Reducing the Planning Horizon through Reinforcement Learning

Logan Dunbar (✉)[1][0000−0001−7486−7684], Benjamin Rosman[2][0000−0002−0284−4114], Anthony G. Cohn[1,3][0000−0002−7652−8907], and Matteo Leonetti[4][0000−0002−3831−2400]

[1] School of Computing, University of Leeds, Leeds, UK
{sclmd,a.g.cohn}@leeds.ac.uk
[2] University of the Witwatersrand, South Africa
benjamin.rosman1@wits.ac.za
[3] Tongji University, China; Alan Turing Institute, UK; Qingdao University of Science and Technology, China; Shandong University, China
[4] Department of Informatics, King's College London, London, UK
matteo.leonetti@kcl.ac.uk

Abstract. Planning is a computationally expensive process, which can limit the reactivity of autonomous agents. Planning problems are usually solved in isolation, independently of similar, previously solved problems. The depth of search that a planner requires to find a solution, known as the planning horizon, is a critical factor when integrating planners into reactive agents. We consider the case of an agent repeatedly carrying out a task from different initial states. We propose a combination of classical planning and model-free reinforcement learning to reduce the planning horizon over time. Control is smoothly transferred from the planner to the model-free policy as the agent compiles the planner's policy into a value function. Local exploration of the model-free policy allows the agent to adapt to the environment and eventually overcome model inaccuracies. We evaluate the efficacy of our framework on symbolic PDDL domains and a stochastic grid world environment and show that we are able to significantly reduce the planning horizon while improving upon model inaccuracies.

Keywords: Planning · Planning Horizon · Reinforcement Learning

1 Introduction

Planning is a notoriously complex problem, with propositional planning shown to be PSPACE-complete [5]. The planning horizon is the maximum depth that a planner must search before finding a solution, and the number of paths through the search graph grows exponentially with a deepening planning horizon. This exponential growth makes graphs with an even moderate branching factor slow to traverse. Furthermore, frequent replanning when solving such problems limits the reactivity of agents, making it challenging to incorporate planning into real-time applications.

Machine learning is increasingly used to take advantage of previously solved planning instances to guide the search and plan faster [12]. Much of the work has focused on using plans from simpler problems to learn generalised heuristics for larger problems [30, 31, 20]; however, even with improved heuristics the planning horizon is left untouched, leaving the problem prohibitively expensive in general. Methods that learn a policy from a set of plans [1, 23, 3] have been studied to compile the deliberative behaviour of the planner into a reactive policy. Such methods can be highly effective, but rely on large training sets of plans, and are not suitable for online learning.

Our work is inspired by psychological experiments, which have shown that humans have two distinct decision making systems, commonly known as the habit/stimulus-response system and the goal-directed system [6, 14, 25]. Early work thought these two systems were in competition for resources, but Gershman et al. [8] showed that these systems exhibit a more cooperative architecture, where a model-based system was used to train a model-free system that acted on the environment. With this inspiration, we model the goal-directed system as a classical planner with access to a model, and the habit-based system as a model-free reinforcement learning (RL) algorithm.

In this paper, we propose the Plan Compilation Framework (PCF), a framework that uses model-free RL to compile the plans of a classical planner into a reactive policy. We target an agent that repeatedly executes a task, learning to plan less and less as the agent accumulates experience. Our method can be used online as the agent faces the task, does not require a training set to be initialised, and is agnostic to the type of planner used. Eventually, the behaviour becomes completely reactive and model-free, with the added benefit that it can leverage reinforcement learning to further optimize the policy beyond what can be planned on the model.

2 Background

In this section we introduce the notation and background on planning and RL that we will use throughout the paper.

2.1 Planning

We consider discrete planning problems, consisting of a tuple $\langle \mathcal{S}, \mathcal{A}, \tilde{\mathcal{T}}, \tilde{\mathcal{C}}, s_0, \mathcal{S}_\mathcal{G} \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\tilde{\mathcal{T}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \{0, 1\}$ is a deterministic transition function that models the environment, $\tilde{\mathcal{C}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the model's cost function, s_0 is an initial state and $\mathcal{S}_\mathcal{G} \subseteq \mathcal{S}$ is a set of goal states. Planning produces a plan $P(s_0) \doteq [a_0, a_1, \dots a_n]$, a sequence of actions that transforms s_0 into a goal state $s_{n+1} \in \mathcal{S}_\mathcal{G}$ when applied sequentially.

Planners can be seen as executing a simple abstract algorithm: from the initial state, choose a next state s_i according to some strategy, check if $s_i \in \mathcal{S}_\mathcal{G}$ and exit with the plan if true, otherwise expand the state into its successors using the model, and loop until a goal is found. Our work only requires changing the goal test logic of existing planners, which affords extensive integration opportunities.

2.2 Reinforcement Learning

A reinforcement learning problem is modelled as a Markov Decision Process (MDP), a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the environment's transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [r_{min}, r_{max}]$ is a reward function and γ is a discount factor.

A policy $\pi(a \mid s)$ is a probability distribution over actions conditioned on the current state $s \in \mathcal{S}$. The return $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ is the cumulative discounted reward obtained from time step t. The expected return when starting in state s, taking action a and following policy π is known as the state-action value function:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a \right]. \tag{1}$$

The value function is bounded by $[q_{min}, q_{max}]$ where $q_{min} = r_{min}/(1-\gamma)$ and $q_{max} = r_{max}/(1-\gamma)$. The goal of RL is to find an optimal policy π^* that maximises $Q_\pi(s, a), \forall s, a$. Two central methods for learning a value function are Monte-Carlo methods and temporal difference learning [27].

Monte-Carlo Methods Constant-α Monte Carlo [27] uses the actual return G_t as the target when updating the value function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[G_t - Q(s_t, a_t) \right]. \tag{2}$$

This produces an unbiased but high variance estimate of the value function, which we leverage to allow our agent to initially favour actions specified by the planner, as discussed in Sec 4.1.

Temporal Difference Learning Q-learning [28] is a common single-step algorithm, whose update target is $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$, which replaces the return in Eq. 2. Methods can also look further than one step ahead, with the n-step return defined as

$$G_{t:t+n} \doteq r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_{t+n-1}(s_{t+n}), \tag{3}$$

where $V_{t+n-1}(s_{t+n})$ is the value of state s_{t+n} and can be approximated by $\max_a Q(s_{t+n}, a)$. This allows the agent to blend actual returns generated by environmental interactions with bootstrapping estimates further into the future.

2.3 Distances Between Distributions

In this work we use the concept of a distance between distributions to categorise when the policy for a particular state has stabilised. A common way to quantify the difference between two probability distributions $P(x)$ and $Q(x)$ is the relative entropy, also known as the Kullback-Leibler Divergence (KLD) [17]:

$$\mathcal{D}_{KL}(P \,||\, Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right). \tag{4}$$

A related measure, which, unlike the KLD, is a distance metric, is known as the Jenson-Shannon Divergence (JSD): [19]

$$\mathcal{D}_{JS}\left(P \,||\, Q\right) = \frac{\mathcal{D}_{KL}\left(P \,||\, M\right)}{2} + \frac{\mathcal{D}_{KL}\left(Q \,||\, M\right)}{2} \tag{5}$$

where $M{=}\frac{1}{2}\left(P + Q\right)$. This measure allows us to quantify the change in our policy when we update the value function for a state, as detailed in Sec 4.2.

3 Related Work

Gershman et al. [8] showed that a simple implementation of the DYNA architecture [26] was able to replicate their psychological findings, lending credence to the idea of a cooperative model-based and model-free system. In DYNA, a model-free algorithm chooses actions, and a model-based algorithm trains the model-free values. While a model can be specified as prior knowledge, generally the agent learns a model of the environment through interactions. Even if a model is specified, DYNA would be unable to initially prefer the actions from this model, as our work does. This negates some of the benefits of pre-specified models, namely fast and reliable goal achievement. DYNA also requires that the model be able to generate accurate rewards, as they are incorporated into the value function from which the model-free algorithm chooses its actions. We require only that the planner generate a plan to achieve the goal, which allows us to use classical planning.

Other techniques for reducing the planning time include Lifelong Planning A* (LPA*) and D* Lite [15], which are incremental heuristic search methods. LPA* repeatedly calculates the shortest distances from the start state to the goal state as the edge costs of the graph change. D* Lite considers the opposing problem and searches from the goal to the current state. This formulation allows the start state to change without needing to recompute the entire search graph. In D* Lite the planning horizon is effectively shortened by the agent moving towards the goal during plan enactment, but for any particular state the horizon does not change. Both LPA* and D* Lite require access to a predecessor model which is often harder to specify than a successor model. We could leverage these planners in our work, but we find the dependence on a predecessor model too limiting.

Our concept of a *learnt* state (Sec. 4.2) is similar to the *known* state from Explicit Explore or Exploit (E^3) [13]. In E^3 a state becomes known when it has been visited and the actions tried sufficiently often to produce an estimate of the transitions and pay-offs with high probability. They use the concept of the *known* state to partition the MDP into known and unknown regions, learning to exploit the current known states or to explore the unknown states. They need to explicitly learn the model of the transition and reward probabilities, which can be slow and require visiting many states that are irrelevant to the current goal. We similarly partition the MDP into *learnt* and *unlearnt* states, but we leverage the prior knowledge of the planner to initialise the *learnt* states, thereby

reducing the environmental interactions while making sure to achieve the goal, as well as providing a more relevant exploration frontier. We also reduce our dependence on the planner over time, moving to efficient model-free learning, while E^3 continues to maintain and update its model estimates, which could become costly in the long-term.

The work by Grounds and Kudenko [9] and Grzes and Kudenko [10] use planning to shape [22, 29] the rewards received by learning algorithms. Grounds and Kudenko [9] learn a low-level Q-learning behaviour for each STRIPS operator by using the STRIPS planner to generate plans from which they derive a shaping value. Grzes and Kudenko [10] use a similar technique, but instead use the generated high-level plan to shape the reward of a single Q-learner. They use the step number of the plan to provide a potential field for shaping. This guides the Q-learner to choose actions that would lead the agent along the planner's path. Although shaping provides suggestions to the agent, it is unable to enforce which actions are chosen. This means the agent can very quickly start behaving like pure Q-learning if the shaping and environmental reward are mismatched. Our goal in this work is to utilise the information contained in the planner as effectively as possibly, preventing Q-learning-like exploration which can leave the agent goal-impoverished while trying to explore the entire state space.

The work most similar to ours is Learning Real-Time-A* (LRTA*) [16], which is generalised by Real-Time Dynamic Programming (RTDP) [2]. LRTA* aims to bring planning to real-time applications by performing local search within a limited horizon and stores the results in an evaluation function which is updated over successive trials. RTDP was subsequently developed as a form of asynchronous dynamic programming [4] that uses Bellman backups to obtain the values of states. RTDP operates under the same horizon and time constraints as LRTA*, making it also suitable for real-time applications. Both LRTA* and RTDP require a perfect model to backup values accurately through the state space. Where our work differs is that LRTA* and RTDP will eventually converge to the optimal values for the possible policies afforded by their models. Through repeated trials they will converge to the model-optimal values, but the search horizon is never reduced, nor is actual environmental information incorporated into the search. Our framework instead leverages the information contained in the planner's model during the initial phase of operation, and gradually cedes control to the model-free learning algorithm. This means the planner will at some point never be called, and the model-free learner will be completely responsible for choosing actions, incorporating real environmental feedback and adapting as necessary.

4 Plan Compilation Framework

We now propose the Plan Compilation Framework with the Plan Compiler (PC) agent, the goal of which is to leverage reinforcement learning to reduce the planning time of a classical planner when repeatedly solving a problem from different initial states, eventually becoming purely reactive. The classical planner

produces plans that take the agent to the goal, while model-free RL learns to compile the planner's behaviour into a value function online. Once the model-free policy of a state has stabilised through successive value function updates, we consider it *learnt* and the planner is no longer required for that state, ceding control to the model-free RL algorithm. We then augment the goal set of the planner with the *learnt* states, generating plans to the goal, or to a *learnt* state, where the model-free RL algorithm can take over and react accordingly. This process reduces the planning horizon through repeated interactions, as well as leverages the ability of model-free RL to explore and improve upon the performance of the planner's model.

The PC agent's algorithm is shown in Alg. 1. It is a classical RL agent computing actions based on the current state and learning from the observed next state and reward. The key elements are described in the following sections.

4.1 Compiling the Planner's Policy

The planner induces a policy π_{pln} over the state space by virtue of being able to generate a plan $P(s)$ and returning the first action from that plan. We use model-free RL to compile this policy into a value function Q by learning to ascribe higher value to the actions chosen by π_{pln} in the initial phases of operation. Q-value initialisation, as shown by Matignon et al. [21], plays a crucial role in the initial behaviour of the agent. We want to replicate the behaviour of the planner, not explore every action. We therefore pessimistically initialise our Q-values, $q_{init} \leftarrow q_{min} - \delta$, driving the agent to prefer states and actions it has previously visited. This requires knowing the lower bound of return we expect to receive while following the planner, which is always possible when r_{min} is known.

The biased nature of bootstrapping TD updates learn to favour actions *not* in π_{pln} in the early phases of operation. The total change to the state-action value during update is

$$\delta Q(s,a) \leftarrow \alpha[r' + \gamma \max_{a'} Q(s',a') - Q(s,a)], \tag{6}$$

which, when all Q-values are uniformly initialised to q_{init}, is dominated by r' due to $\gamma \max_{a'} Q(s',a') - Q(s,a) = \gamma q_{init} - q_{init} \approx 0$ on the first update. Any negative reward therefore lowers an action's value, pushing the policy away from π_{pln}. We want, initially, the actions suggested by the planner, regardless of reward sign, to be preferred over all other actions. To do this, we leverage the unbiased Monte Carlo update (Sec. 2.2), which uses the return as the target rather than a biased bootstrapped value. This, coupled with the pessimistic initialisation, preserves the desired action preferences when learning to replicate π_{pln}, as seen in Alg. 1, Ln. 25, 28.

4.2 Relinquishing Control to the Learner

Once the agent has learnt to replicate the planner's policy in a state, it can query the model-free learner for the action rather than the computationally expensive

Algorithm 1: Plan Compiler Agent

Input:
ϵ, ϵ_{exp}, α, α_l, $\tau_{\mathcal{D}}$, τ_l, ζ
$\forall s, a, \quad Q(s,a) \leftarrow q_{min} - \delta$
$\forall s, a, \quad Q_{exp}(s,a) \leftarrow q_{max} + \delta$
$e \leftarrow 0$

```
1  for each episode do
2      Initialise s
3      repeat
4          a ← act (s)                                          #Ln. 10
5          Take action a, observe r′, s′
6          learn (s, a, r′, s′)                                 #Ln. 21
7          s ← s′
8      until terminal (s)
9  end
10 def act (s):
11     if e ≤ 0 then                                            #Not Exploring
12         if !learnt (s) then
13             return action from planner
14         else if rand() < ε_exp then                          #Start Exploring
15             v ← argmax_a Q(s, a)
16             e ← ξ|v|
17         else
18             return ε-greedy action from Q
19     return ε-greedy action from Q_exp                        #Exploring
20 end
21 def learn (s, a, r′, s′):
22     if learnt (s) and learnt (s′) then
23         bootstrap (Q, (s, a, r′, s′))
24     else
25         traj.append(s, a, r′, s′)
26                                                              } Update Q
27     if terminal (s′) then
28         monte_carlo (Q, traj)
29     else if learnt (s′) then
30         n_step (Q, traj)
31     bootstrap (Q_exp, (s, a, r′, s′))                        } Update Q_exp
32     if e > 0 then
33         e ← e − |r′|                                         #Eq. 11
34 end
```

planner. We define a state as being *learnt* when its policy stabilises through successive Q-value updates. We compute the JSD distance (Eq. 5) between the state's policy before π_{pre} and after π_{post} performing the Q-value update. We

introduce a function $l : \mathcal{S} \rightarrow [0, 1]$ that quantifies how stable the policy is for a particular state, akin to anomaly detection [7]. We track the stability estimate with an exponential recency-weighted average

$$l(s) \leftarrow l(s) + \alpha_l \left[u_l - l(s) \right] \tag{7}$$

where α_l is the stability learning rate, and u_l is the update target. The target is binary, and computes whether the update caused a policy change above a threshold $\tau_{\mathcal{D}}$[5]:

$$u_l = \begin{cases} 1, \text{ if } \mathcal{D}_{JS} \left(\pi_{pre} \,||\, \pi_{post} \right) < \tau_{\mathcal{D}} \\ 0, \text{ otherwise.} \end{cases} \tag{8}$$

When $l(s)$ is above a threshold $\tau_l \in (0, 1)$, meaning the policy for state s has stabilised, we consider s learnt:

$$learnt(s) = \begin{cases} True, \text{ if } l(s) > \tau_l \\ False, \text{ otherwise.} \end{cases} \tag{9}$$

Once a state is considered learnt, and the agent next encounters it, the learner chooses an action from Q, without the need to invoke the planner. We prevent the state from reverting to unlearnt by setting $u_l(s) \leftarrow 1$ for all subsequent stability updates. The learner can now exploit the planner's policy or can choose to explore alternate actions to potentially improve upon the planner's policy.

Once states are considered learnt, the negative effect of the bootstrapping bias is reduced, and the agent can update the value function with TD learning, which allows online updates during the episode. The update procedure is detailed in Alg. 1, Ln. 22-30. The control flow of action selection is shown in Fig. 1 and detailed in Alg. 1, Ln. 10-20.

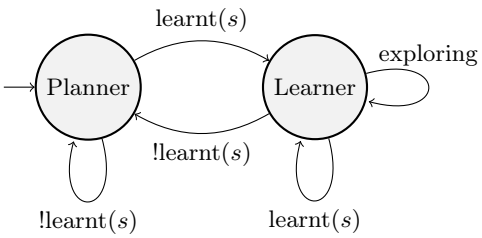


Fig. 1: Control flow of Plan Compiler action selection: the planner has initial control while states are !learnt(s), once a state becomes learnt(s), control passes to the learner. The learner maintains control while successive states are learnt(s), transferring control back to the planner if a state is !learnt(s). The learner also maintains control while exploring, either keeping control or giving control back to the planner once the quota is depleted, depending on the state's learnt status.

[5] This can be prespecified or adapted online as per the work of De Klerk et al. [7].

4.3 Exploration

We facilitate local exploration around the paths provided by the planner with the aim to improve upon its performance without incurring drastic exploration costs. The pessimistically initialised Q-values continuously drive the agent back onto the paths preferred by the planner, preventing exploration.

To enable controlled exploration while deviating from the planner's behaviour we take two steps: (1) we introduce a second value function Q_{exp}, which is optimistically initialised; (2) we establish an exploration budget based on state values and interrupt exploration when the budget is spent. Q_{exp} is updated online with bootstrapping as seen in Alg. 1, Ln. 31.

When in a learnt state, with some small probability ϵ_{exp} the agent begins exploring, choosing actions from the optimistic value function Q_{exp}. The state's current pessimistic value $V(s) \leftarrow \max_a Q(s,a)$ is an estimate of how much return the agent expects to receive from that point onwards, which we use to bound the amount of exploration the agent is allowed. Using this value, we calculate an exploration quota e_t, which is proportional to $V(s)$

$$e_t \leftarrow \xi |V(s_t)|, \tag{10}$$

where $\xi \geq 0$ determines how far, in terms of value, we are willing to explore away from the planner's paths. The agent then exclusively chooses actions from Q_{exp} which takes the agent into unexplored regions of the state space. The exploration behaviour is shown in Alg. 1, Ln. 14-16, 19.

While exploring, we reduce the exploration quota by the reward received

$$e_{t+1} \leftarrow e_t - |r_{t+1}|, \tag{11}$$

shown in Alg. 1, Ln. 33. When the quota drops to or below zero, exploration ends, at which point the agent will be in an unlearnt state, choosing an action from the planner, or in a learnt state, choosing an action from Q. The quota has the effect of preventing the agent from straying too far from the learnt regions, thereby maintaining reactivity, preventing excessive exploration, and making sure the agent achieves the goal regularly and reliably. The two parameters ϵ_{exp} and ξ are tunable to specify how often and how far the agent is allowed to travel from the paths generated by the planner.

4.4 Reducing the Planning Horizon

Once a state has become learnt, we can reduce the planning horizon by augmenting the goal set of the planner:

$$\mathcal{S}'_{\mathcal{G}} \leftarrow \mathcal{S}_{\mathcal{G}} \cup \{s \mid learnt(s)\}. \tag{12}$$

The planner is then able to plan to states for which the learner has a stable policy, from which it can react accordingly. Through repeated interactions with the environment, more states become learnt, shortening the planning horizon until the planner is rarely invoked.

Initially, we may pay a path-length cost due to planning to *learnt* states as opposed to goal states, or due to pernicious initial state distributions. Over time, as the agent explores locally around the planner's paths, the model-free algorithm learns the true state-action values and undoes the reward penalty. Additionally, decoupling the model-free learner from the planner, in the way PCF does, allows the use of deterministic planners in stochastic domains as an alternative to computationally expensive sampling-based stochastic planners. Learning overcomes the inaccurate model used for planning, as shown in the DARLING system [18].

5 Experiments

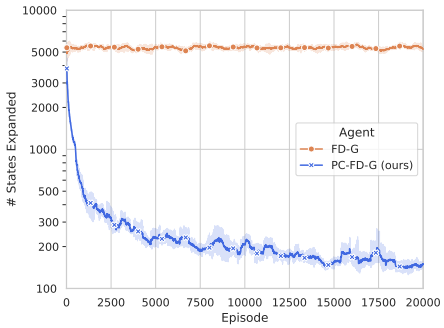
We designed two experiments to evaluate the properties of our method. In the first experiment we show the trade off between the reduction in the number of states expanded by the planner and its cost in terms of sub-optimality of the reward. We use three PDDL domains from the International Planning Competition (IPC)[6] and FastDownward [11] as the planner. In the second experiment we show how, through reinforcement learning, our system can improve over planning with an inaccurate model. For this experiment we cannot use IPC domains, since these do not incorporate any modeling error. Therefore, we use a grid world whose parameters we can control so as to diverge from the model.[7]

5.1 PDDL Domains

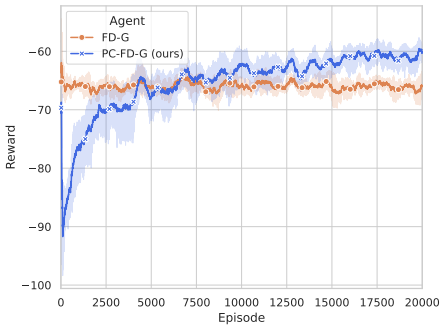
Depot is a logistics-like domain where crates are trucked between depots and stacked in specific orders using hoists, with resource constraints on the hoists and trucks. 15-Puzzle is a classic planning domain with 15 unique tiles in a 4x4 grid that can exchange places with a single blank tile, the objective being to arrange the tiles in ascending order. 15-Blocks is a domain with 15 blocks that can be stacked atop a table or one another, and the goal is to create a particular pattern of stacked blocks. For all three domains, the agent receives -1 reward for every action taken in the environment.

We use two settings of FastDownward, called FD-G and FD-GFF, using the Context Enhanced Additive heuristic and the FastForward heuristic respectively, both of which are non-admissible and produce sub-optimal plans. We chose the faster setting per environment. We show that we can integrate our Plan Compiler agent with these planners, which we label PC-FD-G and PC-FD-GFF. Our Q and Q_{exp} are tables using hashed PDDL states and actions to store the values. The results are averaged over 5 runs of 20k episodes with random initial states. The PC parameters used were: ϵ=0.1, α=1, α_l=1, $\tau_{\mathcal{D}}$=0.01, τ_l=0.9, ξ=0, ϵ_{exp}=0.

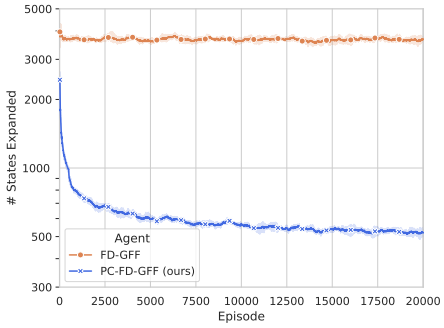
[6] Implemented by PDDLGym [24]
[7] Code at https://github.com/logan-dunbar/plan_compilation_framework



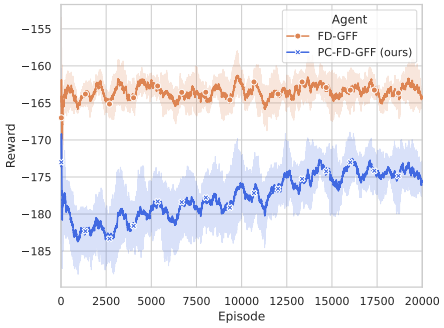
(a) Depot: expanded states per episode



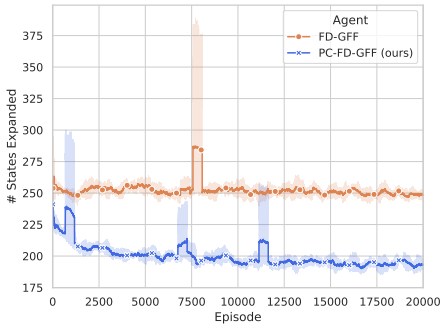
(b) Depot: reward per episode



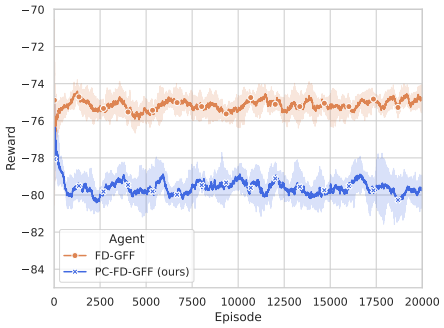
(c) 15-Puzzle: expanded states per episode



(d) 15-Puzzle: reward per episode



(e) 15-Blocks: expanded states per episode



(f) 15-Blocks: reward per episode

Fig. 2: PDDL domains results

Results The results for the PDDL experiments are shown in Fig. 2. The number of states expanded per episode is shown for each domain in Figs. 2a,c,e. The graphs clearly show a large reduction in the number of states expanded during each episode, noting the logarithmic scale for Depot and 15-Puzzle. For Depot, the number of states expanded is reduced by an order of magnitude after approx-

imately 1k episodes, with 25x fewer expansions after 5k episodes. The results are similar for 15-Puzzle, with 5x fewer expansions after 2.5k episodes. This justifies the potentially exponential reduction in computation achieved when reducing the planning horizon, as alluded to in the introduction. For 15-Blocks, the number of state expansions is reduced by 20% after 5k episodes. The FastForward heuristic performs extremely well in this domain, averaging only 250 states expanded for solutions with average path length of 75, which is already close to optimal.

Reward per episode is shown for each domain in Figs. 2b,d,f. In Depot, the reward drops sharply in the very early episodes. However, the model-free RL algorithm quickly begins exploring locally around the paths generated by the planner, bringing the reward obtained to parity after 7.5k episodes, and begins to outperform the planner from that point onward. This means that after 20k episodes the system is both outperforming the planner in terms of reward and has reduced the computational burden of the planner by over an order of magnitude. For 15-Puzzle, the characteristic drop-off in reward is present in the initial episodes, after which there is a clear trend of improvement. In this domain, the reward does not reach parity with the planner after 20k episodes, meaning that the advantage in terms of planning time has a long-lasting cost in terms of reward. However, the trend suggests that longer runtime will once again bring the performance in line with the planner, and the near order of magnitude reduction in number of states expanded might be considered a worthwhile trade-off for a roughly 6% loss of reward performance. If optimality can be foregone, and a slight reduction in reward tolerated, a large reduction in computational requirements can be gained. In 15-Blocks, the reward received initially drops sharply, but then stabilises 7% worse than just using the planner. The huge space and branching factor seems to prevent the model-free algorithm from finding improved plans, but this could also be due to FastForward providing excellent paths very close to optimal, coupled with our constant cost of ϵ-greedy exploration.

5.2 Grid world

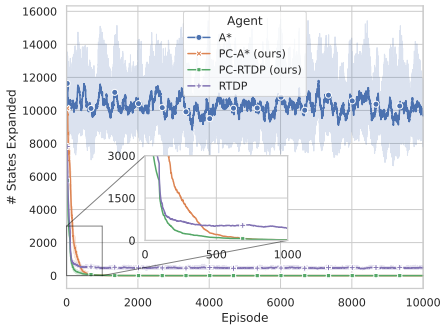
The grid world is a stochastic 50x50 maze-like world with walls, quicksand, random initialisation and a fixed goal location. The grid for each run is randomly generated to contain 20% walls, and 25% of the remaining free space is allocated to be quicksand. Even rows and columns are twice as likely to receive quicksand. This has the effect of creating maze-like paths which the agent can learn to traverse. We operate under the assumption that the goal is reachable, and therefore ensure that at least 50% of the non-wall locations can reach the goal, otherwise we regenerate the grid. In each episode a random initial state is chosen from the set of reachable states. Moving into a wall receives -5 reward, stepping into quicksand receives -100 reward, and every other action receives -1 reward. The agent can move in the 4 cardinal directions, and choosing to move in a direction succeeds 80% of the time, with 20% chance to move in either of the neighbouring directions. Q and Q_{exp} are tables of hashed states and actions storing the values.

The model used for planning is deterministic, and therefore incorrect. Furthermore, the model has no cost, such that the planner aims to compute the shortest path, which may not be optimal in terms of reward. We test our framework against vanilla Q-learning (Q in Fig. 3), A* (A*) and RTDP (RTDP), and we use both A* (PC-A*) and RTDP (PC-RTDP) as our Plan Compiler planners. The results are averaged over 5 runs of 10k episodes. The PC parameters used were: ϵ=0.1, α=0.1, α_l=0.1, $\tau_{\mathcal{D}}$=0.01, τ_l=0.9, ξ=0.5, and ϵ_{exp} is linearly reduced from 0.03 to 0 in 8k episodes.

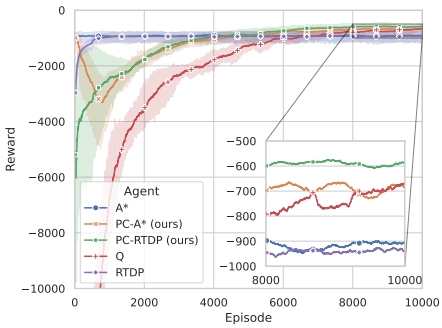
The second part of this experiment showcases the effect of the exploration quota ξ. Higher values of ξ should result in more exploration in the early phases of operation with a commensurate loss in reward, while a lower ξ should remain more faithful to the plans generated by the planner, achieving the goal more regularly in the early phase but possibly losing out on finding better paths in the long run. We use A* as our planner, vary the quota for six different settings ξ=$\{0.05, 0.1, 0.15, 0.3, 0.5, 1\}$, fix the remaining parameters to: ϵ=0.1, α=0.1, α_l=0.1, $\tau_{\mathcal{D}}$=0.01, τ_l=0.9, and ϵ_{exp} is linearly reduced from 0.03 to 0 in 8k episodes.

Results The results for the gridworld domain are shown in Fig. 3. Fig. 3a shows a large reduction in the number of states expanded, where both PC-A* and PC-RTDP have essentially reduced the planning horizon to 0 after 750 episodes. This is possible due to the small state space allowing the agent to visit and learn every state. RTDP also drops off rapidly, but as the planner has no way of reducing the planning horizon to 0, it maintains a constant planning cost. This could become prohibitive in larger environments with an open-loop planning cycle such as implemented here, whereas our method will continue to reduce its computational requirements over time. A* maintains a fairly large computational burden as it has no way of incorporating the previous solutions.

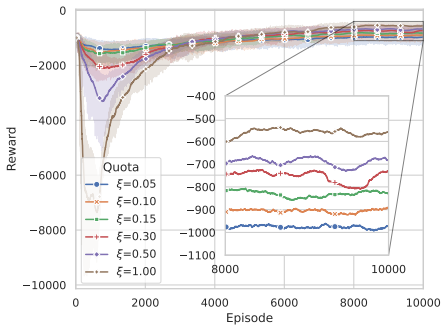
The reward per episode is shown in Fig. 3b. Q-learning is known to be sample inefficient and this can be seen in the asymptotic reward in the first few episodes. This makes Q-learning impractical in real-world settings, because it would be too costly for an agent or robot to explore as much as Q-learning requires. However, the large amount of exploration does mean that Q-learning can find excellent paths and we see this with the agent averaging -700 reward after 10k episodes. A* and RTDP have no knowledge of the true transition function, nor of the real reward function, nor any way to incorporate experience. This means they will only be able to achieve what their models afford them. RTDP initially has behaviour similar to that of Q-learning, as it computes its value function using dynamic programming updates. It manages this quickly in this small world and then reaches a steady state of reward of approximately -950. A* is much the same, with the random initialisation and grids providing some small fluctuations. PC-A* starts at the same performance level as A*, drops slightly until some states become learnt and it can begin exploring. This can be seen in the drop off between episodes 0-1k. The exploration begins to pay dividends at this point and the agent improves until reaching parity with Q-learning, at -700 reward.



(a) Expanded states per episode



(b) Reward per episode



(c) Reward per episode for varying quota values

Fig. 3: GridWorld results

This is remarkable considering the amount of exploration Q-learning required versus that of PC-A*. This confirms that even suboptimal plans can be excellent exploratory guides. PC-RTDP starts with RTDP's characteristic Q-learning like exploratory behaviour, with large variability in the early phases of operation. But it too puts that exploration to good use and quickly outperforms all other agents with an average of -600 reward after 10k episodes. This shows that we can leverage the strengths of other approaches and even improve upon them.

The final experiment in Fig. 3c shows the effect the quota parameter ξ has on exploration. A lower ξ results in less exploration, and this can clearly be seen in the first 2k episodes. ξ=1 is the most exploratory, receiving large negative reward as it traverses the state space, while ξ=0.05 is the least exploratory, remaining truer to the planner's paths. The rest are properly ordered between the two extreme values of ξ. This reverses as the exploratory behaviour finds better paths than what the planner could suggest, resulting in ξ=1 achieving the highest long term reward, ξ=0.05 the lowest, and the rest ordered in between. This demonstrates the flexibility a designer has when using the Plan Compilation

Framework. Depending on the situation, one can vary ξ to either engage in exploratory behaviour or to be faithful to the guiding planner and prefer reducing the computational requirements of the planner.

6 Conclusions

In this paper we have designed and implemented the novel Plan Compilation Framework that combines classical planning with model-free reinforcement learning in such a way as to compile the planner's policy into a model-free RL value function, dramatically reducing the planning horizon over time and improving the long term computational efficiency of the system. We show by experiment in PDDL domains that we are able to reduce the number of states expanded by the FastDownward planner, and we show in a stochastic grid world that we are able to reduce the planning horizon of A* and RTDP planners whilst also improving upon an imperfect model. In future work we will look to introduce the power of function approximation to be able to handle infinite state spaces and robotic applications.

References

1. Argall, B.D., Chernova, S., Veloso, M., Browning, B.: A survey of robot learning from demonstration. Robotics and Autonomous Systems **57**(5) (2009)
2. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence **72**(1) (1995)
3. Bejjani, W., Dogar, M.R., Leonetti, M.: Learning physics-based manipulation in clutter: Combining image-based generalization and look-ahead planning. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (2019)
4. Bertsekas, D.P.: Distributed asynchronous computation of fixed points. Mathematical Programming **27**(1) (1983)
5. Bylander, T.: Complexity results for planning. In: 12th International Joint Conference on Artificial Intelligence (1991)
6. Daw, N.D., Niv, Y., Dayan, P.: Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control. Nature Neuroscience **8** (2005)
7. De Klerk, M., Venter, P.W., Hoffman, P.A.: Parameter analysis of the Jensen-Shannon divergence for shot boundary detection in streaming media applications. SAIEE Africa Research Journal **109**(3) (2018)
8. Gershman, S.J., Markman, A.B., Otto, A.R.: Retrospective revaluation in sequential decision making: A tale of two systems. Journal of Experimental Psychology: General **143**(1) (2014)
9. Grounds, M., Kudenko, D.: Combining reinforcement learning with symbolic planning. In: Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning. Springer (2005)
10. Grzes, M., Kudenko, D.: Plan-based reward shaping for reinforcement learning. In: 4th International IEEE Conference Intelligent Systems. vol. 2. IEEE (2008)
11. Helmert, M.: The fast downward planning system. Journal of Artificial Intelligence Research **26** (2006)

12. Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., Borrajo, D.: A review of machine learning for automated planning. The Knowledge Engineering Review **27**(4) (2012)
13. Kearns, M., Singh, S.: Near-optimal reinforcement learning in polynomial time. Machine Learning **49**(2-3) (2002)
14. Keramati, M., Dezfouli, A., Piray, P.: Speed/accuracy trade-off between the habitual and the goal-directed processes. PLoS Computational Biology **7**(5) (2011)
15. Koenig, S., Likhachev, M.: Fast replanning for navigation in unknown terrain. IEEE Transactions on Robotics **21**(3) (2005)
16. Korf, R.E.: Real-time heuristic search. Artificial Intelligence **42**(2) (1990)
17. Kullback, S., Leibler, R.A.: On information and sufficiency. The Annals of Mathematical Statistics **22**(1) (1951)
18. Leonetti, M., Iocchi, L., Stone, P.: A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. Artificial Intelligence **241** (2016)
19. Lin, J.: Divergence measures based on the Shannon entropy. IEEE Transactions on Information theory **37**(1) (1991)
20. Marom, O., Rosman, B.: Utilising uncertainty for efficient learning of likely-admissible heuristics. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 30 (2020)
21. Matignon, L., Laurent, G.J., Le Fort-Piat, N.: Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. In: International Conference on Artificial Neural Networks (2006)
22. Ng, A.Y., Harada, D., Russell, S.: Policy invariance under reward transformations: Theory and application to reward shaping. In: Proceedings of the Sixteenth International Conference on Machine Learning (1999)
23. Pérez-Higueras, N., Caballero, F., Merino, L.: Learning robot navigation behaviors by demonstration using a RRT* planner. In: International Conference on Social Robotics. Springer (2016)
24. Silver, T., Chitnis, R.: PDDLGym: Gym environments from PDDL problems. In: International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop (2020)
25. Solway, A., Botvinick, M.M.: Goal-directed decision making as probabilistic inference: a computational framework and potential neural correlates. Psychological Review **119**(1) (2012)
26. Sutton, R.S.: Dyna, an integrated architecture for learning, planning, and reacting. ACM SIGART Bulletin **2**(4) (1991)
27. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
28. Watkins, C.J.C.H.: Learning from delayed rewards. Ph.D. thesis, King's College, Cambridge (1989)
29. Wiewiora, E., Cottrell, G.W., Elkan, C.: Principled methods for advising reinforcement learning agents. In: Proceedings of the 20th International Conference on Machine Learning (2003)
30. Yoon, S.W., Fern, A., Givan, R.: Learning heuristic functions from relaxed plans. In: Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling. vol. 2 (2006)
31. Yoon, S., Fern, A., Givan, R.: Learning control knowledge for forward search planning. Journal of Machine Learning Research **9**(4) (2008)