm converges to the true optimal policy some time after time t_2. However, early in the search process (e.g. at t_1), the inadmissible heuristic values enable much faster convergence to a suboptimal policy. We detail WBRTDP and the features we used for metareasoning in the supplementary material.

Deep RL Algorithm

In our experiments we use DQN (Mnih et al. 2015), a frequently-used model-free deep RL algorithm. The action space for the algorithm is the EXEC action or choosing to set the heuristic weight to a value $h_l \in \{0.0, 10.0, 20.0, 30.0\}$. Further details on the implementation of DQN for our method can be found in the supplementary material.

Experiment Domains

We generate object-level MDP problems from problem distributions in two domains.

The **deep sea treasure (DST) domain** is similar to that in Figure 1, but larger. Object-level solution improvement in this domain can result from the planner optimising its path to a given treasure, or finding a better treasure. Finding a better treasure will generally cause a larger improvement in the solution expected cost. This makes the domain interesting for metareasoning problems: performance profiles will typically not show diminishing returns behaviour.

The domain was originally proposed as a deterministic multi-objective episodic MDP by Vamplew et al. (2011). We convert it to a probabilistic single-objective SSP. The agent starts in the top left corner of a grid world with randomly sampled dimensions, with actions that probabilistically accelerate it in the eight cardinal directions. With probability P_{fail}, the acceleration action fails. Each action (including no acceleration) has a fixed time cost of 1. Final rewards for gathering treasure are converted to costs. The cost of a treasure-collection action is the maximum value of treasure generated in this problem class minus the value of the treasure collected. The default policy is to proceed directly downwards from the current state. The shape of the sea floor, treasure locations, treasure values and $P_{\text{fail}} \sim \text{Uniform}(0, 0.3)$ are randomly sampled. Deeper states correlate with higher average treasure values. P_{fail}, the x and y dimensions of the problem instance,

and max velocity $v_{\text{max}} \in \{1, 2\}$ are provided as context to the metalevel agent.

The **race track (RT) domain** is a grid world originally described by Barto, Bradtke, and Singh (1995), and used by McMahan, Likhachev, and Gordon (2005) to evaluate their BRTDP algorithm. The state space $(x, y, v_x, v_y) \in S$ is 4 dimensional and consists of 2D position and velocity. Actions are to accelerate in any of the 8 cardinal directions, or to do nothing. Actions fail to have any effect with probability P_{fail}. Colliding with a wall sets the velocity to zero. The initial state is a state on the start line and goal states are the racetrack finish line. The default policy is to proceed at a constant speed of 1 in the direction along the track.

Problem instances for this domain are randomly generated 28×21 racetrack layouts, combined with randomly sampled maximum velocities and action failure probabilities. The maximum allowed velocity of the agent in each axis is $v_{\text{max}} \in \{3, 4\}$ with equal probability, and $P_{\text{fail}} \sim \text{Uniform}(0, 0.3)$. These two values are provided to the metalevel agent as context. Challenges for metareasoning in this domain are the relatively large 4-dimensional state space, and the object-level planner's uninformative heuristic function. Unlike the DST domain, goal states are located some distance from the agent's start state. This can lead to planning algorithms taking a long time to generate an initial solution.

For the experiments we define time in units of object-level planner state visits. For WBRTDP, a state visit consists of evaluating transition probabilities and carrying out a backup at a state, which can be assumed to take constant time if the number of actions and transitions available is similar across all states. Using this instead of elapsed real time aids repeatability by negating external effects, such as varying processor task load impacting processing time. The maximum time for a metareasoning episode is 10K state visits for the DST domain or 100K state visits for the RT domain. This is divided evenly into 20 metalevel timesteps. For each problem instance, the thinking cost per metalevel timestep is sampled from $\text{Uniform}(0.0, 10.0)$. For both domains, illustrations and additional details are given in the supplementary material.

For each method, we evaluate across 8 DQN agents trained from differing random seeds. Agents are trained with 300K steps (RT) or 600K steps (DST) of experience, and their best performing checkpoint (evaluated on a non-overlapping evaluation set) is used in the experiments. Experiments are run using a held-out set of 1000 problems sampled from $p(\mathcal{M})$.

Results

We compare our method to two alternative ways of addressing the lack of solution quality knowledge. These methods use the known-solution-quality cost function from previous works (Sung, Kaelbling, and Lozano-Pérez 2021; Bhatia et al. 2022), but differ in how they estimate the current solution quality. The first estimation method, *MidBound*, uses the midpoint of the BRTDP bounds as an estimate of the expected cost of the solution. This is readily available and cheap to compute, but is likely inaccurate. The second method, *PolicyEval*, evaluates the current object-level policy on the object-level MDP at each metareasoning step. This gives an accurate estimate of the current solution quality, but is expensive to compute. The additional reasoning increases the cost of thinking incurred by the metalevel agent by a constant factor, which we measure empirically as described in the supplementary material.

Additionally, we compare performance with ablations of our method that disable certain components. The *NoFeatures* ablation removes algorithm features from the RL agent state, the *NoContext* ablation similarly removes the problem context, and the *NoTuning* ablation disables hyperparameter tuning so carries out only stopping time optimisation.

We do not compare with the metamyopic agent proposed by Lin et al. (2015) as their setting is online (rather than single-shot) metareasoning. Furthermore their method relies on predicting BRTDP bounds behaviour from the last step's bounds behaviour. In almost all cases in our experiments, it takes more than one metalevel timestep for the planner to generate an initial solution. This would result in the metamyopic agent assuming zero possibility of solution improvement and always executing after a single thinking step.

Figure 3 shows the combined thinking and acting cost for each method in the DST domain. Performance is normalised for each problem instance, where 0.0 is the expected cost of the optimal object-level solution (equivalent to the optimal metalevel cost assuming zero thinking cost) and 1.0 is the expected cost of the default policy. It is not possible for the metalevel agent to achieve a cost lower than the optimal object-level solution, as $\lambda \geq 0$. This is a similar concept to the metareasoning gap presented by Lin et al. (2015).

In the DST domain, our method achieves the lowest cost on average. The NoContext ablation performs the worst of the ablations, as this learning agent does not know the cost of thinking for the current problem. At best it can only optimise its behaviour in expectation for the cost of thinking distribution in the problem distribution. All ablations show a range of deliberation time, demonstrating that they are carrying out metareasoning based on the problem at hand rather than converging on a fixed metalevel episode length that suits the problem distribution. In fact all ablations have a termination time ranging from the earliest timestep to the latest timestep.

The two methods that estimate solution quality, MidBound and PolicyEval, have a much narrower range of deliberation time and incur more cost on average. For MidBound, the agent's estimate of the current solution quality is particularly inaccurate at the start of the episode, where the WBRTDP bounds are very loose. This leads to myopic behaviour and early termination. On the other hand, PolicyEval incurs an increase in thinking cost by having to evaluate its current policy at every metalevel decision step. For this problem setting and object-level implementation, the increase in thinking cost varied from around 50% to 100%. This is too high a cost to be offset by the improved solution quality estimate. Given the high cost of thinking, it converges to always terminating very early in the episode.

For the RT domain (Figure 4), the NoFeatures ablation performs almost identically in cost and metalevel episode length to the full method, although it does have higher standard deviation in cost. This suggests that for this domain the algorithm features are not particularly useful, and good performance can be achieved based on other state features.

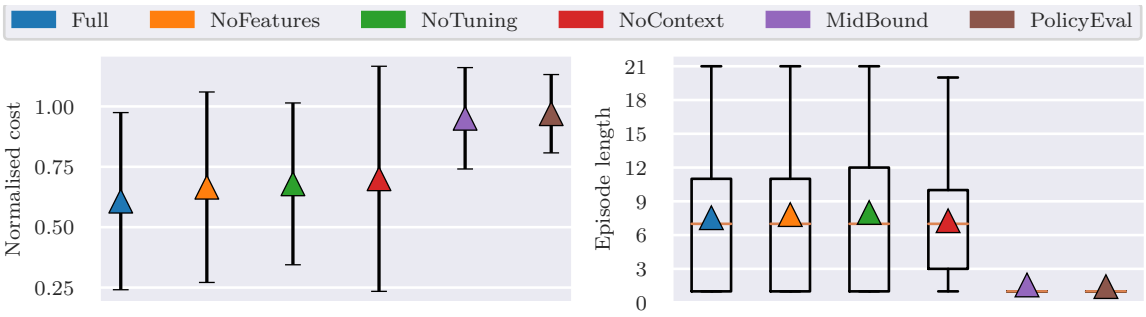

Figure 3: Normalised total thinking and acting cost and metalevel episode length on the held-out problem set in the DST domain. The left hand side plot shows the mean and standard deviation of the incurred cost for each method or ablation, and the right shows the distribution over number of thinking steps before executing. Triangles show mean values, orange lines show medians.

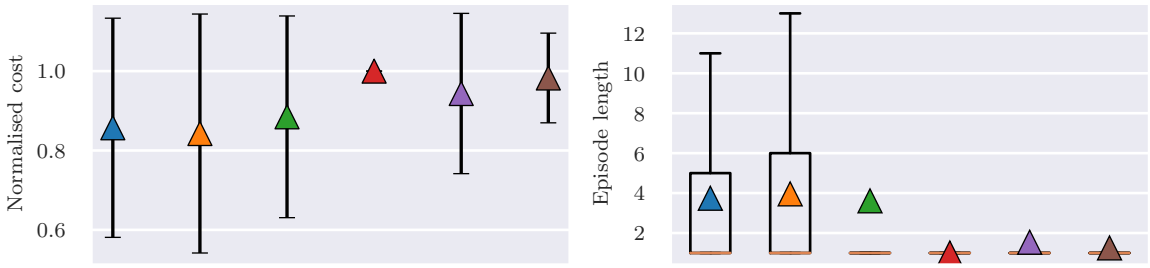

Figure 4: Normalised total thinking and acting cost and metalevel episode length on the held-out problem set in the RT domain.

The mean metalevel episode length for the NoTuning ablation significantly differs from the median and quartiles. This could result from a small proportion of long episodes and a large proportion of immediate execution. This makes sense for an ablation which can only perform the slowest, most optimal planning, and chooses to do this only when the cost of thinking is low. Finally, the MidBound estimation method is more successful in this domain than in the DST domain, but still performs poorly compared to our method.

We analyse the statistical significance of these results using the one-sided Mann-Whitney U test. Our method is significantly better ($p = 0.01$) than all ablations and alternative solution quality estimation methods in both domains, except for the NoFeatures ablation in the RT domain.

In order to demonstrate the metareasoning behaviour of our method, we study metalevel episode length in more detail in the DST domain. For the held-out problem set, Figure 5 shows the relationship between the cost of thinking for an episode $\lambda(\mathcal{M})$ and the timestep at which the metalevel agent chooses the EXEC action. There is a strong negative correlation between the two, indicating that the agent is more likely to terminate early when the cost of thinking is high. The figure also shows the agent's hyperparameter optimisation behaviour. Rather than learning a fixed optimal hyperparameter value for the problem distribution, the agent is clearly adapting its hyperparameter values to the cost of thinking in the current problem. Higher costs of thinking correlate with higher WBRTDP heuristic weights. Higher heuristic weights decrease the time taken for the planner to generate an initial imperfect solution, as illustrated in Figure 2.

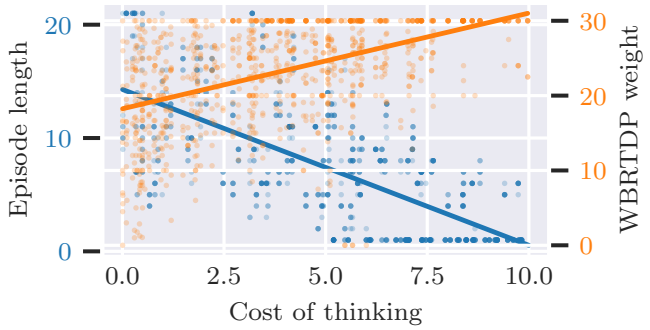

Figure 5: Correlation between the cost of thinking value in an episode and two algorithm behaviour metrics: metalevel episode length and average hyperparameter values chosen by the metalevel agent. Lines are linear regression fits.

Conclusion

This paper has presented a learning-based method to achieve non-myopic metalevel control of probabilistic planning algorithms. Our abstraction of algorithm configuration to algorithm features may lead to a non-Markov state transition function, which could be addressed in future work using a recurrent, LSTM or transformer architecture. We also aim to extend this method to the online planning rather than single-shot planning setting, widening its applicability. Finally, the assumption of a default object-level policy could be lifted by reasoning about both the goal-reaching success probability and the expected cost of the current policy.

Acknowledgements

This work received EPSRC funding via the "From Sensing to Collaboration" programme grant [EP/V000748/1]. Matthew Budd was supported by an Amazon Web Services Lighthouse Scholarship.

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2): 81–138.

Bhatia, A.; Svegliato, J.; Nashed, S. B.; and Zilberstein, S. 2022. Tuning the hyperparameters of anytime planning: A metareasoning approach with deep reinforcement learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 556–564.

Bonet, B.; and Geffner, H. 2012. Action selection for MDPs: Anytime AO* versus UCT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 1749–1755.

Burns, E.; Ruml, W.; and Do, M. B. 2013. Heuristic search when time matters. *Journal of Artificial Intelligence Research*, 47: 697–740.

Callaway, F.; Gul, S.; Krueger, P.; Griffiths, T.; and Lieder, F. 2018. Learning to select computations. In *34th Conference on Uncertainty in Artificial Intelligence (UAI 2018)*, 776–785.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal planning while the clock ticks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, 39–46.

Cox, M. T.; and Raja, A. 2011. *Metareasoning: Thinking about thinking*. MIT Press.

Delalleau, O.; Peter, M.; Alonso, E.; and Logut, A. 2019. Discrete and continuous action representation for practical RL in video games. *arXiv preprint arXiv:1912.11077*.

Falkner, S.; Klein, A.; and Hutter, F. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, 1437–1446. PMLR.

Hansen, E. A.; and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28: 267–297.

Hansen, E. A.; and Zilberstein, S. 2001. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1-2): 139–157.

Hay, N.; Russell, S.; Tolpin, D.; and Shimony, S. E. 2012. Selecting computations: theory and applications. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 346–355.

Lin, C. H.; Kolobov, A.; Kamar, E.; and Horvitz, E. 2015. Metareasoning for Planning Under Uncertainty. In *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*, 1601–1609.

Mausam; and Kolobov, A. 2012. *Planning with Markov decision processes: An AI perspective*. Morgan & Claypool Publishers.

McMahan, H. B.; Likhachev, M.; and Gordon, G. J. 2005. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd International Conference on Machine learning*, 569–576.

Milli, S.; Lieder, F.; and Griffiths, T. 2017. When does bounded-optimal metareasoning favor few cognitive systems? In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533.

Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M.; and Dormann, N. 2021. Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1): 12348–12355.

Russell, S.; and Wefald, E. 1989. On Optimal Game-Tree Search using Rational Meta-Reasoning. In *IJCAI*, 334–340.

Russell, S.; and Wefald, E. 1991. Principles of metareasoning. *Artificial Intelligence*, 49(1-3): 361–395.

Shperberg, S. S.; Coles, A.; Karpas, E.; Shimony, S. E.; and Ruml, W. 2020. Trading plan cost for timeliness in situated temporal planning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*.

Sung, Y.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning when to quit: meta-reasoning for motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 4692–4699. IEEE.

Vamplew, P.; Dazeley, R.; Berry, A.; Issabekov, R.; and Dekker, E. 2011. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning*, 84: 51–80.

Supplementary Materials

Weighted BRTDP (WBRTDP)

Prelims. In heuristic search, a *heuristic function* $h(s)$ guides search by estimating the expected cost of the optimal policy π^* starting from state s. A lower-bound (optimistic) heuristic is admissible if it never overestimates the true cost to reach the goal. An upper-bound (pessimistic) heuristic is admissible if it never underestimates the true cost to reach the goal.

Algorithm overview. WBRTDP is a weighted hyperparameterised version of BRTDP (McMahan, Likhachev, and Gordon 2005). It maintains multiple lower-bound value functions, each of which use different heuristic value initialisations. We define state-action Q-values with respect to a value function v as

$$Q_v(s,a) = c(s,a) + \sum_{s'} T(s,a,s')v(s'). \tag{1}$$

In WBRTDP, $\boldsymbol{v_l}(s)$ is a tuple of N_v lower-bound value functions $\left(v_l^i(s)\right)_{i=0}^{N_v}$, each initialised with a different lower-bound heuristic. Let κ be an index into $\boldsymbol{v_l}(s)$, and let $\boldsymbol{h_l}(s)$ be a N_v-length tuple of lower-bound heuristic functions used to initialise $\boldsymbol{v_l}(s)$. Each lower-bound value function is Bellman updated in parallel during planning using the cost and transition function for the MDP. The action selected for each state during a trial is the action that maximises the lower-bound value function estimate at index κ. κ therefore acts as a hyperparameter which selects which lower-bound heuristic to use during planning, and may be altered during the planning process. The WBRTDP_PLAN method accepts a hyperparameter κ to use for that planning cycle.

$\boldsymbol{h_l}(s)$ is expected to be a tuple of the heuristic function with increasing weights applied, or different heuristic functions with increasing inadmissibility. As we use a tuple of fixed values in our paper, the tuple of heuristic initialisation values are effectively weighted versions of a single fixed-value heuristic function. Such a heuristic function was demonstrated in experiments in the original BRTDP paper (McMahan, Likhachev, and Gordon 2005), and is simpler than the heuristic generation algorithm also described in that paper. We also tested weighted versions of simple heuristic functions, e.g. Manhattan and Euclidean distance, but did not find they performed any better than the fixed-value heuristic in either domain.

Note that only the lower bound heuristics are weighted. The upper bound heuristic remains admissible, and as in BRTDP, the output policy is greedy with respect to the upper bound value function estimate. This means that the upper bound value function remains an upper bound on the true value function, and the weighted heuristics affect only the speed of convergence by biasing search actions during planning. With an admissible upper bound heuristic, the upper

Algorithm 2: WBRTDP

Input: Solution convergence tolerance α, trial termination constant τ, initial state s_0, lower bound heuristic function $\boldsymbol{h_l}(s)$, lower bound heuristic index κ, upper bound heuristic function $h_u(s)$, maximum number of state visits to run $N_{\text{visit}}^{\max}$

Output: Policy $\pi_{s_0}^{v_u}$, greedy with respect to the upper bound value function estimate v_u

procedure WBRTDP_INIT
  $v_u(s) \leftarrow h_u(s)\ \forall s \in S$ ▷ Or lazy initialise on visit
  $\boldsymbol{v_l}(s) \leftarrow \boldsymbol{h_l}(s)\ \forall s \in S$ ▷ Or lazy initialise on visit
  $N_{\text{visit}} \leftarrow 0$
end procedure

procedure WBRTDP_PLAN($\alpha, \tau, s_0, \kappa, N_{\text{visit}}^{\max}$)
  while $(v_u(s_0) - v_l^\kappa(s_o)) > \alpha$ **and** $N_{\text{visit}} < N_{\text{visit}}^{\max}$ **do**
    WBRTDP_TRIAL(s_0)
  end while
  return $\pi_{s_0}^{v_u}$ ▷ Calculate greedy policy wrt v_u
end procedure

procedure WBRTDP_TRIAL(s_0)
  visited $\leftarrow$ (empty stack)
  $s \leftarrow s_0$
  while true **do**
    visited.push(s)
    $v_u(s) \leftarrow \min_a Q_{v_u}(s,a)$
    $a, \boldsymbol{v_l}(s) \leftarrow$ BELLMANTUPLEUPDATE(s)
    $\forall s', b(s') \leftarrow T(s,a,s')(v_u(s') - v_l^\kappa(s'))$
    $B \leftarrow \sum_{s'} b(s')$
    if $B < ((v_u(s_0) - v_l^\kappa(s_0))/\tau)$ **then break**
    end if
    $s \leftarrow$ sample from distribution $b(y)/B$
  end while
  $N_{\text{visit}} \leftarrow N_{\text{visit}} +$ visited.size
  while visited.size > 0 **do**
    $s \leftarrow$ visited.pop()
    $v_u(s) \leftarrow \min_a Q_{v_u}(s,a)$
    $\_, \boldsymbol{v_l}(s) \leftarrow$ BELLMANTUPLEUPDATE(s)
  end while
end procedure

procedure BELLMANTUPLEUPDATE(s)
  $\boldsymbol{Q}_{\text{best}} \leftarrow \left(\min_a Q_{v_l^i}(s,a)\right)_{i=0}^{N_v}$
  $a \leftarrow \arg\max_a Q_{v_l^\kappa}(s,a)$
  return $a, \boldsymbol{Q}_{\text{best}}$
end procedure

bound performance guarantee of BRTDP will therefore be maintained.

If the algorithm is queried for a solution in the middle of running a trial, it returns the current best solution found so far in previous trials.

Algorithm description. The structure of WBRTDP (Algorithm 2) is similar to BRTDP, with the addition of a hyperparameter κ to select the lower bound heuristic to use for the current planning cycle. Some components are separated into subroutines for clarity. The WBRTDP_INIT routine is called at the start of planning, to initialise the value functions and the state visit counter. In practice, the heuristic values are initialised lazily on first visit to a state, rather than all at once at the start of planning.

The WBRTDP_PLAN routine is called to run the planning algorithm. This is expected to be called multiple times by the metalevel agent, each time specifying the cumulative number of state visits to run (line 6). The WBRTDP_PLAN routine carries out trials starting from the root node s_0 (line 8) until the termination condition is met. The trials loop terminates when the difference between the upper and lower bound value functions at the root node is less than a threshold α, or the number of state visits exceeds a maximum $N_{\text{visit}}^{\max}$ (line 7). When the loop terminates, the greedy policy with respect to the upper bound value function is returned (line 10).

Note that the termination condition depends on the current lower bound heuristic index κ. If the solution has converged within α for one lower bound heuristic, it may not have converged for another lower bound heuristic. Planning can be continued using a different lower bound heuristic by calling the plan routine again with a different κ.

The WBRTDP_TRIAL routine carries out a single trial, starting from the root node s_0. The structure is much the same as in BRTDP, with three differences. Firstly, Bellman updates for the lower bound value function (lines 18 and 29) are separated into a subroutine BELLMANTUPLEUPDATE. Secondly, the weighted sampling (line 19) and trial termination steps (line 22) are dependent on the lower bound heuristic index κ. Thirdly, we track the number of trial state visits in line 25.

The BELLMANTUPLEUPDATE subroutine is called to update the lower bound value function at a state s and to return the action greedy with respect to the κth lower bound value function. As the transition and cost function components are the same for each Q-value update in the tuple (line 33), we can compute the Q-value updates for all lower bound value functions in a single pass over actions and outcome combinations. This leads to a negligible computational overhead compared to updating a single lower bound value function.

The overall computational overhead of WBRTDP compared to BRTDP is negligible as long as evaluating the tuple of lower-bound heuristic functions is not signficantly more expensive than evaluating a single heuristic function. This applies when using weighted multiples of a heuristic function.

Algorithm features. Algorithm features for WBRTDP are the value of the upper and lower bound value functions at the root node s_0, the number of trials completed, the total number of state visits during all trials, and the number of state visits during the last trial.

Hyperparameters. For the experiments we provide uniform uninformative (fixed-value) heuristics for the upper and lower bound value functions. The lower bound heuristic function therefore provides fixed values rather than a weighted version of a heuristic function: $\boldsymbol{h_l}(s) = (0.0, 10.0, 20.0, 30.0)\ \forall s$. This results in a metalevel action space size of 5 (4 hyperparameter options and 1 EXEC action). The heuristic initialisation for the upper bound function is fixed at $h_u(s) = 100.0\ \forall s$, which is admissible for all problems.

Deep Q Network

The DQN (Mnih et al. 2015) algorithm implementation is from Stable Baselines 3, version 1.8.0 (Raffin et al. 2021). Training is vectorised with 10 parallel environments. Default parameters are used, other than setting the number of gradient steps to match the number of transitions collected across all parallel environment instances. As a basic hyperparameter evaluation, experiments were also run using the default parameters of the DQN implementation from the work of Bhatia et al. (2022). This yielded slightly decreased performance so these hyperparameters were not used for further experiments.

As is common with deep RL, we normalise algorithm features and the context vector to the range 0 to 1. Value functions are normalised using the maximum possible cost-to-goal from s_0 for the problem distribution $p(\mathcal{M})$. The number of trial state visits feature is normalised by the number of trial state visits corresponding to the maximum number of timesteps per episode.

Alternative solution quality etimation methods

The *PolicyEval* and *MidBound* methods estimate the value of the current solution and use this estimate in the reward function defined by (Bhatia et al. 2022). This consists of rewarding the RL algorithm with the improvement in solution cost minus the cost of thinking. The computational overhead of PolicyEval is evaluated by recording the runtimes of planning and policy evaluation for each metalevel timestep. Using the ratio of these times, policy evaluation cost can be expressed in cost-of-thinking units. The total deliberation cost is the combination of planning cost and policy evaluation cost. The *PolicyEval* RL agent is provided with this total cost during training and evaluation. The same method is used to adjust the incurred cost of thinking during evaluation on the held-out problem set.

Experiments/Reproducibility

The held-out problem set used to evaluate algorithms' performance is reproducibly seeded. Each held-out environment is seeded with the same value across all algorithms. This ensures identical object-level policy execution and planner dynamics across each evaluation.

Experiment-running machines were Amazon Web Services *G4dn.4xlarge* instances with 16 vCPUs, 64 GiB RAM, and 1x 16GiB NVIDIA T4 GPU. The operating system was Ubuntu 20.04. The object-level MDP simulation and WBRTDP were implemented in C++.

Deep Sea Treasure domain

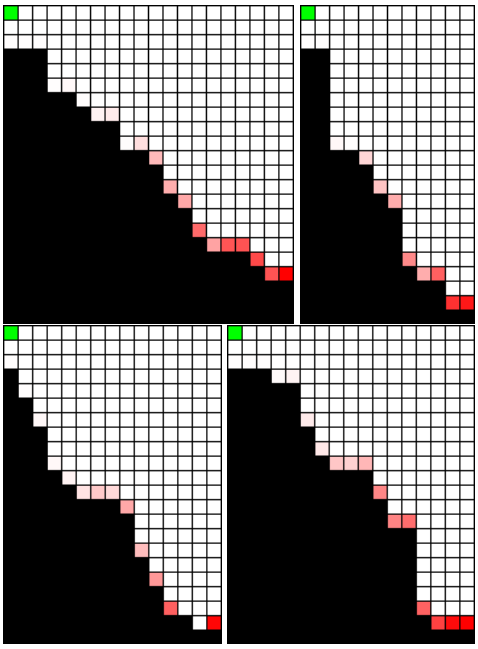

Figure 1: Four randomly generated deep sea treasure problems. The agent start location is shown in green, and increasingly dark red colours show increasingly valuable treasures.

This section describes the generation of problems $\mathcal{M} \sim p(\mathcal{M})$ for the deep sea treasure domain.

Figure 1 shows four randomly generated deep sea treasure layouts. Grid dimensions are sampled with $\text{dim}_x \sim \text{Uniform}\{10, 20\}$ and $\text{dim}_y \sim \text{Uniform}\{18, 25\}$. The ocean floor is generated by sampling a depth $\sim \text{Uniform}\{3, \text{dim}_y\}$ for each for each x up to dim_x, and then sorting these points in ascending order to give monotonically increasing depth with x. A treasure is placed at each point just above the ocean floor, with a probability $P_{\text{treasure}} = 0.9$ for each point. Treasure values are sampled using a Poisson distribution with depth-dependent mean. The mean value for a depth is proportional to a polynomial $(\text{depth})^2$. The maximum treasure value that can be sampled is 99. Problem instances are generated by combining a deep sea treasure layout with a uniformly sampled cost of thinking $\lambda \sim \text{Uniform}(0.0, 10.0)$, a per-axis speed limit $v_{\text{max}} \sim \text{Uniform}\{1, 2\}$, and an action failure probability $P_{\text{fail}} \sim \text{Uniform}(0, 0.3)$.

The initial state is always the top left corner of the grid. The default policy action is always to descend towards the ocean floor, and to proceed to the right if the agent is at the ocean floor but no treasure is present at that location.

Race track domain

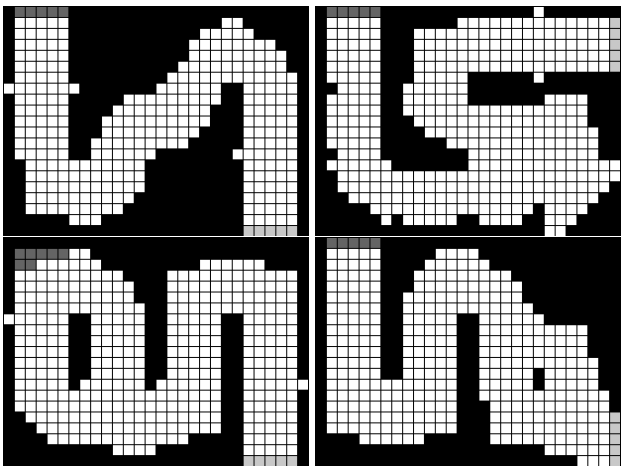

Figure 2: Four randomly generated race track problems. The start line is shown in dark grey, and the finish line is shown in light grey.

This section describes the generation of problems $\mathcal{M} \sim p(\mathcal{M})$ for the race track domain.

Figure 2 shows four randomly generated race track layouts. Race track layouts are generated by sampling 2 random paths in a 4×3 grid of nodes, each with a minimum path length of 5 and a maximum path length of 9. These paths are combined to form a high-level layout which may include forks and loops. The high-level layout is upscaled into a 28×21 cell race track using a set of 7×7 cell templates which map to possible combinations of paths leading into/out of a node. If the resulting race track is too short (minimum path length from start to goal < 50), then the paths are resampled. A race track being too short could result from the combination of the two sampled node paths being too short, even if each individual path matches the path length requirements. The start line is placed at the start of the path in the top left corner, and the finish line is placed at the end of the path. Problems instances are generated by combining a race track layout with a uniformly sampled cost of thinking $\lambda \sim \text{Uniform}(0.0, 10.0)$, a per-axis speed limit $v_{\text{max}} \sim \text{Uniform}\{3, 4\}$, and an action failure probability $P_{\text{fail}} \sim \text{Uniform}(0, 0.3)$.

The centre point of the start line is chosen as the deterministic initial state for the agent. The default policy action is to proceed at a speed of 1 along the track towards the goal.