The Intersection of Planning and Learning through Cost-to-go Approximations, Imitation and Symbolic Regression

by

Stefan O'Toole

ORCID: 0000-0002-7352-4409

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

November 2022

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems

Doctor of Philosophy

by Stefan O'Toole

ORCID: 0000-0002-7352-4409

This thesis explores the intersection between planning and learning methods for autonomous sequential decision-making. Planning is a model-based approach to autonomous sequential decision-making where action policies are derived automatically through a model of an environment. Alternatively, learning methods learn action policies through interaction with an environment. The planning and learning approaches can be likened to current theories of human cognition which propose a fast and associative system works in conjunction with a slow and deliberative one. From this observation previous work has conjectured that in order to create intelligent systems that are more general and robust than existing ones, a combination of planning and learning methods may be required.

Two common high-level approaches for combining planning and learning are to use learning to help guide the search effort of planners and to use planners to teach learning algorithms. This thesis examines these two high-level approaches through the topics of cost-to-go approximations, symbolic regression and imitation. We propose and study a number of new algorithms which provide new insights into methods that combine planning and learning, namely, we introduce methods for learning value and policy functions from lookeaheads; learning from single demonstrations produced by planners; and learning heuristics for planning algorithms.

Declaration of Authorship

I, Stefan O'Toole, declare that this thesis titled, 'The Intersection of Planning and Learning through Cost-to-go Approximations, Imitation and Symbolic Regression' and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the degree of Doctor of Philosophy, except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the 100,000 words in length, exclusive of tables, maps, bibliographies and appendicese.

Signed: [illegible]

Date: 28/11/2022

Preface

This thesis is comprised of original works completed in collaboration with my PhD supervisors Nir Lipovetzky, Miquel Ramirez and Adrian R. Pearce, where I, Stefan O'Toole, was the principle contributor and author. These works were completed solely during my PhD candidature and have not been submitted for any other qualifications.

Below is a list of each of the works comprised within the thesis. For these works I was responsible for greater than 50% of the work including being responsible for algorithm design, running experiments, analysing results, and producing the manuscripts. My supervisors provided ideas, insights, feedback, and assisted in producing the manuscripts.

- Chapter 3 contains materials from the following paper: O'Toole, S., Ramirez, M., Lipovetzky, N., Pearce, A.R. (2019, July). "Width-based lookaheads augmented with base policies for stochastic shortest paths". In the proceedings of the 11th Workshop on Heuristics and Domain Independent Planning at the International Conference on Automated Planning and Scheduling, pg 37-45.

- Chapter 4 contains materials from the following paper: O'Toole, S., Lipovetzky, N., Ramirez, M., Pearce, A.R. (2021, December). "Width-based Lookaheads with Learnt Base Policies and Heuristics Over the Atari-2600 Benchmark" published in the proceedings of the Advances in Neural Information Processing Systems, vol 34, pg 26536–26547.

- Chapter 6 contains materials from the following pre-print paper: O'Toole, S., Lipovetzky, N., Ramirez, M., Pearce, A.R. (2022). "Imitation Learning via Symbolic Pre-Imaging".

- Chapter 7 contains materials from the following;
  - published extended abstract: O'Toole, S., Ramirez, M., Lipovetzky, N., Pearce, A.R. (2022, July). "Sampling from Pre-Images to Learn Heuristic Functions for Classical Planning". In the proceedings of the Fifteenth International Symposium on Combinatorial Search;
  - pre-print paper: O'Toole, S., Ramirez, M., Lipovetzky, N., Pearce, A.R. (2022). "Sampling from Pre-Images to Learn Heuristic Functions for Classical Planning".

This research was financially supported by the following sources:

- Australian Government Research Training Program Scholarship
- The Defence Science Institute, an initiative of the State Government of Victoria, Grant
- Google PhD Travel Grant
- The International Conference on Automated Planning and Scheduling (ICAPS) Travel Grant

Acknowledgements

I have been incredibly fortunate throughout my PhD journey to be surrounded by so many remarkable and supportive people.

First, I would like to acknowledge my amazing supervisors Dr. Nir Lipovetzky, Dr. Miquel Ramirez and Prof. Adrian Pearce. To my two principal supervisors Nir and Miquel, I have been honoured to share this journey with you. I couldn't have wished for more committed, supportive, and knowledgeable scholars to help navigate me through my candidature. I truly appreciate the countless hours you spent workshopping ideas with me, editing my writing, and helping me to understand different research concepts. I have learnt so many valuable things from you both and you have helped me become a better researcher, coder, and writer. Nir, you struck the perfect balance between challenging me when required and always making me feel at ease. Thank you for all that you have done from helping me refine a presentation the night before a conference to introducing me to the best food spots in Melbourne. Miquel, your expertise in such a diverse set of fields truly astounds me and I feel privileged to have been able to work with you. Thank you for always going above and beyond whether it was by helping me through a code review or promptly replying to one of my questions at 10 pm on a Saturday night. I count myself very lucky to have had two principal supervisors who showed such dedication in helping me through my PhD. Adrian, thank you for supporting me throughout my PhD, I really appreciate the time you invested into this journey. To my Academic Chair Prof. Ben Rubinstein, thank you for your time and efforts running my progress review meetings, I very much appreciate it.

I have been blessed to have had the unwavering support of my family. I am thankful to my partner Linna for her support through out my PhD. It meant everything to be able to share the highs and lows of the experience with you. Thank you for making our 2 years of lockdowns together so enjoyable and for being my number one supporter. I truly would not have been able to complete this work without you and I am lucky to have you as my partner in life. To my parents, Peter and Trish, thank you for providing me the very best opportunities to excel and for always believing in me. To my sister Nadia, thank you for always supporting me.

Lastly, I was extremely fortunate to have met such an amazing group of peers throughout my time at the University of Melbourne. In particular, one of the highlights of my journey was forming friendships with so many of the incredible people from within the agent lab group; Prashan, Steven, David, Anubhav, Xin, Lyndon, Daniel, Fatma, Ruihan, Chenyuan, Guang, Abeer, Michelle, Ronal, and Tim, thank you.

Contents

List of Figures

List of Tables

Abbreviations

ALE	**A**rcade **L**earning **E**nvironment
ASDM	**A**utonmous **S**equential **D**ecision **M**aking
BC	**B**ehavioural **C**loning
BFS	**B**readth-**F**irst **S**earch
BL	**B**ackwards **L**earning
BLP	**B**ackwards **L**earning using **P**olytopes
BLPP	**B**ackwards **L**earning with **P**iecewise-defined **P**olicies
CPL	**C**ritical **P**ath **L**earning
CTP	**C**anadian **T**raveller **P**roblem
DFS	**D**epth-**F**irst **S**earch
DCS	**D**eep **S**kill **C**haining
FF	**F**ast-**F**orward
GAIL	**G**enerative **A**dversarial **I**mitation **L**earning
GBFS	**G**reedy **B**est-**F**irst **S**earch
HGN	**H**yper**g**raphs **N**etworks
IBL	**I**terative **B**ackwards **L**earning
IPC	**I**nternational **P**lanning **C**ompetition
IPPC	**I**nternational **P**robabilistic **P**lanning **C**ompetition
IRL	**I**nverse **R**einforcement **L**earning
MCTS	**M**onte **C**arlo **T**ree **S**earch
MDP	**M**arkov **D**ecision **P**rocess
MPC	**M**odel **P**redictive **C**ontrol
MROS	**M**obile-**R**obot-**O**bstacles-**S**tay
N-CPL	**N**ovelty guided **C**ritical **P**ath **L**earning
N-RSL	**N**ovelty guided **R**egression based **S**upervised **L**earning

NN	**N**eural **N**etwork
PDDL	**P**lanning **D**omain **D**efinition **L**anguage
PPO	**P**roximal **P**olicy **O**ptimisation
SE	**S**tochastic **E**numeration
SMRF	**S**parseness of **M**eaningful **R**eward **F**eedback
RDDL	**R**elational **D**ynamic influence **D**iagram **L**anguage
RIW	**R**ollout-**IW**
RL	**R**einforcement **L**earning
RTDP	**R**eal-**T**ime **D**ynamic **P**rogramming
RNN	**R**ecurrent **N**eural **N**etwork
RSL	**R**egression based **S**upervised **L**earning
SL	**S**upervised **L**earning
SSP	**S**tochastic **S**hortest **P**ath
TD	**T**emporal **D**ifference
TSL	**T**eacher based **S**upervised **L**earning
UCT	**U**pper **C**onfidence bounds applied to **T**rees

Chapter 1

Introduction

This thesis is focused on the intersection of planning and learning methods for Autonomous Sequential Decision Making (ASDM) problems. The ASDM problem can be described through the agent-environment interaction, illustrated in Figure 1.1. An agent is an entity that can observe the state of its environment and interact with it through selecting actions which yield resulting rewards or costs from the environment. Planning and learning methods are two high-level approaches for solving ASDM problems. The goal of both planning and learning methods is to create an action policy that maximises the expected rewards received from an environment. Learning and planning are distinguished by the method in which they create an action policy for the agent. Learning approaches learn an action policy through agent interaction with the environment (Sutton and Barto 2018). Conversely, Planning approaches are model-based methods where the action policy is not learned through interaction but instead derived automatically through a model of the actions, senors, and goals of the agent and environment (Geffner and Bonet 2013).

The planning and learning approaches to ASDM problems can be likened to current theories of human cognition which propose that a fast and associative system (System 1) works in conjunction with a slow and deliberative one (System 2). As described by Geffner (2018) there are parallels between model-free based approaches like Reinforcement Learning (RL) with System 1 and a model-based approach like Automated Planning with System 2. Following from these observations is the idea that in order to create ASDM agents that are more general and robust, a combination of the System 1 and System 2 like approaches may be required. In this

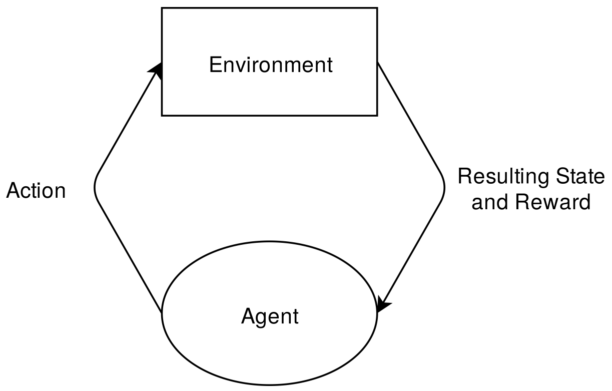


FIGURE 1.1: The agent-environment interface.

thesis we explore the interaction between System 1 and System 2 approaches by exploring the interface between planning and learning through proposing a number of new algorithms.

The AlphaGo (Silver et al. 2016), and AlphaZero (Silver et al. 2018) family of algorithms are particularly famous examples of successfully combining planning and learning approaches. The AlphaGo algorithm was able to defeat the world champion at the game of Go, a feat long considered a grand challenge in the field of Artificial Intelligence research. At the core of the family of algorithms are a couple of key interactions between the planning and learning systems. First a policy function is learnt to guide a planner and a learnt value function is used to evaluate states at the limits of the planner's search tree. Second, the value and policy functions are learnt from observations of the planner playing against itself with previous iterations of the learnt value and policy functions. The success of AlphaGo and AlphaZero suggests that using and learning policy and value functions in this way to guide a planner's exploitation and exploration of the search space is useful.

In many planning and learning methods (Guo et al. 2014; Anthony, Tian, and Barber 2017; Sun et al. 2018; Ferber, Helmert, and Hoffmann 2020) it is common that the planning system, that is slow and deliberative, is interfaced as a teacher to train a learning algorithm, just as the AlphaGo and AlphaZero algorithms do. In this setup the learning algorithm aims to imitate the policy that results from calling the planning algorithm. As discussed next this thesis explores the idea of using a planner as a teacher for a learner as well as the idea of using learning to guide planning algorithms.

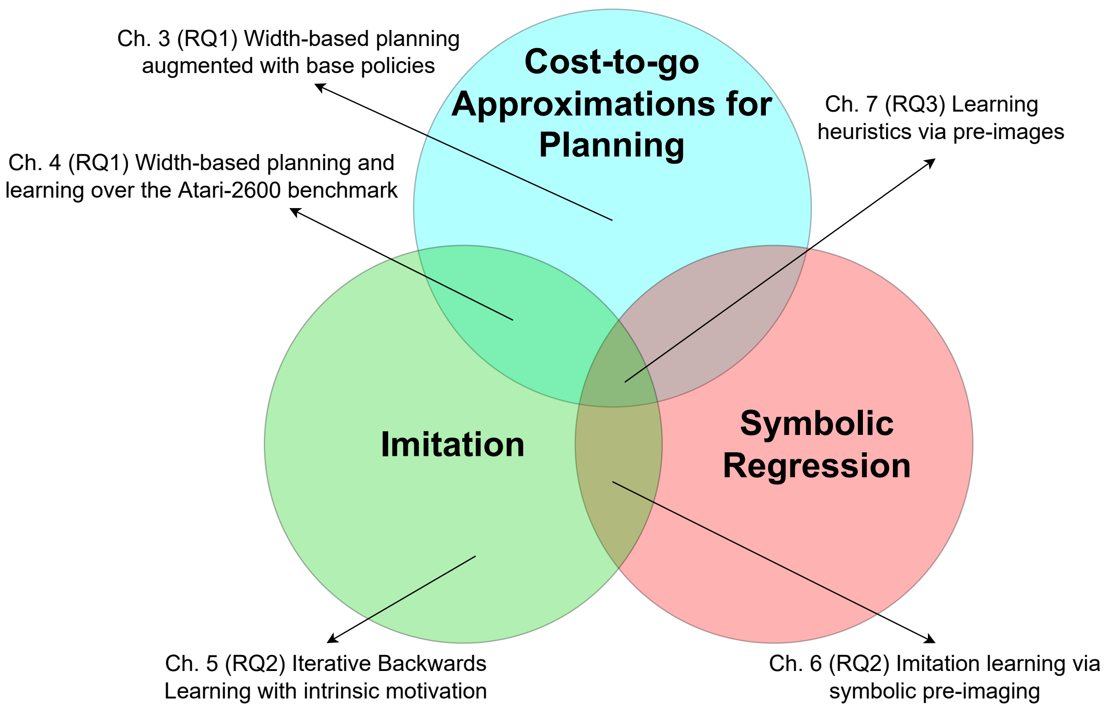

Ch. 3 (RQ1) Width-based planning augmented with base policies
Cost-to-go Approximations for Planning
Ch. 7 (RQ3) Learning heuristics via pre-images
Ch. 4 (RQ1) Width-based planning and learning over the Atari-2600 benchmark
Imitation
Symbolic Regression
Ch. 5 (RQ2) Iterative Backwards Learning with intrinsic motivation
Ch. 6 (RQ2) Imitation learning via symbolic pre-imaging

FIGURE 1.2: Overview of the research areas explored in each Chapter that underpin the research contributions.

1.1 Research questions and contributions

There are many different ways that planning and learning methods for ASDM can interact with one another. This thesis focuses on two high-level ideas for combining planning and learning. The first of these ideas is to use learning to help guide and trade-off the exploration versus exploitation of a planner, and the second is using planners as teachers for learning algorithms. We explore these high-level ideas through the topics of cost-to-go approximations, symbolic regression, and imitation learning. Figure 1.2 illustrates the topics explored in each Chapter which address the research questions introduced below.

Our first research question investigates exploration versus exploitation for black-box simulator-based planning and learning methods,

> **RQ1** How can planning and learning interact to trade-off exploration and exploitation for model-free simulator-based problems?

We investigate this research question in Chapters 3 and 4. Chapter 3 proposes a new model-free simulator-based planning algorithm RIW-λ for Stochastic Shortest Paths (SSPs) based upon the state-of-the-art model-free simulator-based planner Rollout-IW (RIW). Chapter 3 shows the benefit of adding cost-to-go approximations to RIW and suggests that given tight simulator budgets learning cost-to-go approximations could be advantageous in the exploration versus exploitation trade-off. Chapter 4 follows up from Chapter 3 by introducing a new planning and learning algorithm, Novelty based Critical Path Learning (N-CPL), that *learns* cost-to-go approximations and policies that can guide the search of RIW. The novelty mechanisms of the RIW planner drives the exploration of N-CPL while the learnt policies and cost-go-approximations that guide the planner exploit knowledge gained from previous episodes executed by the N-CPL agent. We show that N-CPL's trade-off of exploration versus exploitation pays-off as N-CPL outperforms the previous best model-free simulator-based planning and learning agents on the Atari-2600 games.

As previously discussed a clear intersection between planning and learning is using planners as teachers for learning systems. There is vast literature on imitation learning algorithms, however the algorithms often require complete demonstration trajectories of states, or state-action pairs that are valid for the given environment (Abbeel and Ng 2004; Ho and Ermon 2016; Torabi, Warnell, and Stone 2018b). There are many environments with underlying dynamics that are too complex to plan over however through relaxing the environment's dynamics we can create relaxed solution trajectories. Our next research question tackles how symbolic regression with a full, relaxed or partial demonstration can be exploited to learn effective action policies that generalise over the state-space of an environment.

> **RQ2** To what extent can symbolic regression given a full, relaxed or partial demonstration trajectory assist learning?

We consider **RQ2** in Chapters 5 and 6. Chapter 5 investigates and proposes improvements to learning policies from states that are progressively further away from a goal by regressing through a given demonstration. Following from the open questions and conclusions of Chapter 5, Chapter 6 proposes a new symbolic regression based learning algorithm which is shown to outperform previous methods given full, relaxed and partial demonstrations trajectories.

Finally, we explore how to autonomously discover actions over which to perform regression as opposed to using a given demonstration. We also investigate how symbolic regression can be used by a learning algorithm to learn cost-to-go approximations for use by a forward search planner.

RQ3 To what extent can learning with symbolic regression produce useful cost-to-go approximations?

We study **RQ3** in Chapter 7, where we introduce a new method for learning cost-to-go approximations using symbolic regression named Regression based Supervised Learning (RSL). We measure the usefulness of the learnt approximations by RSL through forward planning and show that RSL learns more useful cost-to-go approximations than previous learning-based methods with two-orders of magnitude less compute.

1.2 Thesis outline

The thesis will be set out as follows,

- Chapter 2 will provide the necessary background of the concepts and related works which underpin each of the research questions we address.
- Chapters 3 and 4 address **RQ1**,
  - Chapter 3 proposes an improvement to a state-of-the-art planning algorithm through using cost-to-go approximations,
  - Chapter 4 extends the improvements laid out in Chapter 3 through learning cost-to-go approximations and base policies.
- Chapters 5 and 6 address **RQ2**,
  - Chapter 5 explores a learning from demonstration algorithm which uses regression and proposes improvements,
  - Chapter 6 addresses some of the issues discovered in Chapter 5 through introducing a new symbolic regression algorithm for learning from demonstration.
- Chapters 7 addresses **RQ3** through presenting and exploring a method which uses symbolic regression to learn cost-to-go approximations.
- Chapter 8 concludes the thesis by summarising its key contributions, and suggesting ideas for future work.

Chapter 2

Background

In this chapter we provide a broad overview of the concepts and related works which are central to the thesis. A discussion of how previous works relate to the contents of each individual Chapter is not discussed and is instead included in each relevant Chapter's related work sections. This chapter starts by defining how we model the ASDM problem and the ASDM benchmark environments we use. Additionally, the Chapter covers some standard definitions that are used throughout the thesis. Finally, we discuss methods used to solve for ASDM problems that include Planning, Reinforcement Learning, Imitation Learning, and methods that combine Planning and Learning.

2.1 Markov Decision Processes

We model the ASDM problems as Markov Decision Processes (MDPs). MDPs model problems as having fully observable states and allow for stochastic actions. We formalise MDPs, as described by Geffner and Bonet (2013),

Definition 2.1. A MDP is the tuple $M = (\mathcal{S}, s_0, A, T, c)$, containing,

- a state-space $\mathcal{S} \subseteq \mathbb{R}^d$,
- an initial state $s_0 \in \mathcal{S}$,
- sets of *discrete* or *continuous* applicable actions A, such that $A(s)$ is a set of actions applicable in $s \in \mathcal{S}$,

- a set of distributions T, such that $T(s,a,s')$ gives the probability of the transition from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ given action $a \in A(s)$,
- a cost function c such that, $c(s,a)$ returns the cost for performing action $a \in A(s)$ from state $s \in \mathcal{S}$. Note that it is common for RL literature to use rewards instead of costs. Rewards can easily be mapped to costs using the relationship $c(s,a) = -R(s,a)$, where R is the reward function.

An action policy for an MDP, π, maps a state $s \in \mathcal{S}$ and an action $a \in A(s)$ into the probability of the policy taking action a when in s. That is, $\pi : s, a \rightarrow [0,1]$ such that for any $s \in \mathcal{S}$, $\sum_{a \in A(s)} \pi(a,s) = 1$. The expected value function of an MDP describes the expected costs from a state s, from following a given policy π, and is defined as,

$$V^{\pi}(s) = \sum_{a \in A(s)} \pi(s,a) \left(c(s,a) + \sum_{s' \in \mathcal{S}} T(s,a,s') V^{\pi}(s') \right) \tag{2.1}$$

The optimal policy $\pi*$ for a problem will produce an expected value function equal to the optimal value function V^* as described through the Bellman optimality equation (Bellman 1957),

$$V^*(s) = \min_{a \in A(s)} \left[c(s,a) + \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right] \tag{2.2}$$

2.1.1 Finite-horizon MDPs

In this thesis we consider a special case of MDPs, finite-horizon MDPs, where accumulated costs need to be minimised over a given number of stages $k = 1, \ldots, H$, starting at an initial state, s_0. Terminal states in finite-horizon MDPs are absorbing states. That is, if s is a terminal state and we are at time step k, every action a will map (s, k) into $(s, k+1)$ and will be cost-free i.e. $c(s,a) = 0$. Our goal is to produce a policy such that it minimises the expected accumulated cost received for an episode of the MDP,

$$\text{argmin}_{\pi} E \left\{ \sum_{k=0}^{H-1} \sum_{a \in A(s)} \pi(s_k, a) c(s_k, a) \right\} \tag{2.3}$$

where the expectation is over $s_{k+1} \sim T(s_k, a, s_{k+1})$.

2.1.2 Goal MDPs: stochastic shortest path problems

We also consider another special case of MDPs, Goal MDPs which are used to describe stochastic shortest path problems. Following the definition by Geffner and Bonet (Geffner and Bonet 2013),

Definition 2.2. A goal MDP is the tuple $M^G = (\mathcal{S}, s_0, \mathcal{S}_G, A, T, c)$, where $\mathcal{S}, s_0, A, T, c$ are as defined in Definition 2.1 except that $c(s, a)$ returns only positive costs for applying any action and $\mathcal{S}_G \subseteq \mathcal{S}$ is a set of goal states which are considered as terminal states of M^G.

The expected value and optimal value function definitions remain as defined in Equations 2.1 and 2.2, except that for states $s \in S_G$, $V^\pi = V^*(s) = 0$.

2.2 Planning models and languages

Planning methods derive action policies for MDPs through a given model of the action, transition and cost functions of an environment. The planning model can be given explicitly through a symbolic description or implicitly through a simulator of the environment.

2.2.1 Classical planning model

Classical planning is concerned with Goal MDPs with deterministic actions, that is $T(s, a, s')$ can only equal 1 for one state s' and 0 for any other state s'', where $s, s', s'' \in \mathcal{S}$, $s' \neq s''$ and $a \in A(s)$. For classical planning problems there are a number of languages that explicitly express their models in a compact form. In this thesis we use the classical planning language of the STRIPS formulation (Fikes and Nilsson 1971).

Definition 2.3. STRIPS defines a planning problem as the tuple $\Pi = \langle F, O, I, G \rangle$, where F is a set of atoms, O is a set of actions, $I \subseteq F$ is the initial state and $G \subseteq F$ is the goal set.

Each action $a \in O$ is represented in STRIPS by the tuple $\langle Add(a), Del(a), Pre(a) \rangle$ which are each a set of atoms over F. $Add(a)$ and $Del(a)$ describe the atoms that are added and removed from the state respectively, and $Pre(a)$ describes the atoms that must be true in a state in order to apply action a. The STRIPS problem Π implicitly represents in a compact form a deterministic transition system (Geffner and Bonet 2013).

Definition 2.4. The classical planning state-transition model for progression is $\mathcal{S}(\Pi) = \langle S, s_0, S_G, A, f, c\rangle$, where $S \subseteq 2^F$, s_0 is the initial state I, S_G is the set of goal states described as the set $\{s \mid s \supseteq G, s \in S\}$, the actions $a \in A(s)$ are the actions in O that are applicable in s, that is $Pre(a) \subseteq s$, f is the transition function where for action a and state s, the resulting state is $s' = f(a, s) = (s \setminus Del(a)) \cup Add(a)$, and finally $c(a, s)$ is the cost of selecting the transition out of s via action a.

A solution for a classical planning problem is a sequence of actions $a_0,\ldots,a_n$ that select transitions connecting the initial state s_0 to a state within the goal set S_G. The progression state-transition model can be used to find a solution through a forward search for a goal state from s_0.

Planners can also search backwards from the goal through the regression state-transition model (Geffner and Bonet 2013).

Definition 2.5. The regression state-transition model is $R(\Pi) = \langle S, s_0, S_G, A, f, c\rangle$, where s_0 is the partially assigned state G, the goal set S_G is the set $\{s \mid s \subseteq I, s \in S\}$, $A(s)$ are the actions in O that are relevant and consistent for the partial state s, that is $Add(a) \cap s \neq \emptyset$ and $Del(a) \cap s = \emptyset$, and the state transition function to pre-image state s is $f(a, s) = (s \setminus Add(a)) \cup Pre(a)$.

A key difference between the regression and progression state-transition models is that the regression model searches over partial truth assignments, which represent sets of states as opposed to complete truth-assignments. For the progression state-transition model every atom not in a state is false, while for the regression state-transition model the atoms not in a state are simply undefined. The regression model is required to search over pre-images $x \subseteq F$ as the initial pre-image x_0 coincides with the set of goal states from the progression state-transition model S_G. Regression operators also compute *weaker preconditions* so the only atoms being asserted in a pre-image x are those in the precondition $Pre(a)$, while retracting any commitments on the truth value of atoms in $Add(a)$. Conversely, the progression state model will always search over full states $s \in \mathcal{S}(\Pi)$ as I prescribes the truth value of every atom, and the progression transition function $f(a, s)$ provides an explicit mapping between states $s, s' \in \mathcal{S}(\Pi)$.

2.2.2 Simulators

Many problems can have transition functions that are difficult to model using an expressive symbolic planning formulation such as STRIPS, however a simulator of the transition function is readily available (Frances et al. 2017). One benchmark set of environments that we use are the Atari-2600 video games which we discuss in further detail in 2.5.3. While compact symbolic descriptions for each Atari-2600 game are not available, a *black-box* simulator of the Atari games is readily accessible through the Arcade Learning Environment (ALE) (Bellemare et al. 2013) interface.

Black-box simulators provide the agent with only the ability to call the A, T and c functions of an MDP (Definition 2.1) through the same interaction as the agent-environment interface (Figure 1.1). That is, the agent can send an action to the simulator given the simulator's current state to receive the resulting state and cost.

There are a number of different attributes a simulator of an ASDM environment can have. A simulator can have the same transition function as the environment or it can provide an approximation or relaxation of the environment's true transition function. Throughout this thesis we explore problems with simulators that use the same transition function as the environment, that is $T^E(s, a, s') = T^S(s, a, s')$, where T^E is the environment's transition function and T^S is the simulators. Environment simulators typically used by RL algorithms allow for repeated episodes of a Finite-Horizon MDP (2.1.1) problem to be executed, but do not allow the simulator to be set to any state $s \in \mathcal{S}$. We refer to these simulators as non-settable black-box simulators. Simulators that do allow for setting their state to any $s \in \mathcal{S}$, like that ALE Atari-2600 simulator, we refer to as settable black-box simulators. In this thesis we mainly focus on methods that use settable black-box simulators.

2.3 Defining sets of states: polytopes and bounding boxes

One way to define sets of states for MDPs is through polytopes or bounding boxes. We follow definitions as presented by Borrelli et al. (2017).

Definition 2.6 (Convex Set)**.** A set $S \subseteq \mathbb{R}^d$ is convex iff $\lambda z_1 + (1 - \lambda)z_2 \in S$ for all $z_1 \in S, z_2 \in S$ and $\lambda \in [0, 1]$.

Definition 2.7 (Convex Hull)**.** The convex hull of a set $S \subseteq \mathbb{R}^d$, denoted as $conv(S)$, is the set of all convex combinations of the points in S. That is $conv(S) \triangleq \{\lambda^1 x^1 + \ldots + \lambda^k x^k \mid x^i \in S, \lambda^i \geq 0, i = 1, \ldots, k, \sum_{i=1}^{k} \lambda^i = 1\}$.

A convex hull of a set S has the property that if C is any convex set with $S \subseteq C$, then $conv(S) \subseteq C$. There are two equivalent definitions for polytopes, the $\mathcal{H}$-representation and $\mathcal{V}$-representation. We define a polytope, $P \in \mathbb{R}^d$, in $\mathcal{H}$-representation as the bounded intersection of a finite set of m closed half-spaces in $\mathbb{R}^d$:

Definition 2.8 (Polyhedron)**.** A polyhedron $\mathcal{P} \in \mathbb{R}^d$ is an intersection of a finite set of m closed half-spaces in $\mathbb{R}^d$, $\mathcal{P} \triangleq \{x \in \mathbb{R}^d \mid Ax \leq b\}$, where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$.

Definition 2.9 (Convex Polytope ($\mathcal{H}$-representation))**.** A convex polytope, P, is a *bounded* polyhedron $\mathcal{P}$ which does not contain any ray $\{z_1 + tz_2 \mid t \geq 0, z_1 \in \mathcal{P}, z_2 \in \mathcal{P}\}$.

Alternatively, a polytope can be defined by the $\mathcal{V}$-representation as follows:

Definition 2.10 (Convex Polytope ($\mathcal{V}$-representation))**.** A convex polytope P of any finite set of points $S \in \mathbb{R}^d$ is $P \triangleq conv(S)$.

Additionally we define extremes of a polytope as,

Definition 2.11 (Extremes of a Convex Polytope)**.** The set of extreme points of the convex polytope P, denoted as $extremes(P)$ is the set, $extremes(P) \triangleq \{e \in P \mid e \neq \lambda z_1 + (1-\lambda) z_2, \lambda = (0, 1), z_1 \in P, z_2 \in P\}$.

We also define bounding boxes with an ϵ-padding in each dimension as,

Definition 2.12 (ϵ-Bounding Box)**.** The ϵ-Bounding Box for a set $S \subseteq \mathbb{R}^d$ is the hyper-rectangle between the two points $l = [min_0(S) - \epsilon, min_1(S) - \epsilon, \ldots, min_d(S) - \epsilon]$ and $u = [max_0(S) + \epsilon, max_1(S) + \epsilon, \ldots, max_d(S) + \epsilon]$, where $min_m(S)$ and $max_m(S)$ are the minimum and maximum values along dimension m, for the points in S.

2.4 Supervised Learning

Supervised Learning (SL) is a common approach for creating predictors from data samples through empirical risk minimisation (Hardt and Recht 2021) with a goal of generalising to out-of-sample data. That is, the objective of SL given a function class $\mathcal{H} \subseteq \mathcal{X} \rightarrow \mathcal{Y}$ and a set of data samples $(\bar{x}_1, y_1), (\bar{x}_2, y_2), \ldots, (\bar{x}_i, y_i)$ is defined as,

$$\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} loss(h(\bar{x}_i), y_i) \tag{2.4}$$

We make use of SL approaches in Chapters 4 and 7.

2.5 Benchmark environments

In the worst-case solving for optimal policies for MDPs is known to be intractable (Mundhenk et al. 2000), hence it is common practise to compare ASDM algorithms, not on worst-case guarantees, but instead through their relative practical performance on a set of benchmark environments. Here we discuss the different benchmarks that we use throughout the thesis to evaluate algorithms we explore and propose.

2.5.1 SSPs: GridWorld, Canadian Travelers Problem, and RL acid

GridWorld (Sutton and Barto 2018) domains, are instances of a SSP problem. The goal in GridWorld is to move from an initial position in a grid to a goal position. In each state 4 actions are available: to move up, down, left or right. Any action that causes a move outside of the grid results in no change to the agent's position. Actions have a cost of 1, with the exception of actions that result in reaching the goal state, that have a cost of 0. The complexity of GridWorld can be scaled through the size of the grid and the location and number of goals. GridWorld also allows for extensions, which we use to have domains with a stationary goal, moving goals, obstacles and partial observability all of which we explore in Chapter 3.

Having partially observable obstacles in GridWorld provides an instance of the stochastic Canadian Traveller Problem (CTP) (Papadimitriou and Yannakakis 1991). The objective in CTP is to find the shortest path between two locations in a road map, however, there is a known probability for each road in the map being blocked due to weather conditions. A road in CTP can only be observed as being blocked or unblocked by visiting a location connected to it, and once a road status is observed the status remains unchanged. For CTP we model the partially observability through representing the probabilities of the obstacles implicitly in the simulator.

John Langford designed two MDP problems[1] described as Reinforcement Learning (RL) "Acid" intended to be difficult to solve using common RL algorithms, such as Q-learning. Langford's two problems are M tiled corridors which allow two actions from every state. The first of Langford's problems is named Antishaping and uses reward shaping to encourage actions away from the

[1]https://github.com/JohnLangford/RL_acid

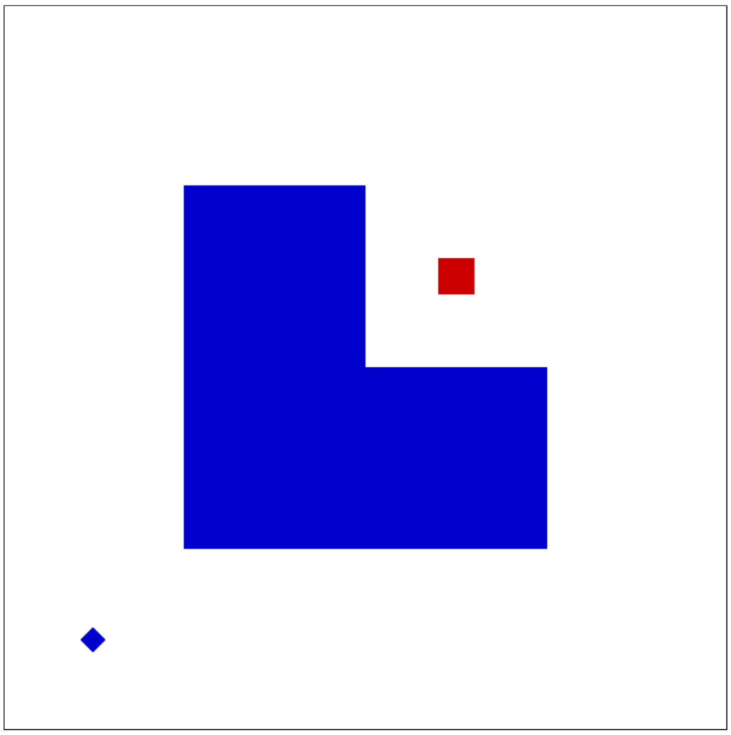


FIGURE 2.1: Mobile-Robot domain. The small blue square is the agent, red square is the goal and the large blue object is a pit from which the WalkBot can not escape.

goal state. Antishaping has one consistent action which always moves the agent towards the goal while the other action moves the agent away from the goal. At either end of the corridor one of the actions cause no change in the agent's location. The cost of each transition in Antishaping is 0.25 divided by the distance between the successor state and the goal, except when the resulting state is the goal location which has a cost of 0. The motivation behind Langford's second problem, Combolock, follows from observing that if many actions lead back to a start state, random exploration is inadequate. The Combolock problem has one action that moves the agent to the end of the corridor that does not contain the goal and one action that results in the agent progressing a single tile closer to the goal. The cost of each action in Combolock is 1 except for the transition that leads to the goal tile where the cost is 0.

2.5.2 Motion planning

The Mobile-Robot domains are instances of motion planning with dynamical constraints. This problem is known to be PSPACE-complete (Reif 1979). The Mobile-Robot domain, illustrated in Figure 2.1 requires an agent to reach a goal square and have its velocities below a threshold. Mobile-Robot's observable state is (x, y, v_x, v_y) where (x, y) is the agent's position in 2 dimensional space and (v_x, v_y) are first-order derivatives of the agent (velocities) in the x and y directions. The agent controls the second-order derivatives of its position through the action space (a_x, a_y), that is, the acceleration in each dimension. The range of acceleration in each dimension is $[-2, 2]$. If the agent moves outside of the bounds ($0 \geq x \geq 10$, $0 \geq y \geq 10$) or moves into an obstacle, illustrated in Figure 6.2, it will remain there for the rest of the episode. At each step when the agent is within the goal square and both $\text{abs}(v_x) \leq 0.2$ and $\text{abs}(v_y) \leq 0.2$ the cost of actions are 0, otherwise the cost is 1. We use 3 instances of Mobile-Robot, each with a time horizon of 100 steps. The instances are: (1) *Mobile-Robot* is free of obstacles and terminates when $s \in \mathcal{S}_G$, (2) *Mobile-Robot-Ob* has obstacles and terminates when $s \in \mathcal{S}_G$, (3) *Mobile-Robot-Ob-Stay* has obstacles and only terminates at time horizon. There are clear high-level modes of behaviour required by each instance. The *Mobile-Robot* domain has to accelerate towards the goal, then decelerate so that its velocity is below the required threshold once in the goal square. When adding the obstacle, given the start state shown in Figure 6.2 the agent needs to accelerate either right or upwards, turn around the obstacle, accelerate towards the goal and then decelerate. *Mobile-Robot-Ob-Stay* requires all the previous behaviours and to stay in the goal area once there.

2.5.3 Atari-2600

The Atari-2600 games have widespread use in literature for comparing and evaluating planning and learning techniques (Bellemare et al. 2013; Machado et al. 2018). The Atari-2600 games provide a challenging set of domains for ASDM agents, as they were designed to be a challenge for humans and each game requires different behaviours in order to maximise rewards. The Atari-2600 games can be accessed through the Arcade Learning Environment (ALE) (Bellemare et al. 2013) which provides a typical Reinforcement Learning (RL) environment interface where given a state, the agent selects an action and receives a resulting state and reward. The Atari-2600 games require 60 actions per second of game play and also have a large branching factor, of 18 actions, making it unfeasible to use a brute-force Planning method (Bellemare et al. 2013). Depending upon the setting, the states perceived by the agent can be either the game's screen pixels or RAM values.

2.5.4 Classical and probabilistic planning benchmarks

Classical planning problems are often specified using the Planning Domain Definition Language (PDDL) (Aeronautiques et al. 1998). PDDL uses two parts to describe a problem. The first part describes the domain which defines a set of predicates and action schemas. The action schemas define action effects on predicates concerning a single object or set of objects. The other part specifies a particular problem instance of the domain by defining the objects that exist as well as the problem's initial state and goal description. Relational Dynamic Influence Diagram Language (RDDL) (Sanner et al. 2010) is an alternative planning language which allows for stochastic processes such that RDDL can model MDP problems where actions are stochastic. Problems are defined in RDDL similarly to PDDL in that they contain separate domain and instance descriptions.

The International Planning Competitions (IPC), which started in 1998 (McDermott 2000), provide sets of classical and probabilistic planning domains which are used to compare and determine state-of-the-art classical and probabilistic planning algorithms. There are a number of different objectives/metrics that can be used when comparing planners on any given domain. Broadly, metrics which are typically used to compare planning algorithms are coverage (how many problems the planner can solve), plan quality, and solving time. The classical planning tracks in the most recent IPC in 2018 [2] included an optimal track, bounded-cost track, satisficing track and an agile track. Each different track compares planner's performance using different metrics, the optimal track focuses on the number of tasks solved optimally, the bounded cost track evaluates the number of solved tasks with a cost equal to or less than a given bound, the satisficing track rewards the number of tasks solved as well as lower cost solutions, and finally the agile track evaluates the time to solve a task while ignoring the cost of the solution paths. In this thesis when using probabilistic planning benchmarks in Chapter 3 and classical planning benchmarks in Chapter 7 we focus on both coverage and the quality of the plans found which is directly related to the satisficing track.

2.6 Planning: uninformed and heuristic search

A classical planning problem, described in 2.2.1 can be interpreted as a path finding problem using weighted directed graphs where nodes represent states, edges are actions and the edge weights correspond to action costs (Geffner and Bonet 2013). Therefore standard path finding algorithms like Depth-First Search (DFS), Breadth-First Search (BFS) or the Dijkstra's algorithm (Dijkstra 1959) can be used to solve for a planning problem. DFS, BFS and Dijkstra are all blind search algorithms meaning that only information gathered from the nodes visited in the search affects

[2]https://ipc2018-classical.bitbucket.io/

the future search behaviour. More specifically, blind search does not use any information from the unexplored nodes/actions in the search space, including information about the goal, to guide its search behaviour (Pearl 1984). In contrast to blind search, heuristic search algorithms use an evaluation function called a heuristic to determine which actions provide the most promise of reaching the defined goal (G in Definition 2.3).

The selection of which heuristic and search algorithm to use depends on both the objective of the search (some example objectives are discussed in 2.5.4) and structure of the problem. Heuristic search algorithms often use Best-first Search methods, of which A* and Greedy Best First Search (GBFS) are common instantiations, which we discuss below,

Best-first Search: Best-first Search methods use an OPEN list of nodes which are yet to be expanded by the search and a CLOSED list which have been expanded. Nodes in the OPEN list are sorted according to an evaluation function $f(s)$ which varies across different search algorithms but in the case of an informed search $f(s)$ makes use of a heuristic function $h(s)$. Best-first starts with an empty CLOSED list and an OPEN list containing the initial state of the problem. Nodes are removed from the OPEN list according to their sorted order and if they are not already in the CLOSED list they are expanded, meaning all their successor states s' and put into the OPEN list, and the state is then put into the CLOSED list. Best-first Search continues its process until a given termination condition which can that a goal state has been found or a certain amount of compute has been consumed.

A*/WA*: A* (Hart, Nilsson, and Raphael 1968) is one instantiation of a Best-first search where $g(s)$ is equal to the accumulated cost of the path to reach s and the evaluation function is $f(s) = g(s)+h(s)$. An admissible heuristic function $h(s)$ for A* is one such that $h(s) <= V*(s)$ (for $V*(s)$ see Equation 2.2). In the case that $h(s)$ is admissible, A* guarantees optimality, as a goal state will not be expanded unless it is impossible for states in the open list to reach the goal with a smaller cost. The Weighted A* (WA*) algorithm (Pohl 1970) trades off the optimally guarantee of A* for the speed of finding a plan by weighting the heuristic function term in the evaluation function. WA* evaluation function is $f(s) = g(s) + \omega h(s)$, where for large ω values WA* priorities expanding states with smaller $h(s)$ values which diminishes the influence of $g(s)$.

GBFS: For the GBFS instantiation of the Best-first Search algorithm $f(s) = h(s)$. GBFS is often able to solve problems faster than methods such as A* as GBFS prioritises speed of finding a goal state over the plan quality found. GBFS prioritises speed as it orders the expansion of nodes only according to the heuristic function, ignoring the cost of the path to a reach a node.

Therefore, GBFS is commonly used over other best-first search methods when the objective of the search algorithm is coverage and solving speed, and not optimally.

The accuracy of a heuristic function is key to the success of Heuristic Search algorithms because as the accuracy of the heuristic decreases the number of states needed to be expanded for a problem can increase exponentially (Helmert, Röger et al. 2008). The overall best performing heuristics over the set of classical planning benchmarks discussed in 2.5.4 tend to be ones which use information about the transition function T, such as the Fast-Forward (FF) heuristic (Hoffmann and Nebel 2001) which solves a delete-relaxation of the problem to provide heuristic values. In the next section we discuss planning methods and heuristics for problems where a symbolic description of the transition function is not provided and is instead provided through a black-box simulator as described in 2.2.2.

In Chapter 3 we introduce a new heuristic search algorithm for online planning and in Chapter 7 we use GBFS to evaluate how informed a learnt heuristic function is.

2.7 Online planning over simulators

Planning over black-box simulators means that we can not use planning or heuristic methods that rely upon a symbolic description of the transition function such as the FF heuristic (Hoffmann and Nebel 2001). Instead, for black-box simulators cost-to-go approximations can be obtained through simulation, which we explore in Chapter 3, or through learning methods which we investigate in Chapters 4 and 7.

Online planning interleaves planning and acting. In online planning the planner iteratively works on what move to do next with a certain planning budget for each time step in the environment. Online planning methods have the advantage over offline methods that they do not require the algorithm to compute a complete solution path upfront before interacting with the environment. Online planning is especially advantageous when the simulator of the environment is inaccurate as plans obtained from offline planning often assume the simulator provides a perfect model of the environment and can not adapt to inaccuracies in the environment that are observed when executing the plan. Online planning lends itself to environments where there is time to plan between each move such as games like Chess or Poker. In Chapters 3, and 4 we explore the online planning over a black-box simulator problem.

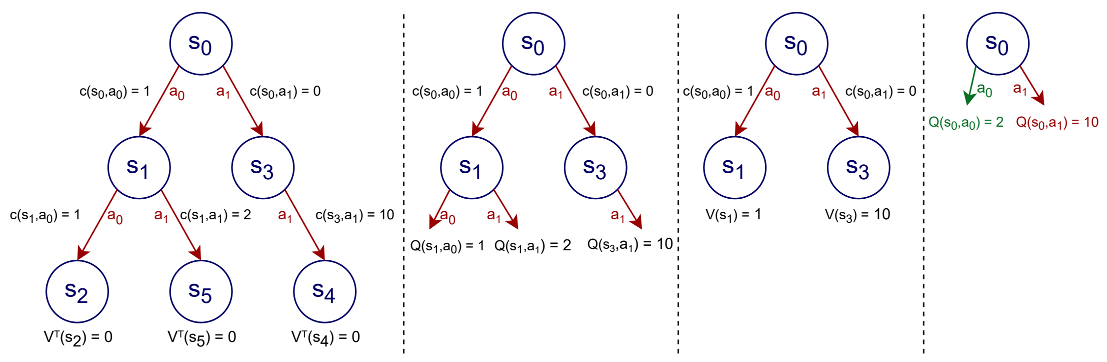


FIGURE 2.2: Example lookahead showing the action selection process for the root node s_0 following Definition 2.14. On the left, we have a fully built lookahead. From left to right, we show the recursive process to determine what is the action to be executed. The transition with action a_0 from s_0 shown in green in the last diagram on the right is the transition added to the lookahead's critical path (Definition 2.15). The V, Q, and V^T functions are as defined in Definition 2.14.

2.7.1 Lookaheads for online planning

Lookaheads can be used to consider costs for different action trajectories from the current state into the future. An example of this is shown in Figure 2.2 where a lookahead is illustrated. We define the notion of lookahead as,

Definition 2.13 (Lookahead)**.** A lookahead is defined as L = (N, C, s_r) where N is a set of nodes defined as state-action paths starting at the root state of the lookahead s_r, and C is a function that given a node $n \in N$ and an action $a \in A(s)$ returns the children of n, that is $n_c \in C(n, a)$ and $n_c \in N$.

Through backing up the costs for each node in the lookahead, as shown in Figure 2.2, an expected value can be found for each action applicable at the current state. That is, where n^s is the last state along the state-action path of node n, the operation of backing up the rewards and selecting which action to execute is,

Definition 2.14 (Action selection of lookahead)**.** Given a lookahead L= (N, C, s_r) (Definition 2.13) the action to execute a is selected at the root n_r, where $n_r^s = s_r$, by $\text{argmin}_{a \in A(n_r^s)}\{Q(n_r, a)\}$, where $Q(n_r, a) = c(n_r^s, a) + \sum_{n \in C(n_r, a)} T(n_r^s, a, n^s)V(n)$, and $V(n) = V^T(n)$, with $V^T(n)$ being a termination cost, when $\bigcup_{a \in A(n^s)} C(n, a) = \emptyset$, otherwise $V(n) = \min_a\{c(n^s, a) + \sum_{n' \in C(n,a)} T(n^s, a, n'^s)V(n')\}$.

Once an action is selected for execution, the lookahead is updated to have its root at the selected action's resulting node and the lookahead continues being constructed from the new root node. We define the action selected by the agent as a part of its *critical path*. That is,

Definition 2.15 (Critical Path)**.** Given the action selected a_t at each time step $t = 0, 1, \dots, m$ following Definition 2.14, the critical path ρ is the sequence of states and actions $\rho = (s_0, a_0, s_1, a_1, \dots, a_{m-1}, s_m)$, such that $s_{i+1} \sim T(s_i, a_i, \cdot)$ for $i = 0, \dots, m-1$.

2.7.2 The rollout algorithm

A particularly effective online planning approach that uses a lookahead to obtain suboptimal controls is *rollout*. The rollout algorithm *approximates* the optimal cost-to-go from the current state s_k, $V^*(s_k)$, by the cost of some suboptimal policy and a d-step lookahead strategy. The seminal Real Time Dynamic Programming (RTDP) (Barto, Bradtke, and Singh 1995) algorithm, is an instance of the rollout strategy where the lookahead is uniform, $d = 1$, and actions $\bar{\pi}(s_k)$ selected at stage k and for state s_k are those that attain the minimum

$$\min_{a_k \in A(s_k)} E\left\{c(s_k, a_k) + \tilde{V}_{k+1}(s_{k+1})\right\} \tag{2.5}$$

where the expectation is over $s_{k+1} \sim T(s_k, a_k, s_{k+1})$ and $\tilde{V}_{k+1}$ is an approximation of the optimal cost-to-go at time step $k+1$, V^*_{k+1}. If $\tilde{V}_{k+1} \leq V^*_{k+1}$, that is the approximation is from below, we will refer to it as a *base heuristic*, and can either be problem specific (Eyerich, Keller, and Helmert 2010), domain independent (Bonet and Geffner 2003; Yoon, Fern, and Givan 2007) or *learnt* from interacting with a simulator (Mnih et al. 2015). Alternatively, $\tilde{V}_{k+1}$ can be defined as approximating the cost-to-go of a given suboptimal policy π, referred to as a *base policy*, where estimates are obtained via *simulation* (Rubinstein and Kroese 2017). We will denote the resulting estimate of cost-to-go as $H_k(s_k)$[3]. The result of combining the lookahead strategy and the base policy or heuristic is the *rollout policy*, $\bar{\pi}$ $\{\bar{\mu}_0, \bar{\mu}_1, \dots, \bar{\mu}_{N-1}\}$ with associated cost $\bar{V}(s_k)$. Such policies have the property that for all s_k and k

$$\bar{V}_k(s_k) \leq H_k(s_k) \tag{2.6}$$

when H_k is approximating from above the optimal cost-to-go V^*_k, as shown by Bertsekas (2017) from the DP algorithm that defines the costs of both the base and the rollout policy. To compute at time k the rollout control $\bar{\mu}(s_k)$, we minimize over the values of the Q-factors of state and

[3]We use the subindex k to emphasize that the result of simulating a policy depends on the time step.

action pairs (s_l, a_l),

$$Q_l(s_l, a_l) = E\{c(s_l, a_l) + \min_{a \in A(s_{l+1})} Q_{l+1}(s_{l+1}, a)\} \tag{2.7}$$

over the expectation $s_{l+1} \sim T(s_l, a_l, s_{l+1})$ and for admissible controls $a_l \in A(s_l)$, $l = k + i$, with $i = 0, ..., d - 1$, and

$$Q_l(s_l, a_l) = E\{H_l(s_l)\} \tag{2.8}$$

for $l = k + d$ and all $a_l \in A(s_l)$. Note that Equations 2.7 and 2.8 are equivalent to the action selection of a lookahead (Definition 2.14) where $V^T(n) = H_l(n^s)$.

2.7.3 Cost-to-go approximation via simulation

The most successful methods for obtaining cost-to-go approximations have revolved around the idea of running a number of Monte Carlo simulations of a suboptimal base policy π (Ginsberg 1999; Coulom 2006), when used within the rollout algorithm this is referred to as Monte Carlo Tree Search (MCTS). The Monte Carlo simulations amount to generating a given number of samples for the expression minimized in Equation 2.5 starting from the states s_l over the set of admissible controls $a_l \in A(s_l)$ in Equation 2.8, and averaging the observed costs. Three main drawbacks (Bertsekas 2017) follow from this strategy. First, the costs associated with the generated trajectories may be wildly overestimating $V^*(s_l)$ yet such trajectories can be very rare events for the given policy. Second, some of the controls a_l may be clearly dominated by the rest, not warranting the same level of sampling effort. Third, initially promising controls may turn out to be quite bad later on.

One of the most striking properties of rollout algorithms is the *cost improvement* property in Equation 2.6, suggesting that upper bounds on costs–to–go can be used effectively to approximate the optimal costs V^*. *Stochastic enumeration* (SE) (Rubinstein and Kroese 2017) sampling techniques can be used to obtain an *unbiased estimator* for upper bounds on costs-to-go, or in other words, estimates of the maximal costs a stochastic rollout algorithm with a large depth lookahead can incur. SE algorithms have been used very successfully to approximate counts of models for CNF models and other #P problems (Rubinstein and Kroese 2017).

SE methods are inspired by a classic algorithm by D. E. Knuth to estimate the maximum search effort by backtracking search (1975). Knuth's algorithm estimates the total cost of a tree T with root u keeping track of two quantities, C the estimate of the total cost, and D the expectation on the number of nodes in T at any given level of the tree, and the number of terminal nodes once the algorithm terminates. Starting with the root vertex u, the algorithm proceeds by updating D to be $D \leftarrow |\mathcal{S}(u)|D$ and choosing randomly a vertex v from the set of successors $\mathcal{S}(u)$ of u. The estimate C is then updated $C \leftarrow C + c(u, v)D$ using the cost of the edge between vertices u and v. These steps are then iterated until a vertex v' is selected s.t. $\mathcal{S}(v') = 0$. While Knuth's algorithm estimates are an unbiased estimator, the variance of this estimator can be exponential on the horizon, as the accuracy of the estimator lies on the assumption that costs are evenly distributed throughout the tree (Rubinstein and Kroese 2017).

2.7.4 Minimising regret

The regret of a policy is its loss from not selecting the optimal action at every time-step. Popular instantiations of the rollout algorithm involve using a selective strategy for the d-step lookahead that make use of upper confidence bounds (Auer, Cesa-Bianchi, and Fischer 2002) like Kocsis and Szepesvari's (2006) Upper Confidence bounds applied to Trees (UCT) algorithm, which aims to minimise regret. For each rollout of the states within the d-step lookahead UCT selects actions greedily according to the current $Q_l(s_l, a_l)$ values as calculated in Equations 2.7 and 2.8 plus a bonus term based upon the upper bound confidence limits. The upper bound confidence limits bonus term uses the number of visits to the node in the lookahead and the number of times action a_l has been selected by the UCT algorithm. Using the bonus term to encourage exploration aims to minimise the cumulative regret of exploiting state-action trajectories that lead to promising Q values versus exploring new or rarely visited state-action trajectories in the search thus far.

2.7.5 Width-based search

Width-based search algorithms both focus the lookahead and have good any–time behaviour. When it comes to prioritisation of applicable actions, width–based methods select first those that lead to states with novel valuations of features defined over states (Lipovetzky, Ramírez, and Geffner 2015; Geffner and Geffner 2015). The original width-based measure introduced by Lipovetzky et al. 2012 is that the novelty of a state is evaluated by the smallest tuple of atoms $t \subseteq s$, where s is the first state that makes t true in the search.

The most basic of the width–based search algorithms is $IW(1)$ (Lipovetzky and Geffner 2012), a plain *breadth–first search*, guaranteed to run in *linear time and space* as it only expands *novel* states. $IW(1)$ classifies a state s_l as *novel* if and only if it encounters a state variable [4] $s^i \subset \mathbb{R}$, whose value $v \in D(s^i)$ has not been seen before in the current search. Note that novel states are independent of the objective function used, as estimated cost-to-go or the accumulated cost is not used to define the novelty of the states, although there have been follow up works that have explored alternative definitions of novelty that incorporate costs-to-go heuristics (Lipovetzky and Geffner 2017; Katz et al. 2017; Tuisov and Katz 2021).

2.7.6 Rollout-IW

Bandres et al. (2018) introduced a *depth-first* version of the IW(1) planner, named Rollout-IW(1) (RIW). RIW(1) is as an instance of a *rollout* algorithm, and aims to contain the same nodes that are expanded by the IW(1) planner. As RIW(1) performs depth-first search it was argued by Bandres et al. that it has better any-time performance than IW(1). The hypothesis for the better any-time performance of RIW(1) is that its search visits states that are further away from the initial state earlier in the search than its breadth-first search counter-part IW(1).

Algorithm 1 Overview of the RIW(1) Algorithm

```
Input  : A lookahead $L = (N, C, s_r)$, and a base policy $\pi_b$
Output : Updated lookahead $L$
1 while ¬ has_solved_label($s_r$) do
2     $s \leftarrow s_r$ // complete depth-first rollout from the root node's state
3     while is_novel($s$) ∧¬ is_terminal($s$) do
4         $s', a \leftarrow$ sample_unsolved_child($s, \pi_b$)
          $L \leftarrow$ update_lookahead($L, s, a, s'$), $s \leftarrow s'$
5     end
6     update_solved_labels($s$)
7 end
```

Algorithm 1 provides an overview of RIW(1) using the base policy π_b. RIW(1) was originally defined to use a random uniform base policy π_b, however any given policy can be used instead. The function `sample_unsolved_child`, samples an action $a \sim \pi_b(s)$ and a transition $s' \sim T(s, a, \cdot)$ provided that s' has not been marked as solved. If the selected transition does not already exist in the lookahead `update_lookahead` adds it. The `update_solved_labels` function adds a solved label to the given state and back-propagates the solved label to its parents

[4]In order to use the notion of novelty, we assume state spaces S to be stationary.

if the parent's children have all been marked as solved. Bandres et al. results showed that RIW(1) outperformed IW(1) over the Atari games greatly when almost real-time budgets for planning were applied.

In Chapters 3 and 4 we investigate and extend the Rollout-IW planning algorithm, and in Chapter 7 we introduce a novelty-based regression algorithm inspired by the family of width-based search algorithms.

2.8 Reinforcement Learning

RL is a trial and error based approach to learning an action policy for ASDM problems. RL uses feedback from the agent's interaction with the environment to update the action policy with the objective of maximising the reward signal (or minimising the costs depending on the problem definition) (Sutton and Barto 2018).

2.8.1 Temporal difference learning

Temporal difference (TD) learning is a method for incremental learning through bootstrapping (Sutton 1988; Sutton and Barto 2018). TD(0) is the simplest TD method where the target value for $V(s_k)$, given the action a_k selected by the agent and the resulting cost and state c_k, s_{k+1}, is $c_k + \gamma V_{s_k+1}$, where γ is the discount factor. An update of TD(0) is,

$$V(s_k) \leftarrow V(s_k) + \alpha[c_k + \gamma V(s_{k+1}) - V(s_k)] \tag{2.9}$$

where α is a constant step size parameter.

Sarsa is another TD method for learning Q values on policy through updates using the target value,

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \alpha[c_k + \gamma Q(s_{k+1}, a_{k+1}) - Q(s_k, a_k)] \tag{2.10}$$

Note that for on policy methods TD's task is to estimate the expected accumulated costs from the given state following the policy which underlies the transitions that it is trained on. Alternatively, Q-learning, which is an off-policy TD method, directly approximates the optimal Q function Q* through updates using the target values,

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \alpha[c_k + \gamma \min_{a \in A(s_{k+1})} Q(s_{k+1}, a) - Q(s_k, a_k)] \tag{2.11}$$

2.8.2 Policy optimisation methods

Policy optimisation methods consist of optimising for the parameters of the action policy π directly. These methods usually optimise the policies parameters through gradient descent using feedback from the agent's interaction with the environment.

One of the most commonly used policy optimisation methods and one we use in Chapters 5 and 6 is Proximal Policy Optimisation (PPO) (Schulman et al. 2017). PPO is a policy gradient method which iteratively uses data sampled from the environment using the policy to optimise an objective function which is a surrogate of the policy gradient theorem (Sutton et al. 1999). The surrogate objective function which PPO maximises via stochastic gradient descent is,

$$L(\theta) = \hat{E}_t \left\{ L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right\} \tag{2.12}$$

where c_1 and c_2 are coefficients, θ represents the parameters of the policy π and the value function V, S is an entropy bonus, L_t^{VF} is the value function loss that is $V_\theta(s_t) - V_t^{targ}$, and,

$$L_t^{CLIP}(\theta) = \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) \tag{2.13}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, ϵ is a parameter and the estimator of the advantage function is,

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.14}$$

where $\delta_t = c_t + \gamma V(s_{t+1}) - V(s_t)$ and c_t is the cost at time t.

2.8.3 Count-based exploration

The problem of addressing how RL methods should explore is very difficult, as observed by Tang et al. (2017) there are no fully satisfying exploration techniques for problems with high dimensional state spaces. Currently many of the state-of-the-art RL algorithms rely upon simulating random policies at the start of learning. When rewards are sparse, RL algorithms relying on random policies can suffer from "cold starts", as the training process requires a considerable number of episodes to improve over random action selection as there is little to no reward signal. There have been a number of attempts to create RL algorithms tailored to deal with sparse rewards and slow learning, which are very often tested on Atari-2600 games like Montezuma's Revenge.

One method that encourages exploration of RL algorithms are the so-called *count-based* RL algorithms (Bellemare et al. 2016a). Count-based methods are of particular relevance to this thesis, due to their relationship with width-based planning, additionally we also explore using count-based methods in Chapter 5. The intuition for count-based algorithms is to encourage exploration by augmenting the given cost function, which can be evaluated on any state but its structure is not known. Count-based methods add intrinsic rewards to state-action pairs inversely proportional to the number of times they have been visited. That is, where $N(s,a)$ is the state-action count and β is an exploration hyperparmeter, the augmented Bellman's equation (Equation 2.2) for count-based methods is as follows:

$$V_\pi(s) = \sum_{a \in A(s)} \pi(s,a) \Big(c(s,a) + \sum_{s' \in S} T(s, as') [V_\pi(s') - \beta N(s,a)^{-1/2}] \Big) \tag{2.15}$$

A number of issues have been observed that affect the applicability and generality of tabulating the visit counts. First, for large state spaces tabulation of state counts can become infeasible due to memory constraints. On top of that, there is no connection between states that are very similar. That is, even if a state is trivially different, for example in the case where the state is screen pixels and there are "background" pixels which provide no useful information as their values change randomly, a state that is otherwise identical besides the "background" pixels will have independent state counts. Bellemare et al. (2016a) and Martin et al. (2017) both tackle the difficulties of tabulating state visit counts by introducing two different methods for producing approximate counts, or pseudo-counts, for states. Bellemare et al. use a density model for states to produce a pseudo-count, while Martin et al. use feature space counts. Another approach to this issue was presented by Tang et al. (2017) where states are mapped to hash codes which are then used for the state counts. This method is particular effective on the Atari-2600 games when set up to represent the hashing function with an autoencoder which is trained online.

2.8.4 Imitation Learning

Three core approaches of Imitation Learning are Inverse Reinforcement Learning (IRL) (Ng and Russell 2000), Generative Adversarial Imitation Learning (GAIL) (Ho and Ermon 2016), and Behavioural Cloning (BC) (Bain and Sammut 1995). IRL aims to find a cost function that explains the observed behaviour. This cost function is used with RL to train a policy that imitates the observed behaviour. GAIL trains a policy on a cost function that is being simultaneously learnt to reward actions selected by the demonstration and not those generated by the policy. That is, each iteration GAIL samples trajectories from the environment using the imitator's policy and uses this data to train a discriminator that identifies if a state-action pair is from the imitator policy or the experts. The discriminator is then used in a cost function that is used to update the imitator policy. BC approaches imitation as a SL problem with demonstration states as input data and its actions as labels. BC is usually implemented by splitting the state-action pairs into a training set and a validation one. Training epochs are then run until the error rate on the validation set starts increasing.

2.8.5 Learning from demonstration and learning backwards

Salimans and Chen (2018) introduced a method for learning from a single demonstration, that instead of having the objective to imitate the demonstration's actions like imitation learning techniques, the objective of Salimans and Chen's (2018) work is to maximise rewards directly from states along a demonstrated trajectory. Salimans and Chen (2018) do this by running PPO learning iterations starting at states which are close to the end of the demonstration. Once the policy performs at least as good as the demonstration for a given threshold percentage of episodes, the starting states for the learning iterations move closer to the beginning of the demonstration. Through this method Salimans and Chen were able to achieve a score on the Atari game Montezuma's Revenge that at the time was the highest published score. A similar method developed independently around the same time as Saliman and Chen is Backplay (Resnick et al. 2018). Backplay differs in that the learning schedule is predefined manually such that the initial learning states progress towards the beginning of the demonstration depending upon the training epoch and not the policy's performance in comparison to the demonstration. Backplay was shown to be effective for Gridworld problems and a multiplayer zero-sum game. It should be noted that Hosu and Rebedea (2016) first proposed the idea of starting learning iterations from states provided by a human demonstration, however Hosu and Rebedea randomly sampled the starting state from the demonstration as opposed to using a learning schedule.

Floerensa et al. (2017) also used a backwards schedule in order to learn robotic manipulation but instead of relying upon a demonstration for the states from which to learn they automatically generated the starting states. Floerensa et al. automatically generate the states by starting at the goal state and then sampling random actions from the goal state in order to get a set of starting states from which to learn from. As training progresses the set of starting states is expanded further by sampling more random actions from states in the set of starting states. Floerensa et al.'s method relies upon the assumptions that at least one goal state is given and that all starting states have a non-zero probability of reaching the goal states and vise versa.

2.8.6 Options and Skill Chaining

The options framework (Sutton, Precup, and Singh 1999) allows for an RL agent to select both the primitive actions of the environment, that is the actions $a \in A(s)$, as well as closed-loop policies for taking actions which are called options. Options are also often referred to as skills.

Skill chaining learns one *option* at a time. An *option* describes the rules around an execution of skill through a description of a skill's initialisation set (where it can be executed from), its termination set, the policy to be executed and the maximum duration of the skill's execution.

Konidaris and Barto (2009a) introduced a method for option discovery that builds chains of skills that start close to the goal and are progressively learnt for states closer to the MDP's initial state. Along with a policy a skill has an initialisation set of states from which the skill can be chosen and a termination set of states from which the skill's policy stops executing and the RL agent is required to select a new action. Skill chaining starts first with a global policy which can be executed anywhere in the state space and executes a single action at a time. The global policy is run until the goal of the problem is found a given number of times. Using the last N transition of successful trajectories that reached the goal a skill is learnt. Then a new skill is learnt with a termination set equal to the initialisation set of the previously discovered skill. This method is repeated until a skill is learnt with an initialisation set that contains the initial state of the problem. In the original skill chaining paper (Konidaris and Barto 2009a), logistic regression classifiers were used to learn the initialisation and termination sets. Deep skill chaining (Bagaria and Konidaris 2020) follows up skill chaining with policies defined through neural networks and trained using Deep Deterministic Policy Gradient (Lillicrap et al. 2015).

The skill chaining algorithm has strong links to learning backwards methods discussed in the previous section, however unlike the learning backwards methods skill chaining only learns from the initial state of the MDP and does not set training episodes to have initial states close to the goal. In Chapter 6 we introduce a method that learns piece-wise policies starting backwards from the goal which is particularly related to skill chaining.

2.9 Planning and learning

2.9.1 Learning models to plan over

Sutton's Dyna-Q architecture is one of the earliest integrations of Planning and learning (Sutton 1990, 1991). Dyna-Q uses the transitions generated during training to create a model of the environment that maps a state and an action into a new state and a reward value. This model is then used to plan over, generating new trajectories where actions are chosen rather than randomly, on the basis of their expected outcomes. While the Dyna architecture is general, it is limited by how well the policy used to acquire experience with the environment can capture appropriately transitions relevant to optimal policies. Sutton proposed the epsilon-greedy policy, for the Dyna-Q algorithm. The epsilon-greedy policy is performed over the value function that is being trained, which is often poorly informed in the early stages of training. Learning a model to plan over is a difficult task. The works by Chiaapa et al. (2017) and Oh et al. (2015) used a Recurrent Neural Network (RNN) architecture to learn a model of the Atari-2600 environments through interaction. The learned model was shown to be accurate on the Atari-2600 games over a large number of time steps, thus enabling a planner to be used on the Atari-2600 games without having access to the simulator. However, the evaluation of the RNN model is significantly slower than simulating the environment directly (Machado et al. 2018). In 2.9.3 we discuss MuZero (Schrittwieser et al. 2020) which successfully learnt a latent model of the Atari-2600 simulator over which it could plan.

2.9.2 AlphaGo and AlphaZero

AlphaGo (Silver et al. 2016) is a Planning and Learning agent that accomplished the long standing challenge of developing an autonomous agent that could defeat a professional human player in the game of Go. AlphaGo uses two Neural Networks to represent a policy and value function which are incorporated into a lookahead policy. Silver et al. bootstrapped AlphaGo's policy network through SL on moves made by human experts and subsequently used RL to train the network through self-play. The value function of AlphaGo is trained to predict the winner of the policy network self-play games. The learnt policy and value function were subsequently used within a Monte Carlo Tree Search (MCTS) lookahead through sampling actions according to the policy network and evaluating the states within the lookahead with the value function network.

Silver et al. (2017) followed up AlphaGo with AlphaZero which represents both the value function and policy network through a single Neural Network which outputs both action probabilities and a value estimation of the probability that the current player wins the game. AlphaZero unlike AlphaGo does not require bootstrapping of the policy network learning with human expert moves but instead trains only on data from self-play. AlphaZero's self-play uses the MCTS lookahead utilising the policy and value function networks as previously described. It then trains the policy and value network using the probability outputs from the MCTS lookahead at each game state and information about which player won the game. The updated policy and value network are then used in the next iteration of self-play games. Through this method AlphaZero was able to defeat AlphaGo 100-0. AlphaZero was later generalised to achieve state-of-the-art performance in the games of Chess and Shogi (Silver et al. 2018).

2.9.3 MuZero

Schrittwieser et al. (2020) followed up on the AlphaGo (Silver et al. 2016) and AlphaZero (Silver et al. 2017) algorithms with MuZero. Unlike AlphaGo or AlphaZero, MuZero does not require a simulator or model of the game environment but instead learns a model of the environment through interaction. MuZero uses the learnt model to perform MCTS lookaheads similar to AlphaZero. MuZero also extends beyond the zero-sum games that AlphaZero was designed for to typical single-agent MDP environments like the Atari-2600 games. MuZero achieves state-of-the-art performance in the Atari-2600 games when compared to existing model-free RL algorithm performances.

2.9.4 Width-based planning and learning

Junyent et al. (2019) follow up on Bandres et al.'s (2018) use of a random policy for the base policy (π_b) of RIW (discussed in 2.7.6) with π-IW(1), an algorithm that replaces π_b with a trained policy defined over a NN. The intended effect is to orient the lookahead to promising areas of the state space. The NN from the trained policy is also used to extract features from the screen pixels for the computation of state novelty. Recently Junyent et al. (2021) introduced π-IW(1)+ and π-HIW(n, 1) as follow ups of π-IW(1). π-IW(1)+ modifies π-IW(1)'s random breaking of ties for the action selection (Definition 2.14) to select the action with the branch of the lookahead that contains the most nodes. π-IW(1)+ also adds a learnt value function, $\tilde{V}$, which is used in the action selection (Definition 2.14) by modifying $V(s)$ to be $\min\{\tilde{V}(s), V^*(s)\}$, where $V^*(s)$ is $V(s)$ as described in Definition 2.14. π-HIW(n, 1) is a hierarchical algorithm that has a high-level planner which uses a coarse down-sampling of the screen pixels as a feature set and a low-level planner which uses π-IW(1)+ with the feature set defined through the policy network as previously described. The high-level planner uses a modified stochastic exploration policy, that selects actions with probability inversely proportional to state visitation counts.

2.9.5 Learning Neural Network defined heuristic functions

Ferber et al. (2021) describe two different methods for learning heuristics for classical planning problems. The first is *per domain* learning, where a heuristic is learnt that is applicable to any instance of a particular domain, where a domain defines a set of action schemas and predicates which are used to formulate instances Π. The second framework introduced by Ferber et al. (2021), is to learn instance-based heuristics. That is, given some instance $\mathcal{S}(\Pi)$ with initial state s_0, per instance learning seeks heuristics h that apply to instances $\mathcal{S}(\Pi_1)$, $\mathcal{S}(\Pi_2)$, $\ldots$, $\mathcal{S}(\Pi_i)$ where all the instances $\mathcal{S}(\Pi_i)$ share all structural elements with $\mathcal{S}(\Pi)$ but initial states. That is, each $\mathcal{S}(\Pi_i)$ features a distinct initial state s_0^i, which is reachable from s_0 in $\mathcal{S}(\Pi)$.

Ferber et al. (2020) provide a great study of learning per instance Neural Network defined heuristic functions for classical planning. Ferber et al. (2020) explore a method, we refer to as Teacher-based Supervised Learning (TSL), which constructs a training set of states that are along solution paths generated from a teacher planner. TSL performs 200 step random walks from the initial state of an instance and then uses a GBFS with h^{FF} to search for a plan. The states along each solution path are labeled with an estimate of their distance from the goal according to the solution path. Ferber et al.'s method then performs SL on the states and goal-distance estimates in order to learn a heuristic.

Yu et al. (2020) introduced an alternative SL method for learning per-instance heuristic functions. Yu et al.'s algorithm (SING) performs backwards searches starting from a randomly sampled goal state $s_g \in S_G$. SING uses multiple depth-first searches (DFS) that prune duplicate states. The DFS use fixed node expansion expansion budgets, to search backwards from fully assigned randomly sampled goal states. SING then uses SL on all states visited within the DFS labelled with the depth at which the states were visited in the search as an estimate of their distance to the goal.

More recently Ferber et al. (2021) introduced three new NN defined heuristics functions for classical planning problems trained with RL inspired approaches. Two of the approaches introduced, h^{Boot} and h^{BExp}, are based on the idea of bootstrapping originally introduced by Arfaee et al. (2011), which entails training a heuristic on successively harder to solve states. h^{Boot} and h^{BExp}, perform regressions from the goal following random walks, and then solve a fully assigned state randomly sampled from the partially assigned state discovered by the regression. At the state found from the regression, a GBFS is run using the current h^{Boot} or h^{BExp} heuristic. h^{Boot} uses the plan's length as the training label for the state's estimated distance from the goal and discards any state where a plan is not found by GBFS. h^{BExp} uses the number of states expanded by the GBFS as the state's label and does not discard unsolved states from the training data. h^{Boot} or h^{BExp} use the states and labels to iteratively train their NNs, and once the GBFS is solving 95% of the states found from the random-walk regressions, the maximum length of the regression is doubled. Ferber et al. also introduced the h^{AVI} heuristic which is trained using approximate value iteration. h^{AVI}, similar to the bootstrapping heuristics, discovers states through random length regressions clipped at a constant maximum length. Instead of solving the sampled states, h^{AVI} performs Bellman updates on a 2 step lookahead from the state, evaluating the leaf states of the lookahead with the current h^{AVI} heuristic or as 0 if they are goal states. Ferber's motivation for the RL inspired approaches versus SL approaches is that the SL approaches are limited to instances small enough for training data generation.

Beyond learning per-instance NN defined heuristics, Shen et al.'s STRIPS Hypergraphs Networks (HGNs) learns per-domain and even domain independent NN defined heuristics. The STRIPS HGN (Shen, Trevizan, and Thiébaux 2020) is an extension of Graph Networks to Hypergraphs used to capture and learn the relationships of atoms within the problems current state, the goal and the action schemas. Shen et al. (2020) showed STRIPS-HGN is able to generalise across different domain instances and even across different domains on small classical planning instances. Similar to TSL, STRIPS-HGN also requires a planner to provide the distance of training states from the goal, and in particular, Shen et al. (2020) use an optimal planner.

In Chapter 7 we introduce a new method for learning per instance heuristics that are defined over Neural Networks.

Part I

Cost-to-go approximation for model-free planning

Introduction to Part 1

Cost-to-go approximations for MDPs problems can help focus a search method on promising areas of the search-space, and many state-of-the-art planning algorithms use such heuristic search methods. For model-free planning problems the task of efficiently creating useful cost-to-go approximations is made difficult as the agent does not have access to the underlying transition or reward functions of the environment. In Chapter 3 we present a modification to the state-of-the-art model-free planning algorithm RIW (Bandres, Bonet, and Geffner 2018) that incorporates cost-to-go approximations via simulation. We show cost-to-go approximations are crucial for ability of RIW, and width-based planning algorithms more generally, to efficiently, in terms of search effort, find solutions to SSP problems. However, computing useful cost-to-go approximations via simulation can require large amounts of compute, particularly in settings with computationally expensive simulators. We follow up on this issue in Chapter 4 by introducing a method that learns cost-to-go approximations through the information collected from the environment in previous episodes executed by agent. Ultimately we address **RQ1** in this Part by showing the effective combination of the learnt cost-to-go approximations with a modified RIW planning algorithm. This planning and learning algorithm achieves state-of-the-art results in the Atari-2600 benchmark in terms of compute efficiency.

Chapter 3

Width-based planning augmented with base policies[1]

3.1 Introduction

This Chapter is concerned with model-free lookahead algorithms over simulators for SSP problems. SSPs are introduced in 2.1 and are one of the three basic types of MDP problems along with problems over an infinite horizon with discounted costs and average cost problems (Bertsekas 2017). As explained in 2.7, model-free lookahead algorithms over simulators can autonomously solve a large variety of ASDM problems without requiring a symbolic model of action effects.

We consider the *width-based* family of planning algorithms, introduced by Lipovetzky and Geffner (2012), that prioritise the exploration of *novel* areas of the state space. Two width-based planners, Lipovetzky and Geffner's IW(1) breadth-first search, and the depth-first search Rollout-IW(1) (Bandres, Bonet, and Geffner 2018), are investigated. We first elaborate on the background information provided in 2.7.5 and 2.7.6 for width-based algorithms, in order to formalise width-based algorithms as instances of the *rollout* algorithm (Bertsekas 2017) (2.7.2). We then illustrate the reasons to augment width-based lookaheads with cost estimates, define the width of SSP problems and propose a novel width-based algorithm that estimates costs-to-go by simulating a general base policy. Our experimental study shows that the algorithm compares

[1]This chapter is adapted from the article "Width-Based Lookaheads Augmented with Base Policies for Stochastic Shortest Paths" published in the proceedings of the 11th Workshop on Heuristics and Domain Independent Planning at the International Conference on Automated Planning and Scheduling 2019, pg 37-45.

favourably to the original Rollout-IW(1) algorithm and other state-of-the-art instances of the *rollout* algorithm.

3.1.1 The rollout algorithm

In this Chapter we explore the online planning framework of the rollout algorithm, introduced in 2.7.2, with cost-to-go approximation via simulation, introduced in 2.7.3. We make a number of assumptions about the d-step selective lookahead rollout algorithm to ensure the viability of lookaheads with $d > 1$. We will assume that we have a settable black-box simulator as introduced in 2.2.2 that can *simulate* the system,

$$s_{k+1} = E\Big\{T(s_k, \mu(s_k), \cdot)\Big\}, \quad k = 0, 1, \ldots, N-1 \tag{3.1}$$

under the base policy μ, so we can generate sample system trajectories and corresponding costs consistent with probabilistic data of the problem. Performing the simulation and calculating the rollout control still needs to be possible within the real-time constraints of the application, which is challenging as the number of Q-factors to estimate and minimisations to perform in $Q_l(s_l, a_l) = E\{c(s_l, a_l) + \min_{a \in A(s_{l+1})} Q_{l+1}(s_{l+1}, a)\}$ (Equation 2.7) and $Q_l(s_l, a_l) = E\{H_l(s_l)\}$ (Equation 2.8) is exponential on the average number of controls available per stage and d, the maximum depth of the lookahead. We avoid the blowup of the size of the lookahead by cutting the recursion in Equation 2.7 and replacing the right hand side by that of Equation 2.8. As detailed in the next sections, we will do this when reaching states s_l that are deemed not to be *novel* according to the notion of structural *width* by Lipovetzky and Geffner (2012). This results in a selective strategy alternative to the upper confidence bounds (Auer, Cesa-Bianchi, and Fischer 2002) used in popular instances of MCTS algorithms like Kocsis and Szepesvari's (2006) UCT, that also are instances of the rollout algorithm (Bertsekas 2017).

3.2 Width-based lookaheads

We instantiate the *rollout* algorithm with an l-step, depth-selective *lookahead* policy using *Width-based Search* (Lipovetzky and Geffner 2012). These algorithms focus the lookahead and have good any-time behaviour. For the prioritisation of expanding states, width-based methods select first states with novel valuations of features defined over the states (Lipovetzky, Ramírez, and Geffner 2015; Geffner and Geffner 2015). As introduced in 2.7.5 the most basic width-based search algorithm is $IW(1)$, a plain *breadth-first search*, guaranteed to run in *linear time and space* as it only expands *novel* states. A state s_l is *novel* if and only if it encounters a state variable [2] s^i, whose value $v \in D(s^i)$, where $D(s^i)$ is the domain of variable s^i, has not been seen before in the current search. Note that novel states are independent of the objective function used, as the estimated cost-to-go V is not used to define the novelty of the states. IW(1) has recently been reformulated as a depth-first search algorithm, and has been shown to perform well with respect to learning approaches with almost real-time computation budgets over the Atari games (Bandres, Bonet, and Geffner 2018).

3.2.1 Depth-first width-based rollout

The breadth-first search strategy underlying IW(1) ensures a state variable s^i is seen for the first time in a lookahead through the shortest sequence of control steps, i.e. the shortest path assuming uniform costs $c(s, a)$[3]. On the other hand, *depth-first* rollout algorithms cannot guarantee this property in general. RIW, introduced in 2.7.6, changes the underlying search of IW into a depth-first rollout. In order to ensure that RIW(1) considers a state to be novel *iff* it reaches at least one value of a state variable s^i_l through a shortest path, we need to adapt the definition of novelty. Intuitively, we need to define a set of state features to emulate the property of the breadth-first search strategy. Let $d(s^i, v)$ be the best upper bound known so far on the shortest path to reach each value $v \in D(x^i)$ of a state variable from the root state s_k. Initially $d(s^i, v) = h$ for all state variables, where h is the horizon, i.e. the maximum search depth allowed for the lookahead, denoting no initial knowledge. When a state s_l is generated, $d(s^i, v)$ is set to l for all state variables where $l < d(s^i, v)$.

Since RIW(1) starts each rollout from the current state s_k, in order to prove a state s_l to be novel we have to distinguish between s_l being already in the lookahead tree and s_l being new. If s_l is *new* in the tree, to conclude it is novel, it is sufficient to show that there exists a state variable s^i whose known shortest path value $d(s^i, v) > l$. If s_l is *already* in the tree, we have to prove the state contains at least one state variable value s^i whose shortest path is $l = d(s^i, v)$, i.e. state s_l is still novel and on the shortest path to s^i. Otherwise if the state is non-novel the rollout is terminated.

[2]In order to use the notion of novelty, we assume state spaces S to be stationary.
[3]This can easily be generalized to non-uniform costs by using Dijkstra's algorithm instead.

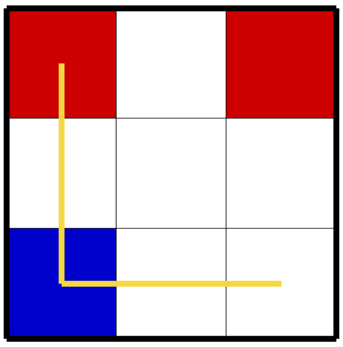


FIGURE 3.1: 3x3 GridWorld problem in which the blue square is the agent's initial position and the red squares are goal locations. The yellow lines represent two action trajectories the agent can perform from the initial state.

In order to ensure the termination of RIW(1), terminal and non-novel states are marked with a *solved* label. The label is back-propagated from a state s_{l+1} to s_l if all the admissible action inputs $a \in A(s_l)$ yield states s_{l+1} already labeled as *solved*. RIW(1) terminates once the root state is labeled as *solved* (Bandres, Bonet, and Geffner 2018). Non-novel states s_l are treated as terminals and their cost-to-go is set to 0. This can induce a bias towards non-novel states rather than true terminal states. In the next section we give an example of the pathological-behaviour of RIW(1) when a state s_l is non-novel and discuss the importance of estimating upper-bounds on the cost-to-go $H_l(s_l)$ instead of assigning termination costs. Both turn out to be essential for RIW(1) over SSPs.

3.3 Width-based lookaheads on SSPs

Despite the successes of *width-based* algorithms on a variety of domains including the Atari-2600 games (Lipovetzky, Ramírez, and Geffner 2015; Bandres, Bonet, and Geffner 2018), the algorithms, as will be shown, have poor performance on SSP problems. We demonstrate below that width-based lookaheads prefer trajectories leading to non-novel states over longer ones that reach a goal. Let us consider a SSP problem with uniform and unitary action costs, shown in Figure 3.1. The task is to navigate to a goal location using the least number of left, right, up or down actions. Any action that would result in the agent moving outside of the grid produces no change in its position. The two features used by the width-based planners are the coordinates *(x), (y)* of the agent's position. Both IW(1) and RIW(1) algorithms, given a sufficient budget, would result in the lookahead represented by yellow lines in Figure 3.1. As expected, both lookaheads contain the shortest paths to make each feature of the problem true. For both IW(1) and RIW(1), we backpropagate the costs found in the lookahead starting from terminal and non-novel states to get the cost-to-go estimates for each action from the current position. In this instance a move down or left from the agent's initial state has no effect, thus immediately producing a non-novel state.When backpropagating values, down and left have an expected cost of 1, which is less than the optimal cost of 2 for up, the action that leads to the top left goal state. This prevents both IW(1) and RIW(1) from ever achieving the goal, as they keep selecting those useless actions. Furthermore if the only goal is the top right square the trajectories produced provide no feedback from a goal position. In this section we propose a method to overcome this pathological-behaviour through cost-to-go approximations.

3.3.1 Novelty, labeling and width of SSPs

Bandres et al. (2018) introduced the algorithm RIW in the context of deterministic transition functions. In this section we discuss its properties in the context of SSPs.

The set of features used to evaluate the novelty of a state is $F = \{(v, i, d) \mid v \in D(s^i)\}$ where $D(s^i)$ is the domain of variable s^i, and d is a possible shortest path distance. Note that the search horizon h is the upper-bound of d. The maximum number of novel states and therefore expanded states is $O(|F|)$, as the maximum number of possible shortest paths for a feature $(v, i, \cdot) \in F$ is h. At worst we can improve the shortest path for $(v, i, \cdot)$ by one step at a time.

The labeling of nodes as introduced by Bandres et al. propagates *solved* labels up the tree once all children of a node have been generated and marked as *solved*. In the deterministic setting, the number of children a node of state s_l has is equal to $|A(s_l)|$, thus ensuring the number of rollouts from the initial state in RIW(1) is at most $O(|F| \times b)$, where $b = max_{s_l}|A(s_l)|$ is the maximal branching factor, the maximum number of control inputs admitted by a state. For the SSP setting the number of children a node can have is $\sum_{a \in A(s_l)} \beta_{s_l}(a)$, where $\beta_{s_l}(a)$ is the number of possible unique resulting states s_{l+1} from applying action a from state s_l. However in the model-free SSP setting β_{s_l} is unknown, so we introduce a new labelling strategy that approximates β_{s_l} with λ which we call RIW(1)-λ.

Definition 3.1. ***(λ-labelling).*** RIW(1)-λ back-propagates a *solved* label to a state s_l iff 1) all admissible control inputs $a \in A(s_l)$ that have been applied for the state s_l result in states labelled as *solved*, and 2) the tree contains at least λ resulting states for each control input $a \in A(s_l)$ applied.

If for any state s_l in the lookahead tree and any $a \in A(s_l)$ $\lambda > \beta(s_l, a)$, RIW(1)-λ will terminate only when the computational budget is exhausted. Note that for the deterministic setting using $\lambda = 1$ will produce the same labelling behaviour as Bandres et al.'s definition. Furthermore, we can reconcile the notion of *width* over classical planning problems (Lipovetzky and Geffner 2012) with SSPs.

Definition 3.2. ***(Width 1 terminal* t *and trajectory* η*).*** A width 1 terminal state $\mathbf{t}$ is a state such that there exists a width 1 trajectory $\eta = s_0, a_0, \ldots, a_{n-1}, s_n$ for $n \leq h$, where h is the search horizon and $s_n = \mathbf{t}$, such that for each s_j in the trajectory η the following three properties are true. First, the prefix $s_0, a_0, \ldots, a_{j-1}, s_j$ reaches at least one feature $f_j = (v, i, d) \in F$ where all $(v, i, d') \in F$ for $d' < d$ are unreachable, i.e., d is the length of the shortest path to reach value v in x^i. Second, any shortest path to f_j can be extended with one additional step to become a shortest path for a feature f_{j+1} complying with the first property in state s_{j+1}. Third, the shortest path for f_n is also a shortest path for $\mathbf{t}$.

Definition 3.3. ***(Width 1 tuple set*** W^1***).*** The set W^1 contains all the state, action, resulting state tuples, (s_l, a_l, s_{1+1}), along all width 1 η trajectories as described in Definition 3.2.

Definition 3.4. ***(Escape probability*** ϵ***).*** The maximum probability of a *width 1 trajectory escape*, that is, for any state action pair, s_l, a_l in a tuple $(s_l, a_l, \cdot) \in W^1$ having successor s_{l+1} such that $(s_l, a_l, s_{l+1}) \notin W^1$, is $\epsilon = \max_{(s_l, a_l, \cdot) \in W^1} (1 - \sum_{(s_l, a_l, s_{l+1}) \in W^1} T(s_l, a_l, s_{l+1}))$.

Theorem 1. Let T^1 be a non-empty set of width 1 terminals. If $\lambda \geq 1$, the probability ρ of RIW(1)-λ finding a width 1 terminal $\mathbf{t} \in T^1$ is $\rho \geq (1 - \epsilon^\lambda)^h$, where h is the search horizon.

Proof sketch. Given a problem with a non-empty set of T^1 width 1 terminals states, by definition for RIW(1)-λ to not find a state $\mathbf{t} \in T^1$, every width 1 trajectory η requires a *solved* label to be backpropagated to a state s_l along the trajectory before reaching the state $\mathbf{t} \in T^1$. For this backpropagation to occur all admissible controls $a \in A(s_l)$ need to be applied, each reaching at least λ different successor states, s_{l+1}, that have been marked as *solved*. Given that we need to apply each $a \in A(s_l) \geq \lambda$ times there is a $\geq 1 - \epsilon^\lambda$ chance of the resulting state s_{l+1} being $(s_l, a_l, s_{l+1}) \in W^1$, along a η trajectory to $\mathbf{t} \in T^1$. Therefore, the probability of RIW(1)-λ reaching a state $\mathbf{t} \in T^1 \geq (1 - \epsilon^\lambda)^h$, since the maximum length of a trajectory to a state in T^1 is h. □

For problems where every tuple $(s_l, a_l, s_{l+1}) \in W^1$ has a high $T(s_l, a_l, s_{l+1})$, ϵ will be small. In this case, given a reasonable h value, the probability of RIW(1)-λ reaching a width 1 terminal will also be reasonable. We discuss next two examples from the literature where the latter can be observed. The first example is self-loop MDPs (Keyder and Geffner 2008) where the probability $T(s, a, s')$ is non-zero for at most one $s' \neq s$. A width one terminal will only be found on trajectories consisting of the $s' \neq s$ transitions. Therefore the lower the probability for $T(s, a, s)$ the higher the lower bound of the probability of RIW(1)-λ finding a width 1 terminal is. The second example is the Atari-2600 games with the so-called sticky actions as introduced by Machado et al. (2018), that modify the games to be stochastic by having a 0.25 chance of repeating the action used for the previous frame instead of the actual action.

3.3.2 RIW(1)-λ with cost-to-go approximation

MCTS algorithms aim at combining lookaheads with stochastic simulations of policies and trading off computational economy with a small risk of degrading performance. We add a new method to the MCTS family, by combining the multi-step, width-based lookahead strategy discussed in the previous section with simulation-based cost-to-go approximations subject to a *computational budget* that limits the number of states visited by both the lookahead and base policy simulation. We use as the base policy a *random walk*, a stochastic policy that assigns the same probability to each of the controls a admissible for state s, and is generally regarded as the default choice when no domain knowledge is readily available. A rolling horizon h is set for the rollout algorithm that follows from combining the RIW(1)-λ lookahead with the simulation of random walks from non-novel states. The maximal length of the latter is set to $h - l$, where l is the depth of the non-novel state. While this can result in trajectories sometimes falling short from a terminal, it avoids extremely long trajectories that eat into the computational budget, and allows sampling other potentially useful non-novel states s_l. Both simulations and the unrolling of the lookahead are interrupted if the computational budget is exhausted.

3.4 Experimental study

3.4.1 Domains

For evaluation we use the SSP benchmark environments introduced in 2.5.1, which include a number of GridWorld (Sutton and Barto 2018) domains, including the stochastic CTP (Papadimitriou and Yannakakis 1991), and John Langford's RL "Acid" domains[4]. Additionally, we present results for the Atari-2600 game Skiing, which is a SSP problem. We use the OpenAI gym's (Brockman et al. 2016) interface of the ALE (Bellemare et al. 2013) and use the slalom game mode of Skiing. In the slalom mode the aim is to ski down a mountain as fast as possible while going through gates. Once at the finish line, a 5 second time penalty is applied for each gate that is missed. The reward values, provided by ALE are the negative value of the time taken plus any time penalties in centiseconds. We use the environment settings as described by Machado et al. (2018) with a frame skip of 5 and sticky actions as described in 3.3.1. The state representation, and therefore features for RIW(1)-λ, is the pixel values of the current gray scaled screen at full 180×210 resolution.

Finally we provide results for the 2018 IPC (introduced in 2.5.4) domains for the probabilistic planning track.

3.4.2 Methodology

We evaluate the depth-first width-based *rollout* algorithm, RIW(1)-λ, with and without being augmented using base policies. $\lambda = 1$ is used for the label back-propagation. We did not observe statistically significant changes with $\lambda = \infty$.

Two additional rollout algorithms are also considered, the one-step *rollout* algorithm, RTDP (Barto, Bradtke, and Singh 1995) and the multi-step, selective, *regret* minimisation, *rollout* algorithm UCT (Kocsis and Szepevari 2006). The exploration parameter of UCT is set to 1.0. For the GridWorld domains that use obstacles we also report the algorithms using Manhattan distances as a heuristic.

Each method is evaluated at different levels of complexity by varying the number of states of the domains, and the *simulation budget*, which is the simulator calls allowed at each time step. For

[4]https://github.com/JohnLangford/RL_acid

each algorithm and domain setting we evaluate performance over 10 different initial states with 20 episodes per initial state, equalling a total of 200 episodes. The values reported here for each algorithm and domain are the average and 95% confidence interval of the costs across the 200 episodes. Each episode was run using a single AMD Opteron 63xx class CPU @ 1.8 GHz, with an approximate run time of 0.75 seconds per 1,000 simulator calls across the different algorithm and domain settings.

The Atari-2600 game Skiing has a maximum episode length of 18,000 frames, with a frame skip of 5 the maximum episode length equals 3,600 actions. The Atari simulator is around an order of magnitude slower than the GridWorld simulator with an approximate run time of 1 second per 100 simulator calls. Therefore, for the evaluation of Skiing we use a simulator budget of 100 and partial caching as described by Bandres et al. (2018), in that we cache simulator state-action transitions, thus assuming determinism, but clear the cached transitions when executing an action in the environment. However, the lookahead tree itself is not cleared when executing an action in the environment as is done for the other domains trialed. Using a simulation-based cost-to-go approximation is infeasible with a simulator budget of 100 and maximum episode length of 3,600 actions. Therefore we report the algorithms using a heuristic, which is the number of gates missed or remaining times the time penalty of 500 centiseconds.

For comparison of RIW(1) with model-based planners, we use the latest International Probabilistic Planning Competition (IPPC) benchmarks. The baselines this time are, the MCTS model-based planner Prost-DD (Geißer and Speck 2018), the winner of the latest competition, and again, RIW(1) without cost-to-go approximations. We use the library from the IPC competitor A2C-Plan (Fern et al. 2018), in order to integrate RDDL environments into the OpenAI gym ones (Brockman et al. 2016), as required by our *model-free* planner. We use the competition settings, namely, an average computation budget of 2.5 seconds per action.

All experiments were run within the OpenAI gym framework (Brockman et al. 2016) and the code used for the algorithms and domains is available through GitHub [5].

3.4.3 Results

The different H functions reported here are $H_{\text{NA}} = 0$, the random policy H_{Rnd}, and the Manhattan distance H_{Man}. The algorithms were also evaluated using Knuth's algorithm (introduced in 2.7.3)

[5]https://github.com/miquelramirez/width-lookaheads-python

Dim.	Alg.	Heu.	Simulator Budget		
			100	1000	10000
10	1Stp	Rnd.	29.6 ± 2.5	13.5 ± 1.6	7.5 ± 0.9
	UCT	Rnd.	29.0 ± 2.6	17.1 ± 2.0	13.3 ± 1.5
	RIW	NA	39.1 ± 2.8	38.1 ± 2.9	38.4 ± 2.9
		Rnd.	33.7 ± 2.5	**6.9** ± **0.7**	**4.7** ± **0.4**
20	1Stp	Rnd.	89.6 ± 3.7	59.8 ± 5.2	29.6 ± 3.1
	UCT	Rnd.	85.2 ± 4.3	72.7 ± 5.8	45.7 ± 4.4
	RIW	NA	79.8 ± 5.5	79.8 ± 5.5	80.2 ± 5.5
		Rnd.	88.2 ± 3.9	55.3 ± 5.2	**10.5** ± **0.9**
50	1Stp	Rnd.	215.2 ± 11.5	201.8 ± 13.5	177.9 ± 13.5
	UCT	Rnd.	220.4 ± 10.8	199.2 ± 13.5	190.6 ± 13.9
	RIW	NA	200.2 ± 13.8	200.2 ± 13.8	200.2 ± 13.8
		Rnd.	223.2 ± 10.4	199.9 ± 13.6	**145.5** ± **12.9**

TABLE 3.1: Average and 95% confidence interval for the cost on GridWorld with a *stationary* goal. Costs reported are from 200 episodes over 10 different initial states (20 episodes per initial state) of the GridWorld with a square grid with width and length equal to the dimension (Dim.) value. The problem horizon of each problem is 5 times the dimension value.

with a different range of rollouts for the cost-to-go estimate, however, the results are not reported here as they are either statistically indifferent or dominated by the results using H_{Rnd} with a single rollout. Bertsekas (2017) suggests that MCTS algorithms should readily benefit from stronger algorithms to estimate costs-to-go by simulation of stochastic policies. Our experiments shows that if this synergies exists it does not manifest when using off-the-shelf stochastic estimation techniques like the ones discussed by Rubinstein and Kroese (2017).

The results on the stationary goal GridWorld domain shown in Table 3.1 provide a number of insights about the *rollout* algorithms reported. First, RIW(1)-λ benefits from using H_{Rnd} as simulator budgets are increased. On the contrary, with H_{NA}, RIW(1)-λ's performance remains constant across the different budgets. The explanation for this can be found in the example of RIW(1) on SSPs we gave previously with the agent preferring shorter trajectories that drive into the boundaries of the grid. Tables 3.1, 3.2 and 3.3 clearly show that using RIW(1)-λ with H_{Rnd} and the largest budget outperforms all other methods.

For the largest simulator budget on CTP reported in Table 3.4 RIW(1)-λ using H_{Rnd} is dominant and for smaller budgets H_{Man} often dominates the H_{Rnd} methods.

Tables 3.5 and 3.6 show that, on the smaller 10 state domains, RIW(1)-λ with H_{Rnd} is statistically dominant over all other methods on Antishaping and Combolock for the 500 and 1000 simulator budgets. However, for the more complex 50 state domains, the results of all algorithms using H_{Rnd} are statistically indifferent. It can be observed that using H_{Rnd} with RIW(1)-λ does improve its performance compared with H_{NA} across all the domain settings with simulator budgets of 500 and 1000, besides Antishaping with 50 states.

Dim.	Alg.	Heu.	Simulator Budget 100	1000	10000
10	1Stp	Rnd.	17.9 ± 2.1	10.1 ± 1.0	6.8 ± 0.5
	UCT	Rnd.	18.9 ± 2.2	11.0 ± 1.3	10.2 ± 1.1
	RIW	NA	39.8 ± 2.7	38.6 ± 2.8	38.9 ± 2.8
		Rnd.	21.3 ± 2.3	**5.7** ± **0.5**	**4.4** ± **0.3**
20	1Stp	Rnd.	81.5 ± 4.2	45.0 ± 4.4	25.5 ± 2.8
	UCT	Rnd.	81.5 ± 4.3	44.9 ± 4.7	38.6 ± 3.7
	RIW	NA	83.5 ± 4.8	82.6 ± 5.0	82.8 ± 4.9
		Rnd.	83.4 ± 4.2	39.8 ± 4.2	**10.8** ± **0.7**
50	1Stp	Rnd.	230.5 ± 7.8	195.3 ± 11.8	141.5 ± 12.1
	UCT	Rnd.	232.7 ± 7.6	196.7 ± 11.6	175.2 ± 11.9
	RIW	NA	212.9 ± 11.3	215.9 ± 10.8	223.5 ± 9.8
		Rnd.	236.2 ± 6.5	200.4 ± 11.4	**110.6** ± **11.8**

TABLE 3.2: Same experimental setting as Table 3.1 over GridWorld with a *moving* goal.

Dim.	Alg.	Heu.	Simulator Budget 100	1000	10000
10	1Stp	Man.	38.1 ± 2.8	39.0 ± 2.7	38.8 ± 2.7
		Rnd.	43.9 ± 1.9	35.9 ± 2.5	25.3 ± 2.4
	UCT	Man.	37.0 ± 2.9	36.4 ± 2.9	36.4 ± 2.9
		Rnd.	43.8 ± 1.9	38.5 ± 2.5	25.9 ± 1.9
	RIW	Man.	36.4 ± 2.9	36.4 ± 2.9	36.4 ± 2.9
		NA	49.8 ± 0.4	48.8 ± 1.0	49.3 ± 0.8
		Rnd.	44.9 ± 1.7	34.5 ± 2.8	**19.3** ± **2.1**
20	1Stp	Man.	76.7 ± 5.5	77.1 ± 5.4	76.7 ± 5.5
		Rnd.	97.9 ± 1.6	88.0 ± 3.4	62.7 ± 4.8
	UCT	Man.	78.0 ± 5.4	78.4 ± 5.3	73.4 ± 5.7
		Rnd.	98.7 ± 1.2	96.4 ± 1.9	77.2 ± 4.2
	RIW	Man.	79.7 ± 4.9	76.7 ± 5.5	76.7 ± 5.5
		NA	100.0 ± 0.0	100.0 ± 0.0	99.6 ± 0.8
		Rnd.	98.5 ± 1.3	88.0 ± 3.4	**29.3** ± **3.1**

TABLE 3.3: Same settings as Table 3.1 over GridWorld with *fully observable obstacles* and a *stationary* goal.

Dim.	Alg.	Heu.	Simulator Budget 100	1000	10000
10	1Stp	Man.	36.6 ± 2.9	35.0 ± 3.0	35.9 ± 2.9
		Rnd.	40.4 ± 2.1	27.2 ± 2.5	15.4 ± 1.6
	UCT	Man.	28.4 ± 2.9	29.5 ± 3.1	18.5 ± 2.7
		Rnd.	41.5 ± 2.0	36.1 ± 2.3	22.2 ± 2.0
	RIW	Man.	28.1 ± 3.0	28.3 ± 3.0	26.5 ± 3.0
		NA	49.5 ± 0.7	49.1 ± 0.9	49.8 ± 0.4
		Rnd.	43.5 ± 1.9	23.4 ± 2.6	**11.1** ± **1.3**
20	1Stp	Man.	71.8 ± 5.8	74.0 ± 5.7	71.9 ± 5.8
		Rnd.	97.0 ± 1.7	82.3 ± 3.9	49.8 ± 4.6
	UCT	Man.	53.8 ± 5.7	60.9 ± 6.2	38.0 ± 5.5
		Rnd.	98.0 ± 1.3	87.4 ± 4.0	63.0 ± 4.8
	RIW	Man.	53.5 ± 5.6	**44.1** ± **5.5**	44.1 ± 5.6
		NA	100.0 ± 0.0	99.5 ± 1.0	99.5 ± 1.0
		Rnd.	97.6 ± 1.5	79.7 ± 4.4	**20.7** ± **1.9**

TABLE 3.4: Same settings as Table 3.1 over GridWorld with *partially observable* obstacles and a *stationary* goal.

For the Skiing Atari-2600 game results in Table 3.7 H_{Heu} is the heuristic value based on the number of gates missed and remaining as described in the previous section. RIW(1)-λ using H_{Heu} dominates all other methods. Comparing RIW(1)-λ using H_{Heu} results with those reported by Machado et al. (2018), it has similar performance to the DQN algorithm (Mnih et al. 2015) after 100 million frames of training. Since the simulation budget per action we use here is

Number of States	Alg.	Heu.	Simulator Budget 100	500	1000
10	1Stp	Rnd.	0.6 ± 0.1	0.5 ± 0.1	0.5 ± 0.1
	UCT	Rnd.	0.6 ± 0.1	0.5 ± 0.1	0.5 ± 0.1
	RIW	NA	0.7 ± 0.1	0.7 ± 0.1	0.7 ± 0.1
		Rnd.	0.5 ± 0.1	**0.3 ± 0.0**	**0.3 ± 0.0**
50	1Stp	Rnd.	1.7 ± 0.1	1.2 ± 0.1	1.1 ± 0.1
	UCT	Rnd.	1.7 ± 0.1	1.3 ± 0.1	1.2 ± 0.1
	RIW	NA	**1.1 ± 0.0**	1.1 ± 0.0	1.1 ± 0.0
		Rnd.	1.7 ± 0.1	1.3 ± 0.1	1.1 ± 0.1

TABLE 3.5: Average and 95% confidence interval for the cost on Antishaping. Costs reported are from 200 episodes over 10 different initial states (20 episodes per initial state). The problem horizon of each problem is 4 times the number of states.

Number of States	Alg.	Heu.	Simulator Budget 100	500	1000
10	1Stp	Rnd.	23.4 ± 2.5	13.5 ± 2.1	10.4 ± 1.7
	UCT	Rnd.	23.6 ± 2.5	12.7 ± 2.0	9.6 ± 1.6
	RIW	NA	27.4 ± 2.6	27.0 ± 2.6	27.9 ± 2.5
		Rnd.	22.9 ± 2.2	**3.6 ± 0.4**	**3.6 ± 0.4**
50	1Stp	Rnd.	200.0 ± 0.0	196.1 ± 3.8	191.2 ± 5.7
	UCT	Rnd.	199.0 ± 1.9	196.1 ± 3.8	190.2 ± 6.0
	RIW	NA	200.0 ± 0.0	200.0 ± 0.0	199.0 ± 1.9
		Rnd.	199.0 ± 1.9	193.1 ± 5.0	190.2 ± 6.0

TABLE 3.6: Same settings as Table 3.5 over Combolock.

Alg.	Heu.	Simulator Budget 100
1Stp	Heu.	16,524.8 ± 396.1
UCT	Heu.	16,220.5 ± 310.0
RIW	Heu.	**14,222.2 ± 373.9**
	NA.	15,854.0 ± 332.9

TABLE 3.7: Average and 95% confidence interval for the cost on the Atari-2600 Skiing game over 100 episodes.

	Domains AA	CD	CR	EO	M	PL	RF	WP	Sum
PRT-DD	.70	1.00	1.00	.40	.50	.92	.66	1.00	6.18
RIW	.00	.39	.00	.06	.10	.05	.52	.00	1.12
$\text{RIW}_{\text{Rnd.}}$	.14	.04	.03	.90	.00	.16	.89	.59	2.75

TABLE 3.8: 2018 IPC domains, Academic Advising (AA), Chromatic Dice (CD), Cooperative Recon (CR), Earth Observation (EO), Manufacturer (M), Push Your Luck (PL), Red-finned Blue-eye (RF), Wildlife Preserve (WP). Values reported are the average IPC scores over the first 10 instances of each domain. The RIW algorithms had 30 trials for each instance, while Prost-DD (PRT-DD) used 75 trials.

equivalent to 500 frames, and given that the maximum episode duration spans 3,600 actions, RIW(1)-λ achieves the performance in Table 3.7 considering $\leq$1.8 million frames.

The 2018 IPC domains results in Table 3.8 show that RIW(1) with H_{Rnd} improves over RIW(1) with H_{NA} in 6 out of 8 domains. In all these domains RIW(1) with H_{Rnd} outperforms the model-free baseline, RIW(1) with H_{NA}. We also observe that in the Earth Observation (EO) and Red-finned Blue-eye (RF) domains RIW with H_{Rnd} outperforms the competition winner Prost-DD. However, it is worth noting that Prost-DD was outperformed in the EO and RF domains by $\geq$ 50% of algorithms in IPC 2018 competition. Additionally for the EO domain Prost-DD failed to execute on 6 of the 10 instances, yet for the RF domain Prost-DD successfully executed for all instances. As discussed by Geißer et al. (2019), the RF domain has the largest median number of actions, action preconditions and planning horizon.

3.5 Related work

Bertsekas (2017) considers AlphaGo Zero (Silver et al. 2017) to be state-of-the-art in MCTS algorithms. It combines the reasoning over confidence intervals first introduced with UCT (Kocsis and Szepevari 2006) and the classic simulation of base policies (Ginsberg 1999), adding to both supervised learning algorithms to obtain, offline, parametric representations of costs-to-go which are efficient to evaluate. The resulting algorithm achieves super-human performance at the game of Go, long considered too hard for AI agents. Rather than using descriptions of states directly as we do, AlphaZero uses a NN to extract automatically features that describe spatial relations between game pieces. Like us, AlphaZero's lookahead uses a stochastic policy to select what paths to expand, but rather than Q-factors, uses estimated win probabilities to prioritise controls, and *simulates* the opponent strategy via self-play to generate successor states. Given we are interested in real-time solutions with limited computational budgets, the training phase is precluded and AlphaZero reduces to a similar version of the UCT algorithm used in our experiments.

The PROST (Keller and Eyerich 2012) and UCT* (Keller and Helmert 2013) algorithms make a number of improvements over the UCT algorithm and achieve state-of-the-art performance on many probabilistic planning problems. However the improvements made require a model, and therefore are not applicable to our setting, although we still provide a comparison with the PROST-DD planner in Table 3.8.

Trevizan et al. (2017) showed how to compute domain independent heuristics for SSPs that do not rely upon determinising the problem. However the heuristics presented by Trevizan et al., like heuristics that determinise the problem, require a model. Any such heuristics that rely upon a model are not applicable to the model-free simulator-based setting, which allows us to apply the algorithms to domains difficult to model but easy to simulate such as the Atari-2600 games.

3.6 Discussion

MCTS approaches typically combine lookaheads and cost-to-go approximations, along with statistical tests to determine what are the most promising directions and focus their sampling effort. The width-based methods described in this chapter do so too, but in ways which are, at first sight, orthogonal to existing strategies. It remains an area of active research to map out exactly how the width-based methods described in this chapter and Chapter 4, as well as those described elsewhere by Junyent et al. (2019; 2021) too, provide alternatives to the limitations of existing MCTS approaches. Having said this, there is no general theory guiding the design of MCTS algorithms either (Bertsekas 2017). As shown in our experiments, it is important to follow strict protocols that alert a potential lack of statistical significance in results, while relying on a diverse set of benchmarks that can be both easily understood and highlight limitations of existing state-of-the-art methods.

This chapter addressed **RQ1** through exploring a limitation in state-of-the-art width-based search algorithms for online planning. We showed that a simple approximation of costs-to-go via simulation, improves the performance of Rollout IW significantly. We compared too with model–based planners on the challenging IPC benchmarks, again improving often on the original algorithm by Bandres et al. and surprisingly enough, performing comparably to the winner of the competition in one domain. The next Chapter follows up on this work by exploring more alternative methods for cost-to-go-approximation, that do not require calling the simulator, but instead use a training phase to learn cost-to-go-approximations.

Chapter 4

Width-based planning and learning over the Atari-2600 benchmark[1]

In this chapter we introduce and explore a new width-based planning and learning algorithm that we benchmark over the Atari-2600 games. The planning and learning algorithm follows up from Chapter 3 by introducing and evaluating a method for learning cost-to-go approximations that are used within a RIW search. Additionally in this chapter, we will introduce a framework for training a planning and learning method and explore the characteristics of Atari-2600 games that affect the performance of planning and learning algorithms.

4.1 Introduction

The Atari-2600 games provide useful environments for benchmarking autonomous agents due to the diversity of behaviour required across the different games. The two main approaches used by autonomous agents applied to the Atari-2600 games have been RL methods (Mnih et al. 2015; Liang et al. 2016; Hessel et al. 2018) and Planning methods (Lipovetzky, Ramírez, and Geffner 2015; Bandres, Bonet, and Geffner 2018). The RL approaches have had great success surpassing the performance of human players for many of the Atari-2600 games. However, RL approaches require long training times in order to train the NNs used for policy and value functions. Planning agents do not require training time and instead use a bounded, fixed computational budget to

[1]This chapter is adapted from the article "Width-based Lookaheads with Learnt Base Policies and Heuristics Over the Atari-2600 Benchmark" published in the proceedings of the Advances in Neural Information Processing Systems 2021, vol 34, pg 26536–26547.

decide which action to take at each time step of the game. The budget allowed for planning for each action is set as part of the experimental setting and can be set in such a way that the agent can play a game in real-time. Through the ALE interface, the agent is not provided a description of the transition or reward functions as is the case of models described through languages such as the PDDL (Haslum et al. 2019). Instead, planning agents applied to the Atari-2600 games are required to work with a simulator, treating the environment's transition and reward functions as a black-box (Lipovetzky, Ramírez, and Geffner 2015).

Width-based planning agents, introduced in 2.7.5 and 2.7.6, have been shown to be particularly successful on the Atari-2600 games when compared to other planning agents (Lipovetzky, Ramírez, and Geffner 2015; Bandres, Bonet, and Geffner 2018). Width-based planners prioritise search effort on states deemed to be novel. The novelty of a state can be defined in a number of ways. Previously, novelty tests have been obtained from the RAM of the game (Lipovetzky, Ramírez, and Geffner 2015), handcrafted features computed from screen pixels (Bandres, Bonet, and Geffner 2018) and learnt features extracted from the screen pixels through a NN (Junyent, Jonsson, and Gomez 2019; Dittadi, Drachmann, and Bolander 2021; Junyent, Gómez, and Jonsson 2021). In this chapter we consider planners with a novelty measure that does not require extensive feature engineering, or the internal state of the simulator, but is instead defined directly over the values of screen pixels.

Recent approaches have combined the RL and planning methods into single agents that are applied to the Atari-2600 games (Junyent, Jonsson, and Gomez 2019; Schrittwieser et al. 2020; Junyent, Gómez, and Jonsson 2021). As introduced in 2.9.4 Junyent et al. (2019) combined a width-based planner with a learnt policy defined over a NN in order to guide the planner to promising areas of the search space. The learnt NN was also used to extract features from which the novelty of states were defined over. In this chapter we introduce new width-based planning and learning methods that learn both policy and value networks using a methodical learning schedule.

Through analysing previous width-based methods we construct and benchmark new width-based approaches for the Atari-2600 games. We also classify the Atari-2600 games according to their particular characteristics. The resulting game taxonomy helps us to gain insight into the performance of the algorithms we propose and benchmark. The Chapter contributions are: (1) an analysis of the previous width-based planning methods that have been applied to the Atari-2600 games, (2) introducing new width-based planning and learning approaches for playing the Atari-2600 games, (3) defining a methodical learning schedule for planning and learning methods, and (4) identifying characteristics of the Atari-2600 games that influence the performance of different planning approaches.

4.2 Related work

The MuZero algorithm introduced by Schrittwieser et al. (2020) and discussed in 2.9.3, achieved state-of-the-art performance in the Atari-2600 games when compared to existing model-free RL algorithm performances. Similar to MuZero we also explore using learnt value and policy networks within a lookahead but consider width-based methods as opposed to MCTS. MuZero's experimental setting is different to the one considered in this chapter as we require access to a simulator in the planning phase and use significantly less computing power.

Width-based planning methods, which are introduced in 2.7.5 and 2.7.6, have had particular success over other planning agents when applied to the Atari-2600 games. Width-based planners were first benchmarked over the Atari-2600 games by Lipovetzky et al. (2015), where they applied IW(1) over the RAM values of the game state as features. Lipovetzky et al., and subsequent works that use IW(1) (Shleyfman, Tuisov, and Domshlak 2016; Jinnai and Fukunaga 2017), show that it outperforms breadth-first search and UCT (Kocsis and Szepevari 2006) planners. Bandres et al. (2018) later introduced the RIW planner, discussed in 2.7.6, and showed it outperformed IW(1) greatly when almost real-time budgets for planning were applied over the Atari-2600 games. Most recently as discussed in 2.9.4, Junyent et al. (2019; 2021) introduced the width-based planning and learning methods, π-IW(1), π-IW(1)+ and π-HIW(n, 1), with π-HIW(n, 1) outperforming all the previous planning methods over the Atari-2600 games.

4.3 Width-based planning and learning for the Atari-2600 games

In this section we step through different design considerations when constructing a width-based planning and learning algorithm. We compare the design decisions made by previous works and propose new algorithms to test over the Atari-2600 games.

4.3.1 Novelty definitions: classic and depth-based

The novelty definition of a width-based lookahead dictates which states to prune. In Algorithm 1 the novelty definition determines the output of the `is_novel` function. We refer to IW(1)'s novelty definition as discussed in 3.2 as the "Classic" definition and define it as,

Definition 4.1 ("Classic" Novelty)**.** Given a feature set $F = \{f_1, \ldots, f_i, \ldots, f_k\}$ s.t. $f_i : S \rightarrow \{\top, \bot\}$, and a lookahead L= (N, C, s_r), a node n is novel, if n contains the first state generated s.t. $f(n^s) = \top$ for some $f \in F$, that is, $\forall n' \in N$, $f(n'^s) = \bot$ and $n' \neq n$.

The "Depth" novelty definition introduced for RIW(1) by Bandres et al. (2018) and also discussed in 3.2, is,

Definition 4.2 ("Depth" Novelty)**.** Given a lookahead L= (N, C, s_r), a newly generated node, $n \notin N$, reached after doing $d(n)$ actions from s_r, is novel, if $f(n^s) = \top$ for some $f \in F$, and $\forall n' \in N$, such that $d(n') \leq d(n)$, $f(n'^s) = \bot$.

Note that the effect of the "Classic" pruning definition can change depending on the type of search algorithm used. For example, unlike BFS, DFS using the "Classic" definition may prune states that are along a shorter path to a feature that has previously been generated in the search. Later we show that in a width-based planning and learning algorithm based upon the RIW(1) algorithm the original "Classic" novelty is competitive and can sometimes outperform the depth-based one over the Atari-2600 games. In what follows, we refer to Bandres et al.'s original configuration of RIW(1) as RIW$_D$ and refer to RIW(1) where one replaces the "Depth" novelty definition with the "Classic" one as RIW$_C$.

4.3.2 Features for novelty from graphical game outputs

Width-based methods require a feature set F to be defined over the observable state-space S. There are two types of observations that can be used for the Atari-2600 games, the internal states of the Atari-2600 machine (the RAM), and the colours of screen pixels. Either of these enable features to be defined, as arbitrary Boolean functions over the observable variables. For the internal state observables we have b × x variables, where b is the size of the Atari memory word (8 bits) and x is the size of the physical RAM given by the number of distinct memory addresses (128 addresses). There are c × w × h screen observable variables, where c is 128, w is 160, and h is 210, corresponding to the colour depth, and the number of screen pixels along the horizontal and vertical directions.

When RIW(1) was introduced (Bandres, Bonet, and Geffner 2018), Bandres et al. stated that features capturing "meaningful structure" would yield better results than using raw features. Hence, Bandres et al. mapped the observable screen variables into the feature set B-PROST, first proposed by Liang et al. (2016). The B-PROST feature set attempts to capture temporal and spatial relationships between the past and present screen pixel values. In order to compute the set of B-PROST features there are a number of steps required. First a set of basic features needs to be computed through dividing the screen into 16×14 tiles comprised of 10×15 pixels. For each tile, (w, h), where $w \in \{1, \ldots, 16\}$ and $h \in \{1, \ldots, 14\}$, there are K features where K is equal to the colour depth of the Atari-2600 pixels (128). The basic feature in the B-PROST set is $f_{w,h,c}$, where $c \in \{1, \ldots, K\}$ is true if the tile (w, h) contains at least one pixel with the colour value c. A second tier of features, the Basic Pairwise Relative Offsets in Space (B-PROS) set, is computed from the basic ones. A B-PROS feature $f_{c_1,c_2,i,j}$, is true if $f_{w,h,c_1} \wedge f_{w+i,h+j,c_2}$ for any w, h. Finally, a third tier of features, the Basic Pairwise Relative Offsets in Time (B-PROT) set, are computed. A B-PROT feature considers the current screen's tiles (w, h) and the previous game screen's tile (w', h') so that a feature $f^t_{c_1,c_2,i,j}$ is true if $f_{w,h,c_1} \wedge f_{w'+i,h'+j,c_2}$ for any w, h where $w' = w$ and $h' = h$. The B-PROST set is the union of basic, B-PROS, and B-PROT feature sets.

The feature set can also be defined dynamically through a NN (Junyent, Jonsson, and Gomez 2019). π-IW, π-IW(1)+ and the lower level planner of π-HIW(n, 1) use a feature set that is defined as the output values of the rectified linear units from the last hidden layer of the policy NN treating zero values as $\perp$ and positive as $\top$. The policy NN input are the last four screens, processed to map colours to a suitably defined grayscale, and down sampled to a size of 84×84. While the policy network is being trained the features extracted through it will also change. This is similar to Dittadi et al. (2021), who use Variational Autoencoders (VAE) to learn a set of features from the Atari game screen using a training set of game screens created from a RIW(1) execution using B-PROST. RIW(1) using the VAE features was shown to outperform RIW(1) using the B-PROST features.

While width-based planning methods using both the BPROST and NN extracted feature sets have been shown to perform well over the Atari games, previous width-based methods have not tested simpler feature sets defined directly over the screen pixel values. With the motivation of presenting a simpler width-based algorithm, we define our feature set directly over the values of the current down sampled 84×84, 8-bit grayscaled, observable screen variables. Each feature is defined as $f_{i,j,c}$ and is true if the downsampled pixel (i, j) has the grayscaled colour c, where $i, j \in \{1, \ldots, 84\}$ and $c \in \{1, \ldots, 256\}$. Despite using a simpler feature set than previous work, in the next section we show that our algorithm outperforms the methods that use dynamically defined NN based features.

Algorithm 2 Novelty guided Critical Path Learning (N-CPL)

```
   // Perform $K$ training iterations
8  for $i = 0, \ldots, K$ do
9      $\mathcal{T}^i \leftarrow \emptyset, E^i \leftarrow \emptyset$ // Iteration's critical path transitions and episode rewards
10     while $\neg$ train_interval_exhausted() do
11         $s \leftarrow s_0, R \leftarrow 0, L \leftarrow$ initialise_lookahead($s_0$)
            while $\neg$ is_terminal($s$) do
12             $L \leftarrow$RIW($L, \pi_b$) // Algorithm 1 in Chapter 2
13             $a, r, s', L \leftarrow$ select_next_transition($L, V^t$) // Selected using Def. 2.14
14             $\mathcal{T}^i \leftarrow \mathcal{T}^i \cup (s, a, r, s'), s \leftarrow s', R \leftarrow R + r$
15         end
16         $E^i \leftarrow E^i \cup R$ // Episode rewards from current iteration
17     end
       // Train and update network parameters according to learning schedule
18     $\pi_b, V^t \leftarrow$ update_network_parameters($\mathcal{T}^i, E^i$)
19 end
```

4.3.3 Learning base policies and termination costs

AlphaGo (Silver et al. 2016) and π-IW (Junyent, Jonsson, and Gomez 2019) showed the power of using a learnt base policy defined through a NN in order to guide a lookahead search. Similarly, AlphaGo and π-IW(1)+ (Junyent, Gómez, and Jonsson 2021) also use a learnt value function defined through a NN. π-IW(1)+ used its learnt value function to modify the definition of $V(n)$ (Definition 2.14) allowing the rewards received from the transitions in the lookahead to sometimes be ignored in preference of the value network's valuation.

We propose a new algorithm based on Algorithm 1, Novelty guided Critical Path Learning or N-CPL for short, that incorporates both a learnt policy and value function network. An outline of N-CPL is shown in Algorithm 2. Like π-IW does, N-CPL defines the base policy used by Algorithm 1 to be a policy network. Besides that, N-CPL uses a cost-to-go approximation which we implement with a NN as a value function, which is evaluated at the non-terminal leaf nodes of the lookahead. As shown in Chapter 3 using cost-go-approximations significantly improve the performance of width-based lookaheads over SSPs, but rather than using simulations to obtain the cost-to-go estimates, we rely on a learnt heuristic function. That is, instead of assigning termination costs, V^T, of 0 as done by previous width-based methods, if the state is not terminal the valuation of the learnt value function network is used. We note that unlike π-IW(1)+ we do not use the learnt value function to modify the $V(s)$ definition as defined in Definition 2.14.

4.3.4 Learning from critical paths

The policy network of N-CPL uses the state action pairs, (s_j, a_j) for $j = 0, \ldots, H-1$, of previous episodes performed by N-CPL with NN parameters, $<\theta_i^\pi, \theta_i^V>$, in order to train new NN parameters θ_{i+1}^π. This is similar to how π-IW(1) trains its policy function, except for the fact that π-IW(1) uses the Q values within the lookahead tree. If multiple actions in π-IW's lookahead have the same Q value for a given state, instead of the training vector assigning a probability of one to the executed action, π-IW uniformly distributes the probability across the actions with the same Q values. We have taken the simpler approach of just using 1-hot encodings for the single selected action along the critical path (Definition 2.15) of the N-CPL algorithm. Curating the training data set in this way also means N-CPL does not need access to the internal data structures of the planning agent itself but instead can externally observe any agent interacting with the environment in order to acquire the training data.

Influential deep RL algorithms such as DQN (Mnih et al. 2015) which have been applied to the Atari-2600 games rely on evaluating an ϵ-greedy policy defined over the parameters of its network in order to sample transitions and use Q-learning updates on the parameters of the network. We follow this strategy and perform Temporal Differential (TD) (Sutton 1988) learning to train the value network. The selection of the transitions that are used for the TD learning determines what the value function is estimating. That is, TD's task is to estimate the expected accumulated rewards from the given state following the policy which underlies the transitions that it is trained on. In N-CPL the transitions within the lookahead follow the base policy, π_b which is being learnt by aiming to mimic the policy induced from the N-CPL lookahead, $\pi_{\text{N-CPL}}$. The N-CPL lookahead through selecting actions according to Definition 2.14 can be seen as a policy improvement operator over π_b and hence the execution of $\pi_{\text{N-CPL}}$ is not necessarily equivalent to the π_b. Therefore it does not make sense to approximate the expected accumulated reward of the lookahead with the expected accumulated reward of π_b. Instead, as shown in Line 8 of Algorithm 2, N-CPL trains only the transitions on the critical path of the lookahead

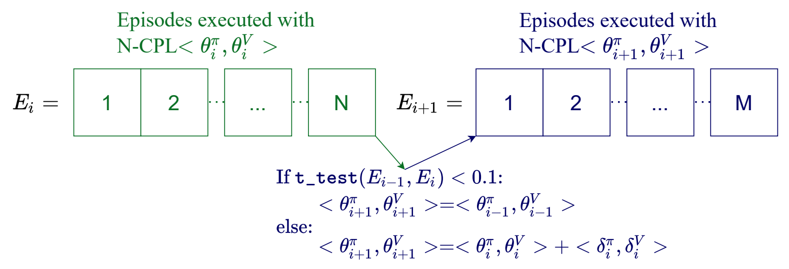

Episodes executed with
N-CPL$< \theta_i^\pi, \theta_i^V >$
Episodes executed with
N-CPL$< \theta_{i+1}^\pi, \theta_{i+1}^V >$
$E_i =$
1
2
...
N
$E_{i+1} =$
1
2
...
M
If t_test$(E_{i-1}, E_i) < 0.1$:
$< \theta_{i+1}^\pi, \theta_{i+1}^V > = < \theta_{i-1}^\pi, \theta_{i-1}^V >$
else:
$< \theta_{i+1}^\pi, \theta_{i+1}^V > = < \theta_i^\pi, \theta_i^V > + < \delta_i^\pi, \delta_i^V >$

FIGURE 4.1: Illustration of the schedule for the network parameter updates, $<\theta^\pi, \theta^V>$, of N-CPL. The E arrays contain episodes executed by N-CPL and each episode in the array is represented by a square in the diagram. The `t_test` function returns the p-value for Welch's t-test (Welch 1947) of the episode rewards executed with the old $i-1$ parameters being better than the new ones. δ_i^π and δ_i^V are real vectors of the same dimensions as θ_i^π and θ_i^V respectively. δ_i^π and δ_i^V are functions over E_i.

(Definition 2.15), that is, the transitions selected by $\pi_{\text{N-CPL}}$. This results in the value function approximating the expected accumulated reward of executing $\pi_{\text{N-CPL}}$ from a given state.

4.3.5 Adding a learning schedule

The previous width-based planning and learning methods continuously learn and update their policy and value networks, while a key mechanism of the Alpha-Zero algorithm (Silver et al. 2017) is the use of a learning schedule. Alpha-Zero evaluates each new set of network parameters θ' that are trained against the current set of network parameters θ to ensure θ' improves AlphaZero's performance. Here we introduce a general learning schedule mechanism that is applicable to sequentially executed and trained planning and learning methods applied to single-player domains, that N-CPL uses for updating its network parameters for its policy network, θ^π and value function network, θ^V. As illustrated in Figure 4.1, the learning schedule determines whether network parameter updates $<\theta_i^\pi, \theta_i^V>$ can be accepted or if the $<\theta_{i-1}^\pi, \theta_{i-1}^V>$ parameters are kept, by evaluating their performance when used within N-CPL. This test is implemented in the `update_network_parameters` function shown in Algorithm 2. For the test N-CPL performs a Welch's t-test (Welch 1947) on its performance with the new i parameters vs. the old $i-1$ parameters. The update is rejected if the t-test suggests, with a p-value of less than 0.1, that performance could deteriorate if the new parameters were accepted. This training and parameter update schedule allows learning steps to be completed at each time step like done by π-IW (Junyent, Jonsson, and Gomez 2019), except that the updated parameters are not used for data generation by N-CPL until they have been accepted by the proposed learning schedule.

4.4 Experimental study

We benchmark width-based planning and learning methods with variations of the design decisions explained in the previous section. Here we explain our experimental methodology and provide results across the different algorithms over the Atari-2600 games.

4.4.1 Methodology

Given vast compute resources it would be preferable to conduct a full ablation study over the Atari-2600 benchmark on each element of a width-based planning and learning algorithm discussed in the previous section. However, due to computational constraints we instead select five planners which will provide the most insight. Two of the planners we evaluate are based on RIW(1) without learnt policy or value function networks. One of the planners uses the "Depth" definition of novelty (Definition 4.2) and the other uses the "Classic" definition (Definition 4.1), we name these version RIW$_D$ and RIW$_C$ respectively. Note that RIW$_D$ is as described in Bandres et al.(2018), except that the features are defined directly over the screen's pixel values as discussed in the previous section. The other 2 width-based planners we benchmark are two versions of N-CPL, as previously described. Again we test both the "Classic" and "Depth" novelty definitions (Definitions 4.1, 4.2), and refer to them as N-CPL and N-CPL$_D$ respectively. For the policy networks of N-CPL and N-CPL$_D$ we use the same architecture used by Mnih et al. (2015). The value network uses the same architecture as the policy network except that instead of the output layer being a dense softmax layer with an output for each action, the output layer of the value network is a dense linear layer with a single output value. Additionally we test a version of N-CPL that does not prune for novelty, we refer to as CPL, i.e. for CPL the `is_novel` function in Algorithm 1 always returns true.

We compare RIW$_D$, RIW$_C$, CPL, N-CPL, and N-CPL$_D$ to the π-IW, π-IW(1)+ and π-HIW(n, 1) planners with the results as given by Junyent et al. (2021). We do not directly compare our results to those given in the original RIW(1) planner as Bandres et al. (2018) use a different experimental design. That is, Bandres et al. and previous works such as Lipovetzky et al. (2015) benchmarked their width-based planners over the Atari-2600 games all using the full action set of 18 actions per state. We found in the code provided for the π-IW(1) work that it was benchmarked against the games using the minimal action set for each game. Using the minimal action set results in many games having much smaller branching factors, for example, instead of

Breakout having a branching factor of 18, it has a branching factor of just 4. Additionally it is worth noting that the true average branching factor of each game is often much smaller than the minimal action set (Nelson 2021) and learning the minimal action set can help avoid unnecessary simulator interactions (Jinnai and Fukunaga 2017). Due to computational constraints we could not benchmark our algorithms over the games with full, minimal and learnt minimal actions sets. Instead, we decided to benchmark using the minimal action set as Junyent et al. (2021) do.

The results are also not directly compared with those from MuZero due to discrepancies in the evaluation protocols and computing resource requirements. For example, MuZero uses a smaller frameskip for the environment time steps and uses a longer allowed episode length of 108,000 frames compared to the 18,000 frame maximum episode length we impose. While our experiments run on a single vCPU for each trial for both training and evaluation, MuZero required 40 third generation Google Cloud TPUs for each run, 8 for training and 32 for its self-play. Furthermore the results for MuZero on each domain were only made available for a 20 billion frame training budget. However in tables 4.9, and 4.10 we do provide a comparison to the RL algorithms DQN (Mnih et al. 2015) and Rainbow (Hessel et al. 2018) and we note the differences in the evaluation protocols used for each algorithm in Table 4.8.

For the evaluation of each algorithm on each game we run 5 independent trials. Once training has completed we evaluate each trial over 10 episodes. Following previous width-based planning papers (Bandres, Bonet, and Geffner 2018; Junyent, Jonsson, and Gomez 2019; Junyent, Gómez, and Jonsson 2021) we use a frameskip of 15. We keep our experimental settings the same as Junyent et al. (2021) including a training budget of 2×10^7 simulator interactions and allowing 100 simulator interactions at each planning time step, which allows almost real-time planning. Note that previous width-based algorithms have varied in how they apply planning budgets, Lipovetzky et al. enforce a budget of 30,000 simulator interactions with a frameskip of 5, while Bandres et al. enforce time budgets of 0.5 and 32 seconds with a frameskip of 15. We ran 80 independent trials at once over 80 Intel Xeon 2.10GHz processors with 720GB of shared RAM, limiting each trial to run on a single vCPU. The average vCPU run-time per time step needed across both the planning and learning steps were 1.28 and 1.11 seconds for N-CPL, and N-CPL$_D$ respectively, resulting in each trial taking just under 3 days to complete. For RIW$_C$ and RIW$_D$, which do not require any learning steps or evaluation of NNs, the average run-times per step were 0.55 and 0.54 seconds respectively. Note that given Atari operates at 60 frames per second and we use a frameskip of 15 a real-time planner would be required to execute with a run-time of 0.25 seconds per time step.

Lookahed Parameters	
Sim. Interactions Budget	100
Lookahead Horizon	100
Value and Policy Network Parameters	
Batch size	128
Learning Rate	2.50×10^{-4}
Epochs	8
Loss Function for Policy	Categorical crossentropy
Loss Function for Value Function	Huber
Discount factor used in TD Learning	0.99
Time steps between target network updates (for value network)	10,000
Interval size of learning schedule	1×10^6 sim. interactions

TABLE 4.1: N-CPL hyperparameters.

The transitions within the lookahead are cached inline with previous width-based planners (Lipovetzky, Ramírez, and Geffner 2015; Bandres, Bonet, and Geffner 2018; Junyent, Jonsson, and Gomez 2019; Junyent, Gómez, and Jonsson 2021). That is, when the search revisits a transition between two nodes of the lookahead within the same episode, the simulator does not need to be recalled and hence does not affect the simulator budget. Also following previous work, transitions that are cached from previous time steps are not considered by the novelty Definitions 4.1, and 4.2, and hence will never be pruned.

Table 4.1 shows the different hyperparameters of N-CPL along with the selected values used for the experiments. Due to computational restraints we could not tune the hyperparameters of N-CPL. The lookahead horizon is the maximum search depth allowed, that is the maximum number of actions allowed from the root node of the lookahead. For the value and policy network parameters both the batch size and number of epochs used affects the training time. We selected both the batch size and number of epochs such that the time spent training the networks using a single vCPU is around the same time as the planning steps of N-CPL. We used the same learning rate, loss function (for the value function) and discount factor for the TD learning as used by DQN (Mnih et al. 2015). Following Junyent et al.(2019) we use a crossentropy loss function for the policy network. The interval size of the learning schedule dictates the size of the data set used to update the networks and how often to reject or accept parameter updates. The interval size of the learning schedule is illustrated in Figure 4.1 to be of a size of N episodes. Instead of setting the interval size to be a set number of episodes we set it as 1×10^6 simulator interactions.

4.4.2 Results

Table 4.2 summarises the results of N-CPL, N-CPL$_D$, CPL, RIW$_C$, RIW$_D$, π-IW(1), π-IW(1)+ and π-HIW(n, 1), additionally tables 4.3 and 4.4 show the performance of each algorithm on each

	Number of games with higher average score than								
	N-CPL	N-CPL$_D$	CPL	RIW$_C$	RIW$_D$	π-IW	π-IW+	π-HIW	**Total (ave. win %)**
N-CPL		26	35	49	47	32	39	32	**260 (70.1%)**
N-CPL$_D$	26		29	48	46	32	38	30	**249 (67.1%)**
CPL	18	24		43	45	31	39	27	**227 (61.2%)**
RIW$_C$	3	4	10		23	19	18	17	**94 (25.3%)**
RIW$_D$	5	6	8	29		20	18	17	**103 (27.8%)**
π-IW	20	20	22	33	32		30	25	**182 (49.1%)**
π-IW+	14	15	14	35	35	23		23	**159 (42.9%)**
π-HIW	21	23	26	36	36	28	30		**200 (53.9%)**

TABLE 4.2: A pairwise comparison of the width-based planning algorithms over the full benchmark set made up of 53 Atari Games. Numbers represent the number of games an algorithm had a higher average evaluation score over the 5 learning trials than the algorithm it is being compared to.

Atari-2600 game tested. Using the pairwise comparison of the different algorithms across the 53 games it is clear that N-CPL$_D$ and N-CPL are the most performant. Comparing the "Depth" vs. "Classic" novelty definition methods as RIW$_D$ vs. RIW$_C$, the former performs better than the latter. The superiority of the "Depth" over the "Classic" definition of novelty does not follow when using our CPL method. The "Classic" method of N-CPL slightly outperforms the "Depth" method N-CPL$_D$, with Table 4.2 indeed showing that N-CPL is the best performing algorithm overall. Interestingly our CPL method that does not use novelty pruning, still outperforms all previous methods which shows the large contribution learning and using the policy and value function networks, as described in the previous section, has on performance.

To better understand the performance of the algorithms we segment the benchmark set according to a couple of different characteristics. The game characteristics we examine are the *branching factor*, and the *sparseness* of meaningful reward feedback (SMRF). We consider rewards as meaningful when they provide information to a player about how to maximise the accumulated reward of an episode. For a given game, SMRF is determined by executing a random policy and a Real-Time Dynamic Planner (RTDP) (Barto, Bradtke, and Singh 1995) over each of the games. RTDP is an online planner that uses a one step lookahead (Definition 2.13) and an approximation for the termination cost V^t at each of the leaf nodes. For the approximation of $V^t(s')$ we use the accumulated reward from a random policy executed from s' for 10 time steps. We run both the random policy and RTDP for 50 time steps (750 frames). If the results from the RTDP planner are not better than the random policy according to Welch's t-test (Welch 1947) with $p < 0.1$, the game is classified as having SMRFs. It should be noted that this classification does not guarantee that a domain classified as having a SMRF actually has a SMRF, for instance the returns from RTDP and a random policy could be similar due to their stochasticity. The results of the random policy vs. RTDP can be seen in Table 4.5. For example, in the game of Pong, RTDP will be able

GAME	RIW$_D$	RIW$_C$	CPL	N-CPL$_D$	N-CPL	π-IW	π-IW+	π-HIW(n,1)
Alien	4,365.20	4,478.40	6,209.00	**7,640.60**	6,943.40	3,969.78	2,585.77	4,609.18
Amidar	1,014.84	897.00	1,433.68	**2,404.60**	2,118.32	950.45	374.20	1,076.17
Assault	764.80	768.00	**3,222.98**	3,185.44	3,079.92	1,574.91	922.30	2,344.28
Asterix	52,940.00	54,090.00	39,860.00	48,364.00	48,226.00	**346,409.11**	247,063.36	90,017.25
Asteroids	1,480.20	1,397.20	6,145.20	9,000.80	**9,152.80**	1,368.55	1,490.87	990.95
Atlantis	48,930.00	46,464.00	**173,786.00**	120,650.00	119,636.00	106,212.63	143,177.73	17,539.22
BankHeist	453.46	436.90	336.40	**957.12**	709.00	567.16	256.29	501.68
BattleZone	102,780.00	88,560.00	220,680.00	165,340.00	153,880.00	69,659.40	30,848.95	**309,137.79**
BeamRider	4,124.37	3,521.80	6,164.64	3,743.80	3,560.88	3,313.11	8,428.96	**11,931.41**
Berzerk	600.00	620.00	3,148.00	4,642.20	4,120.60	1,548.23	960.03	**7,417.26**
Bowling	65.38	63.10	**161.06**	101.40	103.24	26.28	78.18	50.09
Boxing	52.44	54.58	80.96	84.30	86.40	**99.88**	88.19	6.81
Breakout	64.34	53.36	197.82	**320.64**	302.04	92.07	107.64	252.88
Centipede	52,685.12	55,495.78	53,608.80	60,157.38	62,654.16	126,488.35	**141,070.19**	80,685.48
ChopperCommand	3,768.00	3,466.00	17,908.00	4,570.00	3,786.00	11,187.44	3,431.74	**70,787.12**
CrazyClimber	40,520.00	39,387.50	78,266.00	90,332.00	91,912.00	**161,192.01**	138,648.58	102,205.99
DemonAttack	8,499.88	8,449.00	10,560.00	10,829.90	10,876.00	26,881.13	**35,022.64**	16,007.64
DoubleDunk	6.72	6.00	19.76	23.76	**23.96**	4.68	-16.80	3.51
Enduro	1.90	1.34	231.44	250.88	220.28	**506.59**	63.83	44.47
FishingDerby	-67.76	-62.46	-23.94	-29.22	-7.62	**8.89**	-28.02	-53.76
Frostbite	280.00	273.80	**9,956.80**	5,255.60	6,508.00	270.00	1,636.51	7,242.60
Gopher	6,311.43	5,990.83	11,181.60	13,326.40	12,539.60	**18,025.91**	7,061.76	15,001.18
Gravitar	1,755.00	1,725.00	**2,382.00**	2,246.00	2,284.00	1,876.80	1,532.33	1,154.01
Hero	17,438.30	17,159.70	29,200.40	34,083.40	36,099.80	**36,443.73**	22,097.39	36,231.21
IceHockey	21.48	21.92	19.62	**29.58**	26.96	-9.66	-4.02	-2.36
Jamesbond	2,378.00	2,485.00	18,666.00	17,572.00	**18,790.00**	43.20	104.91	1,380.13
Kangaroo	1,556.00	1,504.00	5,332.00	**9,222.00**	9,202.00	1,847.46	2,918.98	6,861.57
Krull	2,176.58	2,127.60	5,631.22	4,900.90	4,422.88	8,343.30	**13,014.77**	4,121.81
KungFuMaster	5,668.00	5,886.00	27,288.00	**43,520.00**	42,388.00	41,609.03	24,871.94	20,680.65
MontezumaRevenge	0.00	0.00	2.00	0.00	0.00	0.00	810.49	**5,275.89**
MsPacman	15,697.56	15,729.28	11,922.22	16,150.08	**18,285.18**	14,726.33	5,916.86	4,523.47
NameThisGame	6,247.50	6,145.25	8,287.40	8,017.20	8,339.40	12,734.85	**18,167.55**	9,977.12
Phoenix	4,992.80	4,912.60	6,616.40	8,033.40	**8,777.40**	5,905.12	7,647.67	7,508.63
Pitfall	-44.86	-62.66	-0.48	-1.68	**-0.42**	-214.75	-2.46	-128.82
Pong	-4.52	-4.92	-12.28	6.18	**10.94**	-20.42	2.14	-9.70
PrivateEye	625.72	1,249.68	153.22	120.00	100.00	452.40	1,766.13	**29,548.76**
Qbert	4,499.50	4,426.50	28,182.50	31,625.50	30,618.50	32,529.60	23,337.90	**40,449.72**
RoadRunner	17,326.00	20,580.00	86,650.00	49,828.00	57,212.00	38,764.81	43,813.29	**87,953.53**
Robotank	31.03	31.33	31.04	**37.94**	36.16	15.66	9.68	10.63
Seaquest	1,609.60	1,576.20	3,523.00	2,878.60	3,922.80	**5,916.05**	559.28	867.51
Skiing	-31,013.00	-22,234.22	-20,510.82	-29,080.30	-20,041.24	-19,188.32	**-13,852.04**	-15,417.86
Solaris	3,110.00	3,085.00	**7,741.60**	3,106.40	4,704.00	3,048.78	1,832.93	3,524.69
SpaceInvaders	2,592.40	2,622.50	3,447.40	3,680.40	**4,289.40**	2,694.09	1,622.49	2,946.18
StarGunner	17,510.00	17,597.56	18,340.00	**20,700.00**	20,320.00	1,381.24	1,642.82	1,864.64
Tennis	**1.40**	-0.53	-2.80	1.24	0.00	-23.67	-8.26	-20.00
TimePilot	24,455.00	24,342.50	22,780.00	24,950.00	24,150.00	16,099.92	11,126.86	**34,610.25**
Tutankham	159.40	154.86	184.94	181.48	203.54	**216.67**	181.44	199.06
UpNDown	39,834.00	40,649.00	59,650.80	58,783.00	58,867.20	**107,757.51**	59,497.75	80,991.07
Venture	22.00	32.00	**1,732.00**	1,466.00	1,564.00	0.00	15.68	10.73
VideoPinball	136,531.50	138,748.35	148,995.14	134,489.58	139,799.22	**514,012.51**	387,308.60	184,720.01
WizardOfWor	26,956.25	27,991.11	54,988.00	43,556.00	43,436.00	**76,533.18**	30,383.68	12,027.43
YarsRevenge	59,779.48	60,940.90	133,647.42	142,568.26	135,089.68	102,183.67	64,544.51	**159,496.20**
Zaxxon	9,342.00	9,520.00	28,102.00	**33,268.00**	30,818.00	22,905.73	10,159.01	21,135.58
Total times best	**1**	**0**	**7**	**10**	**8**	**12**	**5**	**10**

TABLE 4.3: Comparing averages directly over 53 Atari-2600 games, as no confidence interval or standard deviation data is provided for the results of π-IW(1), π-IW(1)+, π-HIW(n, 1). The highest average score is highlighted in green. Freeway is excluded from this table as Junyent et al. (2021) do not report the results for it due to its slow simulator time. See Table 4.4 for the Freeway results of RIW$_D$, RIW$_C$, N-CPL$_D$, and N-CPL.

to discover states through its 10 step approximation of $V^t(s')$ where either player has scored. Using information from $V^t(s')$ about which players have scored, RTDP will be able to have a better informed policy than the random policy, so Pong would not be considered a SMRF game. In a game like Skiing, where a skier is required to ski down a mountain and pass through gates on its path down, RTDP will not discover any meaningful rewards in its 10 step rollouts. This is because in Skiing, despite a constant negative reward at each time step, there is no meaningful reward feedback until the skier reaches the bottom of the mountain where a negative reward is applied for each gate the skier missed.

Table 4.6 groups the games according to their branching factor. Comparing Table 4.6 and 4.2 we can see that for games with larger branching factors, the relative performance gap between

GAME	**RIW$_D$**	**RIW$_C$**	**CPL**	**N-CPL$_D$**	**N-CPL**
Alien	4,365.20±471.95	4,478.40±375.51	6,209.00±415.77	7,640.60±615.52	6,943.40±507.28
Amidar	1,014.84±70.76	897.00±70.28	1,433.68±123.49	**2,404.60±69.72**	2,118.32±114.28
Assault	764.80±49.88	768.00±68.03	3,222.98±98.37	3,185.44±123.18	3,079.92±122.33
Asterix	52,940.00 ±1,486.56	54,090.00 ±1,323.40	39,860.00 ±410.25	48,364.00 ±1,844.10	48,226.00 ±1,184.51
Asteroids	1,480.20±75.48	1,397.20±113.97	6,145.20±318.23	9,000.80±143.15	9,152.80±189.24
Atlantis	48,930.00 ±4,017.54	46,464.00 ±3,132.23	**173,786.00 ±2,343.28**	120,650.00 ±892.42	119,636.00 ±830.95
BankHeist	453.46±31.86	436.90±40.51	336.40±25.25	**957.12±105.73**	709.00±76.09
BattleZone	102,780.00 ±22,127.98	88,560.00 ±16,866.49	**220,680.00 ±14,274.79**	165,340.00 ±28,612.76	153,880.00 ±28,305.50
BeamRider	4,124.37±384.99	3,521.80±371.92	**6,164.64±287.17**	3,743.80±466.05	3,560.88±415.25
Berzerk	600.00±32.84	620.00±32.37	3,148.00±386.38	4,642.20±367.59	4,120.60±423.64
Bowling	65.38±1.66	63.10±2.00	**161.06±5.46**	101.40±3.22	103.24±3.32
Boxing	52.44±2.00	54.58±2.86	80.96±2.63	84.30±2.10	86.40±0.97
Breakout	64.34±17.89	53.36±10.91	197.82±33.59	320.64±8.03	302.04±18.31
Centipede	52,685.12 ±2,811.22	55,495.78 ±1,398.86	53,608.80 ±1,567.38	60,157.38 ±1,726.88	**62,654.16 ±1,107.16**
ChopperCommand	3,768.00±634.47	3,466.00±378.06	**17,908.00±1,432.95**	4,570.00±602.78	3,786.00±601.16
CrazyClimber	40,520.00 ±550.63	39,387.50 ±486.25	78,266.00 ±3,572.79	90,332.00 ±2,414.91	91,912.00 ±2,184.99
DemonAttack	8,499.88±252.38	8,449.00±320.73	10,560.00±188.97	10,829.90±274.19	10,876.00±193.47
DoubleDunk	6.72±0.84	6.00±0.94	19.76±1.02	23.76±0.15	**23.96±0.07**
Enduro	1.90±0.57	1.34±0.38	231.44±10.04	**250.88±8.23**	220.28±9.71
FishingDerby	-67.76±1.96	-62.46±2.20	-23.94±4.32	-29.22±4.30	**-7.62±5.99**
Freeway	5.50 ± 0.24	5.52 ± 0.28	28.86±0.45	29.02 ± 0.45	28.96 ± 0.30
Frostbite	280.00±4.56	273.80±3.72	**9,956.80±1,066.62**	5,255.60±389.33	6,508.00±588.79
Gopher	6,311.43±440.94	5,990.83±479.70	11,181.60±72.70	**13,326.40±240.79**	12,539.60±158.58
Gravitar	1,755.00±171.35	1,725.00±192.75	2,382.00±228.46	2,246.00±201.67	2,284.00±231.37
Hero	17,438.30 ±1,011.92	17,159.70 ±949.20	29,200.40 ±1,804.42	34,083.40 ±938.86	**36,099.80 ±553.32**
IceHockey	21.48±0.73	21.92±0.83	19.62±1.09	**29.58±0.90**	26.96±1.03
Jamesbond	2,378.00±1,224.25	2,485.00±1,210.76	18,666.00±598.37	17,572.00±820.54	18,790.00±907.58
Kangaroo	1,556.00±203.70	1,504.00±151.18	5,332.00±727.16	9,222.00±546.79	9,202.00±572.19
Krull	2,176.58±89.61	2,127.60±88.85	**5,631.22±272.50**	4,900.90±141.45	4,422.88±149.38
KungFuMaster	5,668.00 ±425.75	5,886.00 ±521.52	27,288.00 ±1,908.35	43,520.00 ±1,108.67	42,388.00 ±643.13
MontezumaRevenge	0.00±0.00	0.00±0.00	2.00±3.26	0.00±0.00	0.00±0.00
MsPacman	15,697.56 ±1,148.98	15,729.28 ±1,138.48	11,922.22 ±1,044.99	16,150.08 ±1,064.72	**18,285.18 ±703.36**
NameThisGame	6,247.50±98.52	6,145.25±91.49	8,287.40±117.61	8,017.20±129.51	8,339.40±109.81
Phoenix	4,992.80±242.42	4,912.60±262.97	6,616.40±316.44	8,033.40±687.80	8,777.40±776.53
Pitfall	-44.86±22.53	-62.66±26.84	-0.48±0.78	-1.68±1.33	-0.42±0.52
Pong	-4.52±1.49	-4.92±1.28	-12.28±1.29	6.18±1.29	**10.94±1.22**
PrivateEye	625.72±685.72	1,249.68±938.31	153.22±22.74	120.00±9.30	100.00±0.00
Qbert	4,499.50±756.07	4,426.50±688.09	28,182.50±2,855.33	**31,625.50±406.22**	30,618.50±693.66
RoadRunner	17,326.00 ±3,626.95	20,580.00 ±3,760.02	**86,650.00 ±3,065.43**	49,828.00 ±3,244.08	57,212.00 ±4,094.90
Robotank	31.03±1.16	31.33±1.33	31.04±0.63	**37.94±0.73**	36.16±0.84
Seaquest	1,609.60±291.85	1,576.20±281.51	3,523.00±255.00	2,878.60±285.43	**3,922.80±245.47**
Skiing	-31,013.00 ±700.82	-22,234.22 ±872.31	-20,510.82 ±1,199.55	-29,080.30 ±1,034.25	-20,041.24 ±1,139.52
Solaris	3,110.00±232.31	3,085.00±532.56	**7,741.60±825.51**	3,106.40±169.24	4,704.00±615.29
SpaceInvaders	2,592.40±301.45	2,622.50±303.10	3,447.40±269.92	3,680.40±310.85	**4,289.40±301.15**
StarGunner	17,510.00±240.75	17,597.56±397.74	18,340.00±217.12	20,700.00±178.37	20,320.00±335.94
Tennis	1.40±1.53	-0.53±1.52	-2.80±0.93	1.24±1.45	0.00±1.51
TimePilot	24,455.00±615.53	24,342.50±544.82	22,780.00±839.59	24,950.00±528.00	24,150.00±670.57
Tutankham	159.40±4.77	154.86±4.86	184.94±4.59	181.48±3.80	**203.54±4.82**
UpNDown	39,834.00±830.87	40,649.00±822.57	59,650.80±1,078.03	58,783.00±604.10	58,867.20±752.05
Venture	22.00±20.96	32.00±23.00	**1,732.00±49.99**	1,466.00±176.15	1,564.00±149.66
VideoPinball	136,531.50 ±8,459.01	138,748.35 ±8,349.10	148,995.14 ±8,492.72	134,489.58 ±7,483.99	139,799.22 ±7,019.84
WizardOfWor	26,956.25 ±2,793.49	27,991.11 ±2,907.59	**54,988.00 ±1,906.42**	43,556.00 ±4,275.90	43,436.00 ±4,142.91
YarsRevenge	59,779.48 ±2,070.84	60,940.90 ±2,234.00	133,647.42 ±4,698.49	**142,568.26 ±3,934.50**	135,089.68 ±4,813.68
Zaxxon	9,342.00 ±944.44	9,520.00 ±1,221.31	28,102.00 ±1,264.90	**33,268.00 ±1,192.32**	30,818.00 ±1,722.99
Best algorithm (t-test, p <0.1)	0	0	11	9	9

TABLE 4.4: Average scores with 90% confidence intervals over the set of 54 Atari Games. Algorithm scores that are the best according to the Welch's t-test (Welch 1947) using p< 0.1 are highlighted in green.

our N-CPL$_D$ and N-CPL planners and Junyent et al.'s π-IW(1), π-IW(1)+ and π-HIW(n, 1) increases as the branching factor increases. For example N-CPL and N-CPL$_D$ perform better than π-IW(1)+ in 28/33 (84.8%) games and 27/33 (81.8%) games respectively for the games with a branching factor greater or equal to 10. However, for games with a branching factor less than 10, N-CPL and N-CPL$_D$ only perform better than π-IW(1)+ in 11/20 (55%) and 11/21 (55%) games respectively.

GAME	RTDP	Random
Alien	379.50 ± 180.1	135.00 ± 39.4
Amidar	30.90 ± 18.0	8.70 ± 15.8
Assault	139.65 ± 38.3	49.35 ± 27.6
Asterix	625.00 ± 91.5	155.00 ± 70.5
Asteroids	397.50 ± 137.6	74.00 ± 42.0
Atlantis	1350.00 ± 915.7	635.00 ± 979.9
BankHeist	27.50 ± 20.7	3.50 ± 4.8
BattleZone	3350.00 ± 2725.3	450.00 ± 669.0
BeamRider	121.00 ± 30.7	50.60 ± 46.7
Berzerk	324.00 ± 109.4	102.50 ± 68.0
Bowling	0.00 ± 0.0	0.00 ± 0.0
Boxing	12.90 ± 5.8	−2.00 ± 4.1
Breakout	3.10 ± 0.8	1.00 ± 0.7
Centipede	2189.75 ± 226.5	548.85 ± 565.0
ChopperCommand	790.00 ± 260.6	205.00 ± 120.3
CrazyClimber	715.00 ± 127.6	395.00 ± 124.4
DemonAttack	68.50 ± 9.6	29.50 ± 12.4
DoubleDunk	−0.20 ± 1.4	−1.90 ± 1.3
Enduro	0.15 ± 0.5	0.05 ± 0.2
FishingDerby	−1.70 ± 3.1	−9.70 ± 3.0
Freeway	0.00 ± 0.0	0.00 ± 0.0
Frostbite	122.00 ± 21.1	23.50 ± 16.5
Gopher	0.00 ± 0.0	0.00 ± 0.0
Gravitar	177.50 ± 195.2	0.00 ± 0.0
Hero	2206.50 ± 886.9	36.25 ± 41.4
IceHockey	0.80 ± 0.8	−0.40 ± 0.8
Jamesbond	15.00 ± 22.9	5.00 ± 15.0
Kangaroo	100.00 ± 100.0	10.00 ± 43.6
Krull	104.50 ± 38.4	19.50 ± 18.8
KungFuMaster	130.00 ± 145.3	5.00 ± 21.8
MontezumaRevenge	0.00 ± 0.0	0.00 ± 0.0
MsPacman	396.50 ± 193.9	213.00 ± 163.7
NameThisGame	138.00 ± 41.1	44.50 ± 35.3
Phoenix	346.00 ± 93.0	110.00 ± 66.8
Pitfall	0.00 ± 0.0	−1.10 ± 4.8
Pong	−0.65 ± 1.4	−3.30 ± 1.0
PrivateEye	0.00 ± 0.0	4.75 ± 21.9
Qbert	606.25 ± 50.5	77.50 ± 125.5
Riverraid	862.50 ± 210.2	321.50 ± 70.0
RoadRunner	15.00 ± 65.4	5.00 ± 21.8
Robotank	1.05 ± 0.7	0.30 ± 0.5
Seaquest	73.00 ± 26.3	14.00 ± 15.6
Skiing	−1248.00 ± 0.0	−1248.00 ± 0.0
Solaris	0.00 ± 0.0	0.00 ± 0.0
SpaceInvaders	130.75 ± 29.5	41.00 ± 25.6
StarGunner	370.00 ± 95.4	60.00 ± 106.8
Tennis	0.10 ± 0.7	−1.90 ± 0.9
TimePilot	370.00 ± 134.5	105.00 ± 66.9
Tutankham	17.85 ± 11.8	1.95 ± 4.8
UpNDown	1247.50 ± 465.4	272.00 ± 285.5
Venture	0.00 ± 0.0	0.00 ± 0.0
VideoPinball	5026.10 ± 1909.3	521.10 ± 365.4
WizardOfWor	240.00 ± 156.2	30.00 ± 55.7
YarsRevenge	5527.45 ± 2895.4	274.10 ± 315.1
Zaxxon	20.00 ± 60.0	0.00 ± 0.0

TABLE 4.5: Mean with standard deviations of 20 RTDP vs Random episodes for the SMRF test. Bold domains are classified as having SMRF.

Table 4.7 compares the pairwise performance for games classified as SMRF games. Table 4.7 clearly shows that the dominant method for the SMRF games is CPL, that is, the algorithm without novelty pruning. Table 4.7 also shows that the "Classic" novelty (Definition 4.1) methods outperform "Depth" novelty (Definition 4.2). These observations, that contradict previous claims in the literature, required careful analysis. We observed that the "Classic" method prunes states

	Number of games with higher average score than								
	N-CPL	N-CPL$_D$	CPL	RIW$_C$	RIW$_D$	π-IW	π-IW+	π-HIW	**Total (ave. win %)**
N-CPL		14	22	30	29	23	28	22	**168 (72.7%)**
N-CPL$_D$	18		18	31	29	23	27	21	**167 (72.3%)**
CPL	11	15		26	28	23	28	21	**152 (65.8%)**
RIW$_C$	2	1	7		16	15	16	13	**70 (30.3%)**
RIW$_D$	3	3	5	16		16	16	13	**72 (31.2%)**
π-IW	9	9	10	17	16		19	14	**94 (40.7%)**
π-IW+	5	6	5	17	17	14		11	**75 (32.5%)**
π-HIW	11	12	12	20	20	19	22		**116 (50.2%)**

TABLE 4.6: Same as Table 4.2 but for games with a Branching Factor ≥ 10 (33 Games).

	Number of games with higher average score than								
	N-CPL	N-CPL$_D$	CPL	RIW$_C$	RIW$_D$	π-IW	π-IW+	π-HIW	**Total (ave. win %)**
N-CPL		7	5	10	10	7	9	7	**55 (65.5%)**
N-CPL$_D$	4		3	9	9	7	9	6	**47 (56%)**
CPL	7	9		11	11	8	9	7	**62 (73.8%)**
RIW$_C$	1	2	1		6	6	3	4	**23 (27.4%)**
RIW$_D$	1	2	1	5		6	3	4	**22 (26.2%)**
π-IW	4	4	4	5	5		4	3	**29 (34.5%)**
π-IW+	3	3	3	9	9	8		5	**40 (47.6%)**
π-HIW	5	6	5	8	8	9	7		**48 (57.1%)**

TABLE 4.7: Same as Table 4.2 but for SMRF games (12 Games).

Algorithm	**Frameskip**	**Max. ep. length (Frames)**	**Train Budget (Sim. Interactions)**	**Lookahead Budget (Sim. Interactions)**	**Starts**	**Loss of Life signal**
RIW$_D$, RIW$_C$	15	18,000	0	100	-	No
N-CPL$_D$, N-CPL, CPL, π-IW, π-IW+, π-HIW(n,1)	15	18,000	20×10^6	100	-	No
DQN	4	18,000	50×10^6	NA	Rand.no-ops	Yes
Rainbow	4	108,000	50×10^6	NA	Human and Rand. no-ops	Yes

TABLE 4.8: Comparison of experimental settings used for the results of the different algorithms. Note that the train budget includes all the simulator interactions used by the lookahead algorithms even though only a small fraction of simulator interactions are used for training the networks directly.

more aggressively, meaning it is more likely to reach states that are further away from the root node compared with the "Depth" definition. Similarly, as the CPL method does not prune any states due to novelty, CPL's depth-first lookahead trajectories will always reach the lookahead search horizon of 100 time steps at least once, given that the lookahead simulator budget is 100 time steps. This results in CPL on average searching for states that are further away from the root node than any of the novelty pruning methods. By definition, high rewards for SMRF games have a higher probability of being further away than games with dense rewards. Therefore, CPL and the "Classic" novelty methods are more likely to discover the meaningful rewards by searching deeper in the lookahead. Interestingly CPL and N-CPL still outperform, yet are close to π-IW(1)+ and π-HIW(n, 1) on the SMRF games, despite π-IW(1)+ and π-HIW(n, 1) being motivated by such domains.

GAME	Human	DQN	N-CPL
Alien	6,875.00	3,069.00	**6,943.40**
Amidar	1,676.00	739.50	**2,118.32**
Assault	1,496.00	**3,359.00**	3,079.92
Asterix	8,503.00	6,012.00	**48,226.00**
Asteroids	**13,157.00**	1,629.00	9,152.80
Atlantis	29,028.00	85,641.00	**119,636.00**
BankHeist	**734.40**	429.70	709.00
BattleZone	37,800.00	26,300.00	**153,880.00**
BeamRider	5,775.00	**6,846.00**	3,560.88
Bowling	**154.80**	42.40	103.24
Boxing	4.30	71.80	**86.40**
Breakout	31.80	**401.20**	302.04
Centipede	11,963.00	8,309.00	**62,654.16**
ChopperCommand	**9,882.00**	6,687.00	3,786.00
CrazyClimber	35,411.00	**114,103.00**	91,912.00
DemonAttack	3,401.00	9,711.00	**10,876.00**
DoubleDunk	-15.50	-18.10	**23.96**
Enduro	**309.60**	301.80	220.28
FishingDerby	**5.50**	-0.80	-7.62
Freeway	29.60	**30.30**	28.96
Frostbite	4,335.00	328.30	**6,508.00**
Gopher	2,321.00	8,520.00	**12,539.60**
Gravitar	**2,672.00**	306.70	2,284.00
Hero	25,763.00	19,950.00	**36,099.80**
IceHockey	0.90	-1.60	**26.96**
Jamesbond	406.70	576.70	**18,790.00**
Kangaroo	3,035.00	6,740.00	**9,202.00**
Krull	2,395.00	3,805.00	**4,422.88**
KungFuMaster	22,736.00	23,270.00	**42,388.00**
MontezumaRevenge	**4,367.00**	0.00	0.00
MsPacman	15,693.00	2,311.00	**18,285.18**
NameThisGame	4,076.00	7,257.00	**8,339.40**
Pong	9.30	**18.90**	10.94
PrivateEye	**69,571.00**	1,788.00	100.00
Qbert	13,455.00	10,596.00	**30,618.50**
Riverraid	13,513.00	8,316.00	**22,111.20**
RoadRunner	7,845.00	18,257.00	**57,212.00**
Robotank	11.90	**51.60**	36.16
Seaquest	**20,182.00**	5,286.00	3,922.80
SpaceInvaders	1,652.00	1,976.00	**4,289.40**
StarGunner	10,250.00	**57,997.00**	20,320.00
Tennis	-8.90	-2.50	**0.00**
TimePilot	5,925.00	5,947.00	**24,150.00**
Tutankham	167.60	186.70	**203.54**
UpNDown	9,082.00	8,456.00	**58,867.20**
Venture	1,188.00	380.00	**1,564.00**
VideoPinball	17,298.00	42,684.00	**139,799.22**
WizardOfWor	4,757.00	3,393.00	**43,436.00**
Zaxxon	9,173.00	4,977.00	**30,818.00**
# Games >human		23 (47%)	**37 (76%)**
#Games >75% human		27 (55%)	**40 (82%)**
# Games Best	10 (20%)	8 (16%)	**31 (63%)**

TABLE 4.9: Comparison of N-CPL with a Human player's scores and the model-free RL algorithm DQN scores as reported by Mnih et al. (2015). Note that the experimental settings are different between N-CPL and DQN(Mnih et al. 2015), in terms of training budget, frame skips, using no-op starts and loss of life signal, see Table 4.8 for a comparison of experimental settings.

Tables 4.9 and 4.10 compare N-CPL with the RL algorithms of DQN (Mnih et al. 2015) and Rainbow (Hessel et al. 2018) respectively. It is important to note that the experimental settings of the N-CPL, Rainbow and DQN results are very different as shown in 4.8. For example DQN and Rainbow use larger training budgets and provide the agent with a loss of life signal

GAME	Human	Rainbow	N-CPL
Alien	6875	**9491.70**	6943.40
Amidar	1676	**5131.20**	2118.32
Assault	1496	**14198.50**	3079.92
Asterix	8503	**428200.30**	48226.00
Asteroids	**13157**	2712.80	9152.80
Atlantis	29028	**826659.50**	119636.00
BankHeist	734.4	**1358.00**	709.00
BattleZone	37800	62010.00	**153880.00**
BeamRider	5775	**16850.20**	3560.88
Bowling	**154.8**	30.00	103.24
Boxing	4.3	**99.60**	86.40
Breakout	31.8	**417.50**	302.04
Centipede	11963	8167.30	**62654.16**
ChopperCommand	9882	**16654.00**	3786.00
CrazyClimber	35411	**168788.50**	91912.00
DemonAttack	3401	**111185.20**	10876.00
DoubleDunk	-15.5	-0.30	**23.96**
Enduro	309.6	**2125.90**	220.28
FishingDerby	5.5	**31.30**	-7.62
Freeway	29.6	**34.00**	28.96
Frostbite	4335	**9590.50**	6508.00
Gopher	2321	**70354.60**	12539.60
Gravitar	**2672**	1419.30	2284.00
Hero	25763	**55887.40**	36099.80
IceHockey	0.9	1.10	**26.96**
Kangaroo	3035	**14637.50**	9202.00
Krull	2395	**8741.50**	4422.88
KungFuMaster	22736	**52181.00**	42388.00
MontezumaRevenge	**4367**	384.00	0.00
MsPacman	15693	5380.40	**18285.18**
NameThisGame	4076	**13136.00**	8339.40
Pong	9.3	**20.90**	10.94
PrivateEye	**69571**	4234.00	100.00
Qbert	16455	**33817.50**	30618.50
RoadRunner	7845	**62041.00**	57212.00
Robotank	11.9	**61.40**	36.16
Seaquest	**20182**	15898.90	3922.80
SpaceInvaders	1652	**18789.00**	4289.40
StarGunner	10250	**127029.00**	20320.00
Tennis	-8.9	**0.00**	**0.00**
TimePilot	5925	12926.00	**24150.00**
Tutankham	167.6	**241.00**	203.54
Venture	1188	5.50	**1564.00**
VideoPinball	17298	**533936.50**	139799.22
WizardOfWor	4757	17862.50	**43436.00**
Zaxxon	9173	22209.50	**30818.00**
# Games >human		**37 (80%)**	34 (74%)
#Games >75% human		**38 (83%)**	37 (80%)
# Games Best	6 (13%)	**31(67%)**	10(22%)

TABLE 4.10: Comparison of N-CPL with a Human player's scores and the model-free RL algorithm Rainbow using human starts. Note that the experimental settings are different between N-CPL and Rainbow(Hessel et al. 2018), in terms of the training budget, maximum episode length, frame skips, using random no-op and loss of life signal, see Table 4.8 for a comparison of experimental settings.

for Atari games where the agent has multiple lives, while N-CPL uses a lookahead budget of 100 simulator calls for each selected action while the RL methods do not. Not withstanding the caveat of the difference in experimental settings, Table 4.9 shows N-CPL outperforms DQN's performance, while Table 4.10 shows Rainbow to outperform N-CPL however the number of games that Rainbow and N-CPL are better than the human baseline scores are similar.

4.5 Discussion

We have found significant discrepancies in the experimental settings used in the previous width-based planning papers for evaluating their algorithms over the Atari-2600 games. We believe a clear and consistent evaluation protocol should be set out for planning based algorithms applied to the Atari-2600 games to facilitate the direct comparison of their results. This could be similar to the evaluation protocol for the Atari-2600 games set out by Machado et al. (2018), which was mainly focused towards RL agents and included recommendations on episode termination, setting of hyper-parameters, measuring training data, summarising learning performance and injecting stochasticity. However Machado et al. do not discuss evaluation settings that are vital to the deterministic planning setting we have explored in this chapter, such as planning budgets, and caching of transitions within lookaheads. We hope that by having identified some of the discrepancies in the experimental settings of previous width-based algorithms, such as the size of the action set and the planning budget used, future research in planning agents for the Atari-2600 games can be more easily assessed. We were able to observe interesting patterns in the relative performance of algorithms through segmenting the Atari-2600 games by their different game characteristics. We are not aware of other works that analyse the performance of agents in regards to the characteristics of specific Atari-2600 games. We believe this taxonomy will provide useful insights into the behaviour of agents on the Atari-2600 games.

4.6 Conclusion

In this chapter we have focused on width-based planning methods that have been applied over the Atari-2600 games. It is important to note though that these algorithms are defined in a general way to operate over MDPs. We proposed new width-based planning and learning algorithms through the examination of different design decisions made by previous implementations of width-based planners. These new algorithms, particularly N-CPL, are simpler implementations than the previously introduced width-based planning and learning algorithms π-IW(1)+ and π-HIW(n, 1). N-CPL defines its features directly over the grayscaled pixel colours of the game screen and uses a simplified novelty definition. Furthermore, N-CPL learns a value function which is only used for cost-to-go approximations at the leafs of the lookahead search tree. N-CPL also uses a methodical learning schedule we introduced for training both its policy and value networks. We found N-CPL to outperform π-IW(1), π-IW(1)+ and π-HIW(n, 1) not only across the Atari-2600 benchmark, but also over subsets of games with large branching factors and games with sparse meaningful rewards. These results address **RQ1** by showing that N-CPL's integration of planning and learning pays off for almost real-time planning over hard problems.

Part II

Imitation Learning via regression

Introduction to Part 2

Learning algorithms often suffer from a cold-start problem, in which they rely upon random exploration of the environment in order to discover useful reward feedback. This Part explores the interface of planning and learning methods through the lens of using planning algorithms as teachers for learners on ASDM problems where useful reward feedback is sparse and unlikely to found by random exploration. In particular, this part investigates using demonstration trajectories to guide learners to promising areas of the search space. In Chapter 5 we investigate and propose improvements to a learning from a single demonstration method that learns from states that are progressively further away from the environment's goal. Using the insights of Chapter 5 we propose a new algorithm in Chapter 6 that aims to generalise more robustly to environments requiring multiple modes of behaviour. In order to address **RQ2** the algorithms we introduce in this part assume the provided demonstrations can be complete or partial trajectories and that they may have been created using a relaxed version of the environment. This allows us to show in Chapter 6 that our introduced algorithm can use a demonstration produced by a geometric planner on a relaxed version of the environment, to learn policies that can successfully achieve the environment's goal.

Chapter 5

Iterative backwards learning with intrinsic motivation

5.1 Introduction

For ASDM problems with reward functions containing vast plateaus traditional RL algorithms often either never converge to a solution or have a high sample complexity (Kober, Bagnell, and Peters 2013; Arulkumaran et al. 2017). The performance of RL methods on such domains can be improved by providing a demonstration of a solution (Argall et al. 2009). In particular, an idea that has been shown to be effective for RL algorithms to learn a policy that can surpass the performance of a demonstration is to learn from states in a demonstration that are progressively further away from the problem's goal (Salimans and Chen 2018; Ecoffet et al. 2019; Resnick et al. 2018). In this thesis we refer to these methods as Backwards Learning, but note that the training episodes are still generated in a typical RL manner using the environment's actions. "Backwards" simply refers to the selection of initial states used for training iterations starting near the problem's goal and regressing back towards states at the start of a provided demonstration trajectory. We introduce and discuss this Backwards Learning mechanism in 2.8.5.

In this chapter we modify the Backwards Learning method to have an iterative learning schedule. We refer to the introduced method as Iterative Backwards Learning (IBL). Additionally we explore the use of exploration bonuses with Backwards Learning techniques. Through an experimental study we investigate Backwards Learning algorithms on domains with both discrete and continuous action and state spaces.

Symbol	Description	Range
Δ	Interval step size	$[1, \infty]$
N_e	Number of evaluation episodes.	$[1, \infty]$
N_t	Threshold of successful evaluation episodes.	$[0, \infty]$
T_b	Threshold of unsuccessful training steps before backtracking.	$[1, \infty]$

TABLE 5.1: IBL hyper-parameters.

Algorithm 3 Iterative Backwards Learning (IBL)

```
1: Input: A demonstration $\delta = (s_0, R_0), (s_1, R_1), \dots, (s_m, R_m)$, interval $\Delta$, number of evaluation episodes $N_e$, success threshold $N_t$, backtrack threshold $T_b$, environment horizon $H$
2: Output: $\pi$ a policy
3: $k \leftarrow m - \Delta, t_b \leftarrow 0$
4: $\mathbf{I} \leftarrow \{s_k, s_{k+1}, \dots, s_{k+(\Delta-1)}\}$ {Interval of states to learn from}
5: $\pi \leftarrow sample(\Pi)$ {Initialising random policy}
6: repeat
7:   $n \leftarrow 0$ {Counter for successful evaluations}
8:   for $j \leftarrow 1$ to $N_e$ do
9:     $s_i \leftarrow sample(\mathbf{I})$
10:    $r \leftarrow$ EXECUTE_EPISODE($\pi, s_i, H - i$)
11:    if $r \geq R_m - R_i$ then
12:      $n \leftarrow n + 1$ {Episode matched or improved upon demonstration}
13:    end if
14:  end for
15:  if $n \geq 1$ then
16:    $t_b \leftarrow 0$
17:  else
18:    $t_b \leftarrow t_b + 1$ {All evaluation episodes unsuccessful}
19:  end if
20:  if $t_b \geq T_b$ then
21:    $k \leftarrow \min(k + \Delta, m - \Delta)$ {Progress interval closer to end of demonstration}
22:  else if $n \geq N_t$ then
23:    $k \leftarrow \max(0, k - \Delta)$ {Successful, regress interval closer to start of demonstration}
24:  else
25:    $\pi$ = IMPROVE($\pi, \mathbf{I}$)
26:  end if
27:  $\mathbf{I} \leftarrow \{s_k, s_{k+1}, \dots, s_{k+(\Delta-1)}\}$
28: until Trainning Budget Exhausted
```

5.2 Iterative Backwards Learning

The learning method introduced by Salimans et al. (2018) trains a policy π through a schedule of sets of initial states used to run training episodes, we refer to each set of initial states used as a different stage of the schedule. The set of initial states used for training start from states visited at the end of a demonstration and iteratively regress back, with a step size of Δ, towards states visited at the start of the demonstration. That is, the learning schedule uses a sequence of state sets $\{\mathbf{I}^1, \mathbf{I}^2, \dots, \mathbf{I}^j\}$, where $\mathbf{I}^i$ represents a set of states $s \in \mathcal{S}$ used as initial states of training episodes at the i-th stage of the schedule. It is assumed that the agent is provided a demonstration, $\delta = \{(s_0, R_0), (s_1, R_1), \dots, (s_m, R_m)\}$, where $s_i \in \mathcal{S}$ is the i-th state visited in the demonstration and R_i is the total accumulated reward up to step i of the demonstration trajectory. Note that the demonstration does not include actions, this means that a demonstration can be used even when access to its actions are not available. At stage i of the schedule the set of initial states used is defined as $\mathbf{I}^i = \{s_{m-(i-1)*\Delta-1}, s_{m-(i-1)*\Delta-2}, \dots, s_{m-i*\Delta}\}$ with corresponding accumulated rewards of $\mathbf{R}^i = \{R_{m-(i-1)*\Delta-1}, R_{m-(i-1)*\Delta-2}, \dots, R_{m-i*\Delta}\}$. That is, each set of initial state sets is a Δ-sized sub-sequence of the demonstration trajectory δ states. Note that the number of schedule stages i has a range of 1 to $\lceil m/\Delta \rceil$. The definition of $\mathbf{I}$ means that the first set of initial states $\mathbf{I}^1$ used for training episodes are the set of demonstration states within at most Δ steps to the end of the demonstration s_m. That is, s_m is reachable from the states $s \in \mathbf{I}^1$ within a sequence of at most Δ actions. Specifically, $\mathbf{I}^1 = \{s_{m-1}, s_{m-2}, \dots, s_{m-\Delta}\}$. Once a percentage of episodes, N_t/N_e, starting from states $s_i \in \mathbf{I}^1$ have an accumulated reward equal to or greater than $R_m - R_i$, that is the demonstration trajectory's accumulated reward from s_i to s_m, the set of initial states are updated. The update of the initial states involves changing the initial states used for the training episodes from $\mathbf{I}^1$ to $\mathbf{I}^2 = \{s_{m-\Delta-1}, s_{m-\Delta-2}, \dots, s_{m-2*\Delta}\}$. This process repeats until the schedule reaches a set of initial states that includes the start state of the demonstration. That is, the process repeats until a set of states $\mathbf{I}^j$ is reached that includes the initial state of demonstration s_0. From this point on wards we refer to this method introduced by Salimans et al. (2018) as Backwards Learning (BL).

We introduce Iterative Backwards Learning (IBL) which modifies the learning schedule detailed above. IBL allows the set of initial states $\mathbf{I}$ to not only regress over the states of the demonstration towards s_0 but also progress back towards the end of the demonstration s_m. Table 5.1 details IBL's hyper-parameters and Algorithm 3 provides a full description of IBL. IBL starts by initialising the set of initial states to the demonstration states within Δ steps of the end of the demonstration s_m. That is $\mathbf{I} = \mathbf{I}^1 = \{s_k, s_{k+1}, \dots, s_{k+(\Delta-1)}\}$ and $k = m - \Delta$ (lines 3-4 Alg. 3). Additionally IBL initialises a random policy π (lines 5 Alg. 3). N_e evaluation episodes starting from states s_i sampled from $\mathbf{I}$ are then executed using π (lines 8-14 Alg. 3) with a horizon of $H - i$. An evaluation episode in IBL is defined as successful if its accumulated reward $r \geq R_m - R_i$ (lines 11-13 Alg. 3), where $R_i \in \delta$ is the accumulated reward of the demonstration δ up to the demonstration state $s_i \in \mathbf{I}$ and R_m is the demonstration's total accumulated reward. After evaluation there are three possible scenarios. Scenario (1) is that the number of successful evaluation episodes n is greater than or equal to the set threshold N_t. Scenario (2) is that there have been no successful evaluation episodes for the last T_b training iterations, where a training iteration is one call of the IMPROVE function (line 25 Alg. 3) as described below. Lastly, scenario (3) is when neither (1) or (2) is true. For scenario (1) the initial state set $\mathbf{I}$ is updated by regressing the initial state interval Δ steps closer to the start of the demonstration as previously described (lines 22-23 and line 27 Alg. 3). That is where $\mathbf{I} = \{s_k, s_{k+1}, \ldots, s_{k+(\Delta-1)}\}$, k is decremented Δ times. For scenario (2) the evaluations are unsuccessful and the set of initial states is updated to the demonstration states that are Δ steps closer to the end of the demonstration. That is, k is incremented Δ times (lines 20-21 and line 27 Alg. 3). For scenario (3) a training iteration of π is executed by the IMPROVE function using episodes starting from states $s \in \mathbf{I}$ (line 25 Alg. 3). This evaluation and training process is repeated until a defined computation budget is exhausted (lines 6-28 Alg. 3).

5.3 Exploration with Backwards Learning

While backwards learning techniques are designed to work with any on-policy learning algorithm, previous works using backwards learning techniques have only reported results using PPO (Salimans and Chen 2018; Resnick et al. 2018). We follow previous work by using PPO in this work. Please see 2.8.2 for a detailed description of PPO.

Previous Backwards Learning approaches have controlled exploration for PPO via the entropy loss coefficient c_2 in the loss function shown in Equation 2.12. Salimans and Chen (2017) note that to successfully learn from the demonstration required careful tuning of the entropy loss coefficient for the Atari game Montezuma's Revenge, but the tuning method failed to learn successful policies for other Atari games.

State normalisation as described by Mania, Guy and Recht (2018) ensures that the different components of the state have an equal influence on the policy no matter the magnitude of their range of values. State normalisation can also provide a method of non-isotropic exploration as the state value distributions used for the normalisation change over the course of the training. This is likely to be particularly prevalent in the learning schedule methods we explore as they progressively train from different initial states defined through a demonstration, and the distribution of state values may greatly vary over the demonstration.

RL exploration can also be controlled through adding exploration bonuses to the reward function, which is often referred to as intrinsic motivation. This can be implemented replacing r_t in

Equation 2.13 with $\hat{r}_t = r_t + c_3\eta$, where c_3 is the exploration bonus coefficient and η is a method such as count-based exploration (Bellemare et al. 2016b) which we discuss in 2.8.3. In this work we will compare several entropy coefficients and other exploration bonuses. The two different exploration bonuses, η, are state counts,

$$\eta = (N(s') + 0.01)^{-1/2} \tag{5.1}$$

and the UCB1 formula (Auer, Cesa-Bianchi, and Fischer 2002),

$$\eta = \sqrt{2\log(N(s))/(N(s,a))} \tag{5.2}$$

where $N(s)$ is the number of times state s has been visited and $N(s,a)$ is the number of times action a has been executed from state s.

For continuous domains state counts cannot be directly used (Tang et al. 2017). However, a pseudo-state count can be defined by assuming a probability distribution for the values of each state variable $s^i \in s$ that the agent observes (Bellemare et al. 2016a; Martin et al. 2017). In this work we assume the state variable values are normally distributed, $\mathcal{N}(\mu(\mathcal{T}^i), \sigma^2(\mathcal{T}^i))$, where $\mathcal{T}^i$ is the sequence of state variable values s^i that have been observed by the agent. Using the cumulative distribution ϕ_i of $\mathcal{N}(\mu(\mathcal{T}^i), \sigma^2(\mathcal{T}^i))$ we can calculate the approximate visit count as,

$$\tilde{N}(s) = N \prod_{s^i \in s} min(\phi_i(s^i), 1 - \phi_i(s^i)) \tag{5.3}$$

where N is the total number visits to any state and the cumulative distribution ϕ_i is updated with the history of states visited before visiting s. We use $\tilde{N}$ in place of the N function in equation 5.1 and modify equation 5.2 to the following,

$$\eta = \sqrt{2\log(\tilde{N}(s))/(\tilde{N}(s') + 0.01)} \tag{5.4}$$

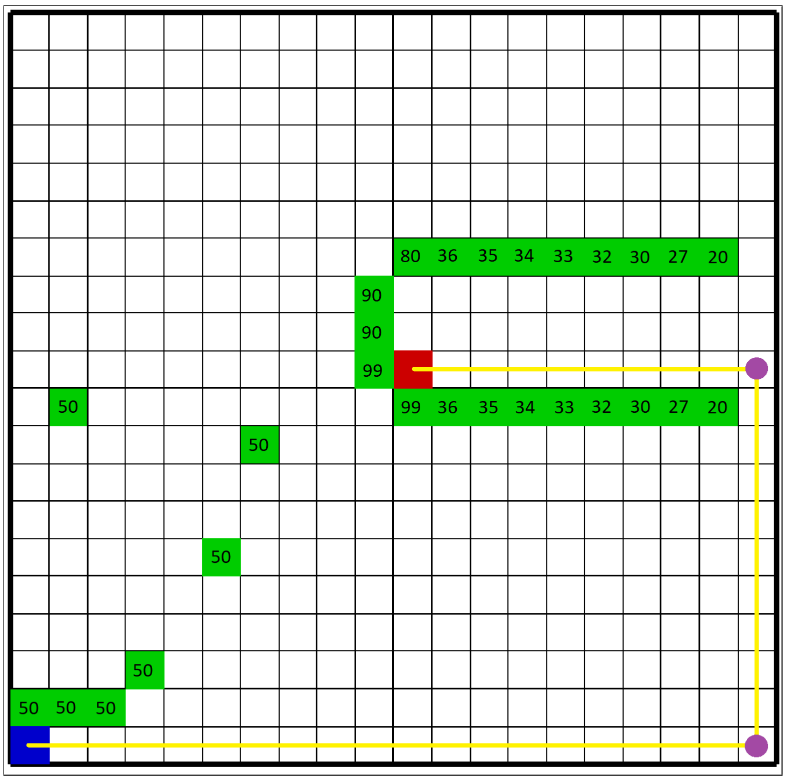


FIGURE 5.1: 20 by 20 Grid World domain. The blue cell is the agent, red cell is the goal and the green cells have the percentage chance represented by the number inside the cell of being a mine. The yellow line shows the trajectory of the demonstration we provide to the learning algorithms, and the purple circles mark states along the trajectory where the moving direction of the agent changes.

5.4 Experimental study

5.4.1 Domains

Discrete State Space and Discrete Actions

We use a slightly modified version of the stochastic CTP (Papadimitriou and Yannakakis 1991), which is introduced in 2.5.1, in order to evaluate and gain insight into the IBL algorithm and the effects of the exploration techniques used.

To allow for easy visualisation we model an instance of CTP through a GridWorld domain with obstacles and a goal. The agent can move up, down, left, or right but any action into the boundary of the GridWorld results in no change to the environment. There is a -1 reward for each action except for an action that leads to the goal state which has a reward of 0 and the episode is terminated once $H = 99$ actions have been executed or the agent reaches the goal. We treat obstacles as mines so that any move into a mine results in the agent remaining in this position for the rest of the episode no matter the action applied. The state given is (x, y, m) where x, y

are the grid coordinates of the agent and m is the number of mines in the adjacent neighbouring cells to the agent. The mine placement in the grid follows a probabilistic distribution that is not directly observable by the agent. The probabilities of cells being a mine are shown in Figure 5.1.

The optimal behaviour on the CTP instance shown in Figure 5.1 is to traverse the row of cells below the mines that are below the goal, and once a cell is reached that has no adjacent mines the agent should directly transverse towards the goal.

Continuous State Space and Continuous Actions

We use the *Mobile-Robot-Ob-Stay* (MROS) domain introduced in 2.5.2, which has both a continuous state and action space. For the MROS domain we use a horizon H of 99 time steps. Additionally, we also report a version of MROS where random disturbances $w_x = \mu(0, \sigma_x^2)$ and $x_y = \mu(0, \sigma_y^2)$ are added to the velocities v_x and v_y respectively at each time step. We set both σ_x and σ_y to be 0.05 and refer to this instance as MROS with noise.

5.4.2 Methodology

For each domain we run PPO with 3 different learning schedules. The first schedule always trains on episodes starting from the initial state of the problem, s_0, we refer to this learning schedule as L0. The schedule used by Salimans et al. (2018) we refer to as BL.

There are a number of different hyper-parameters to consider for the IBL algorithm. We test different values for the interval at which the schedule progresses Δ, the threshold of successful evaluation episodes N_t, the PPO coefficients c_2 and c_3, and the type of exploration used. For each run we set the training budget to be 5 million simulator calls. We run 20 evaluation episodes for recording metrics each time the step in the learning schedule changes and after every 10,000 simulation calls. Note that the evaluation episodes are run from the problem's initial state and not the initial state sets from the learning schedules.

We use openai's PPO implementation (Dhariwal et al. 2017) and use a fully connected network with 2 hidden layers each having 64 units for both the value and policy networks for all domains. For the CTP with mines domain that has discrete actions the policy network outputs the probability of taking each action given the current state. For the MROS domains that have continuous actions the policy network outputs the mean value of each action variable given the state along with standard deviation values, these values are then used to sample an action from the policy.

5.5 Results

5.5.1 Inspecting learnt value and policy functions

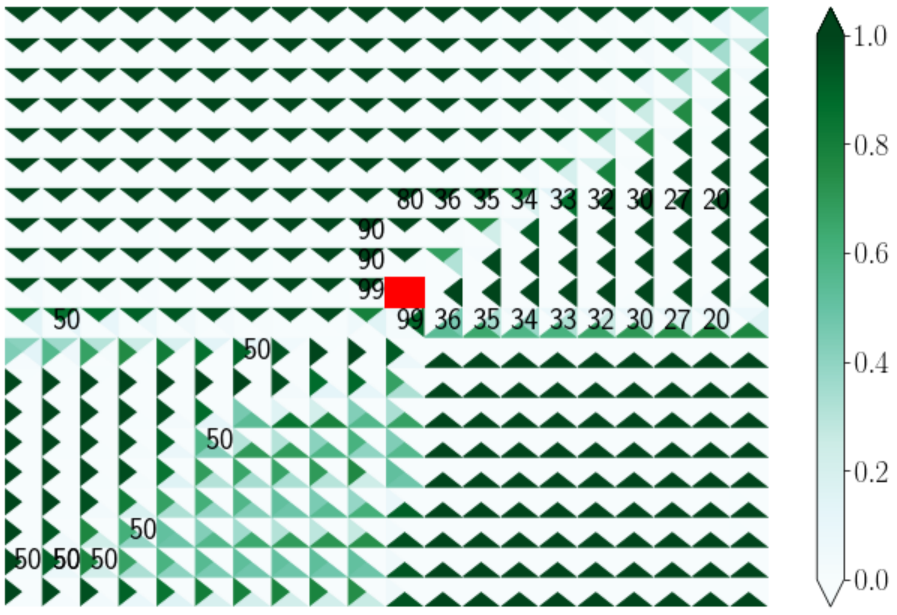
An example of learnt value and policy functions from a successful IBL trial on the CTP with mines problem are shown in Figure 5.2. From Figure 5.1 that shows the demonstration trajectory provided to the IBL algorithm we can see in Figure 5.2 that IBL learns to generalise its policy and value functions beyond the states contained within the demonstration. That is, for any position in the grid if an agent greedily follows the policy represented in Figure 5.2 the agent will end up at the goal location. It is also interesting to note that the learnt policy of IBL does not imitate the demonstration's trajectory but instead learns a policy which deviates from the trajectory in order to reach the goal quicker. This can be observed in the cells on the right half of the bottom row of the grid where the policy shown in Figure 5.2 selects up actions, while the demonstration, shown in Figure 5.1, selects right actions. However, it is important to note that all successful trials do not generalise as well as the learnt functions shown in Figure 5.2. For example, Figure 5.3 shows a different successful trial of IBL on CTP. Figure 5.3 shows that the policy function selects the correct actions for states along the demonstration trajectory (shown in Figure 5.1), however for cells in the upper half of the grid the agent always selects the left action away from the goal. This is because to be successful the agent only has to reliably reach the goal from states along the demonstration trajectory.

5.5.2 Impact of interval size (Δ)

Table 5.2 provides an overview of the performances of L0, BL, and IBL given different intervals sizes. It is clear from the table that the IBL algorithm dominates BL over the three domains given any interval value. It is also interesting to note that the impact of the Δ size appears to be problem dependent as for the CTP problem both BL and IBL benefit from smaller Δ values, while for the MROS problems IBL benefits from the larger Δ values and BL's performance does not appear to be obviously influenced by Δ.

5.5.3 Entropy and exploration bonus terms

Table 5.3 shows the results of the CTP with Mines domain for the algorithms using PPO with no entropy bonus term ($c_2 = 0$). Tables 5.3 and 5.2 show that for the CTP with Mines problem both



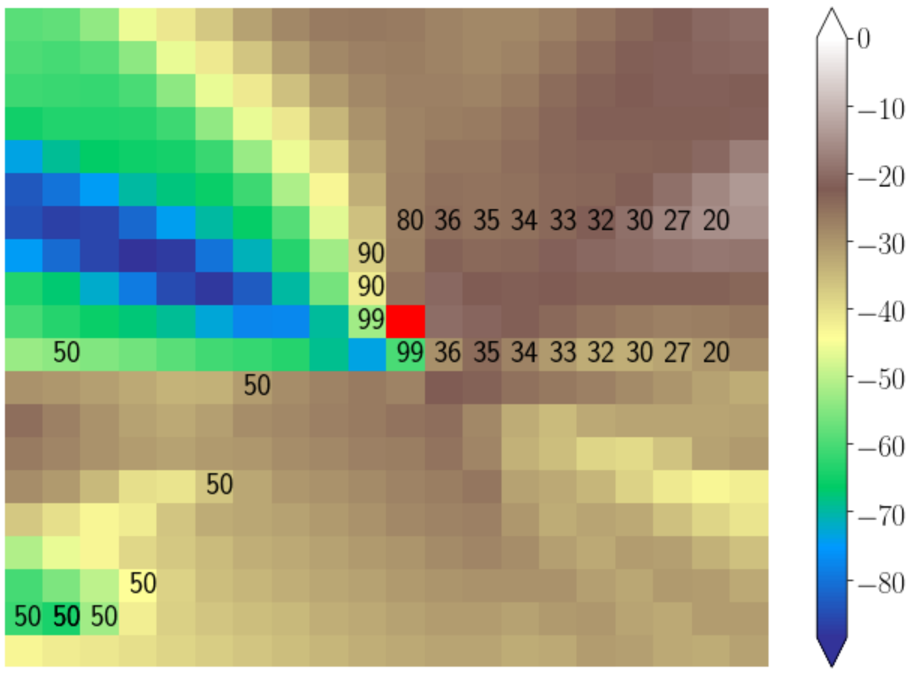


FIGURE 5.2: Representation of the policy and value networks learnt for each state of the CTP with Mines problem. For the policy and value networks values shown each state is assumed to not have any mines in the adjacent cells, that is the values are shown the input states are $(x, y, 0)$. Note that each cell in the grids matches the cells shown in Figure 5.1 where the red square represents the goal position and the numbers inside the cells are the percentage chance of the cell being a mine. The policy network is represented by the top grid where the arrows point towards the direction of action to be selected by the agent, note that each cell has 4 arrows in it that represent the 4 actions available in that cell. Additionally note that the colour scaling indicates the probability of the policy selecting the action. The value network is represented by the bottom grid.

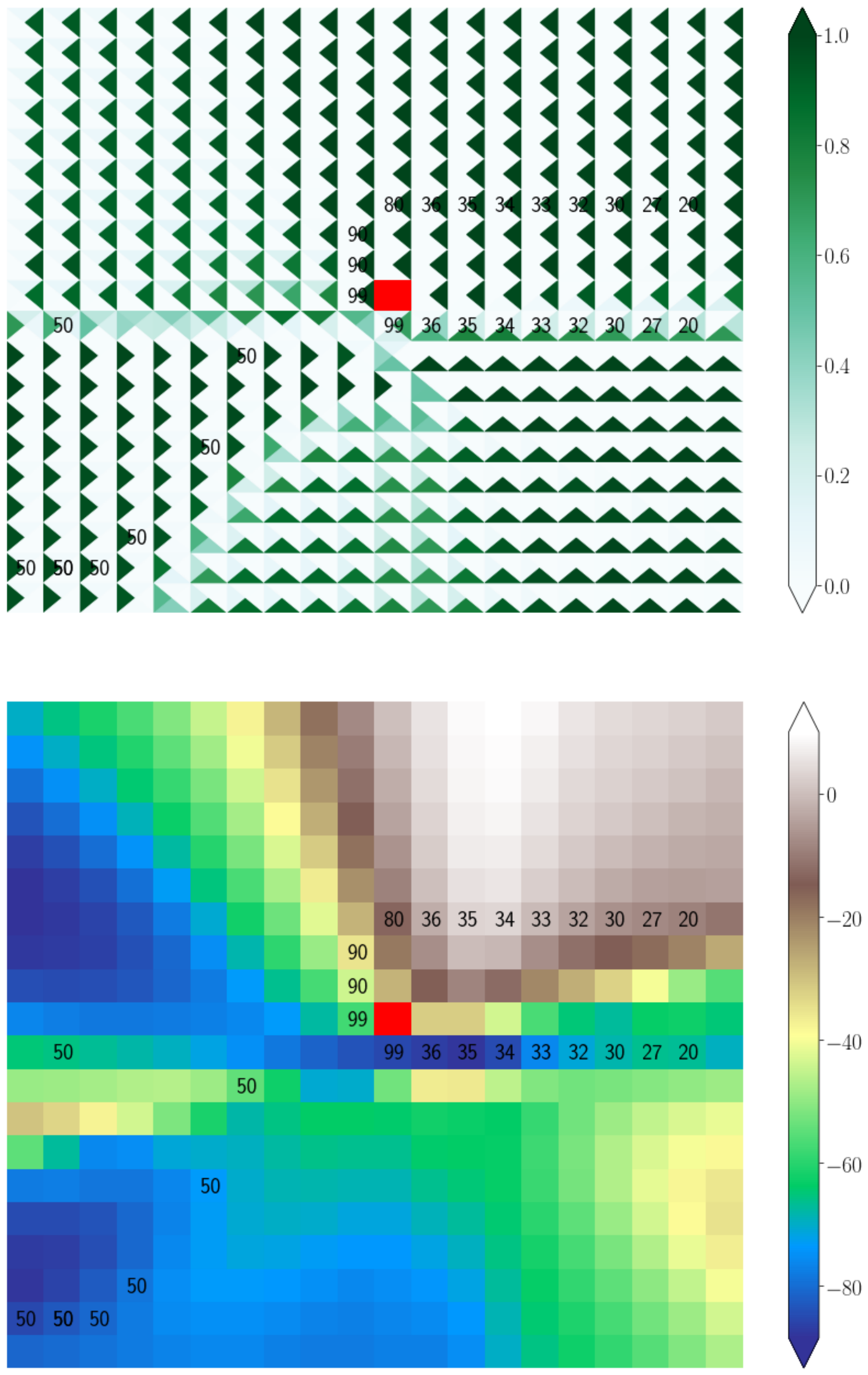


FIGURE 5.3: Same as Figure 5.2 but a different learning trial.

BL and IBL deteriorate from using the entropy bonus term. Additionally, Table 5.3 shows that the Count-based exploration bonus term generally deteriorates the performance of the algorithms.

		% of trials ≥ demo's performance		
Algorithm	**Interval Size (Δ)**	**CTP with Mines**	**MROS**	**MROS-Noise**
L0	NA	0	0	0
BL	1	40	35	20
	2	15	30	35
	3	30	35	35
	4	5	25	35
	5	5	30	15
IBL	1	**55**	50	45
	2	25	75	60
	3	40	55	80
	4	20	**80**	**95**
	5	10	**80**	75

TABLE 5.2: Summary of L0, BL and IBL performances given different interval sizes. A total of 20 trials were used for each setting. For these results the entropy bonus term of PPO is set as $c_2 = 0.01$ and no exploration bonus term discussed in 5.3 is used. Additionally the evaluation threshold is set to $4/5$ ($N_t = 4$, $N_e = 5$).

Eval. Threshold (N_t/N_e)	**Algorithm**	**Interval Size (Δ)**	**Exploration Bonus Term**	**% of trials ≥ demo's performance**
NA	L0	NA	None	0
			Count	0
1/5	BL	1	None	0
			Count	20
		5	None	10
			Count	0
	IBL	1	None	40
			Count	35
		5	None	10
			Count	5
4/5	BL	1	None	65
			Count	50
		5	None	35
			Count	30
	IBL	1	None	**90**
			Count	65
		5	None	40
			Count	25

TABLE 5.3: Summary of L0, BL and IBL performances over the CTP with Mines problem given different evaluation thresholds, interval sizes, and exploration bonus terms. A total of 20 trials were used for each setting. For these results the entropy bonus term of PPO is set as $c_2 = 0.0$ and for the settings with an exploration bonus term use $c_3 = 0.01$.

Figures 5.4, 5.5, 5.6, 5.7, 5.8 and 5.9 show that there are no clear effects of the different exploration bonuses on the performance of the learning methods for both the CTP and MROS domains, as the 95% confidence limits of the different methods almost always overlap.

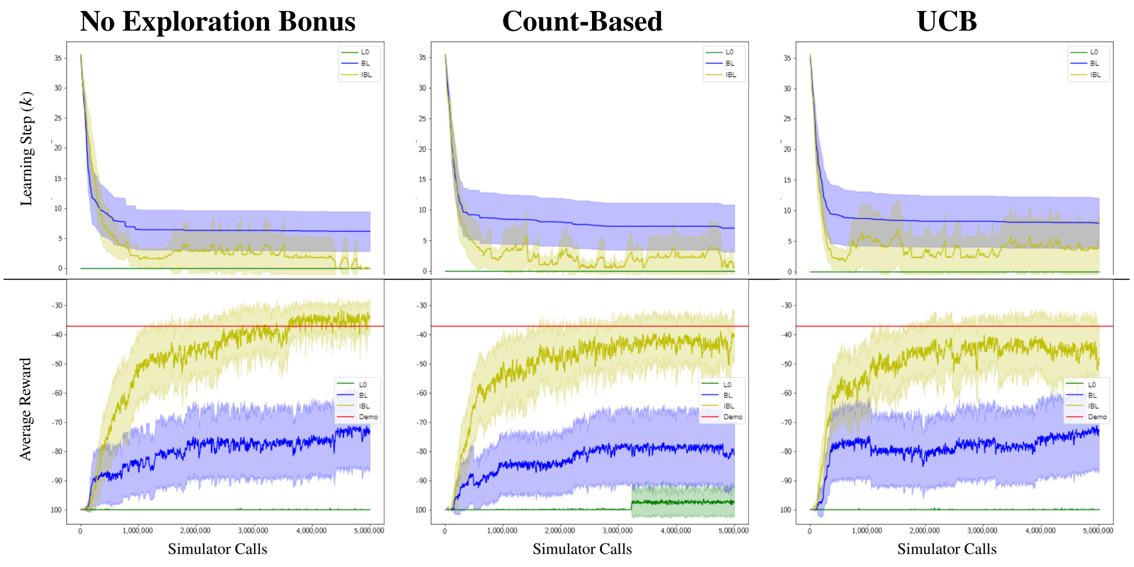


FIGURE 5.4: CTP with mines, with an evaluation threshold of 1/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. The top row of graphs show the demonstration step k which the method is learning from versus the number of simulator calls used. For example for IBL the calculation of k is shown in Algorithm 3. The bottom row shows the average reward received from the learnt policies versus the number of simulator calls. The lines represent the average across 20 trials and the shading represents the 95% Confidence Intervals.

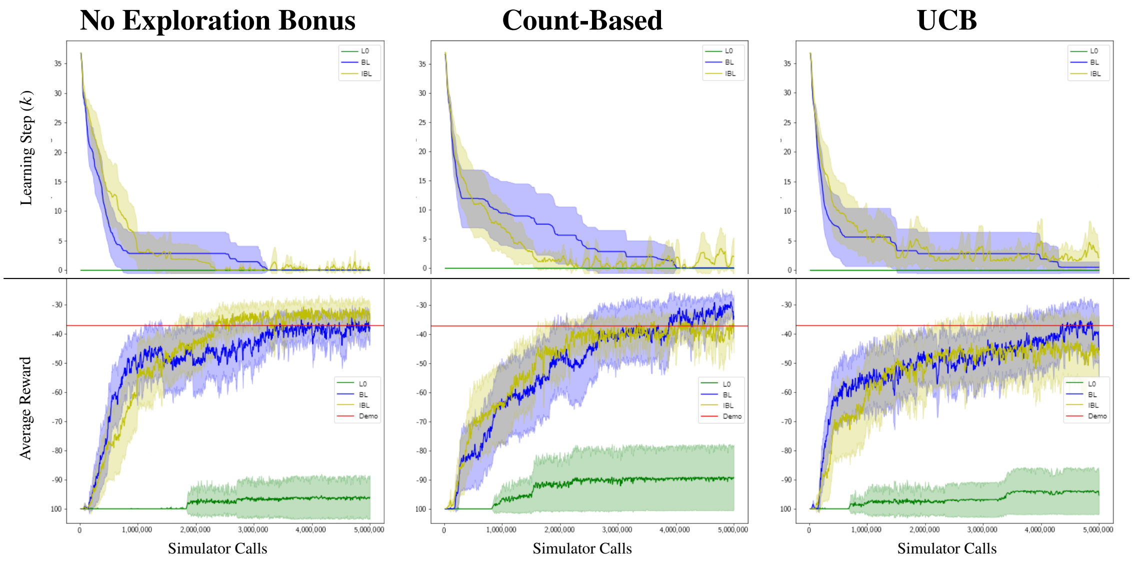


FIGURE 5.5: CTP with mines, with an evaluation threshold of 4/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. See Figure 5.4 caption for graph details.

5.5.4 Evaluation threshold

The evaluation threshold affects how easily the learning schedule of initial states (**I**) moves towards the start of the demonstration. Table 5.3 makes it clear that given the entropy bonus term of $c_2 = 0.0$ for the CTP with mines problem, both BL and IBL benefit from the higher evaluation threshold of $4/5$ versus $1/5$. Furthermore, figures 5.4 and 5.5 contrast the performance of the

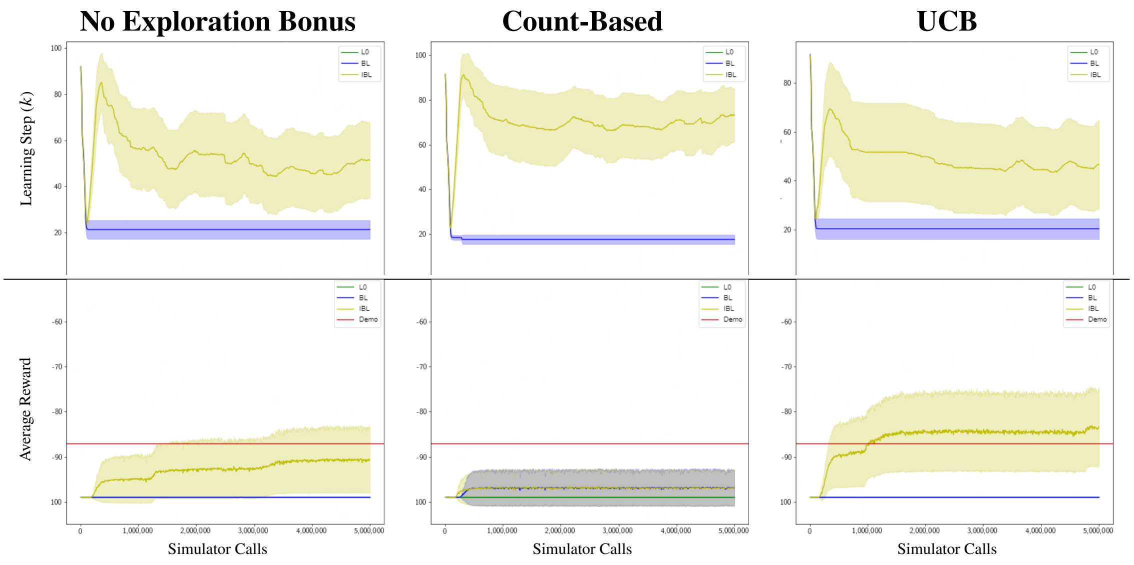


FIGURE 5.6: Mobile-Robot-Ob-Stay domain, with an evaluation threshold of 1/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. See Figure 5.4 caption for graph details.

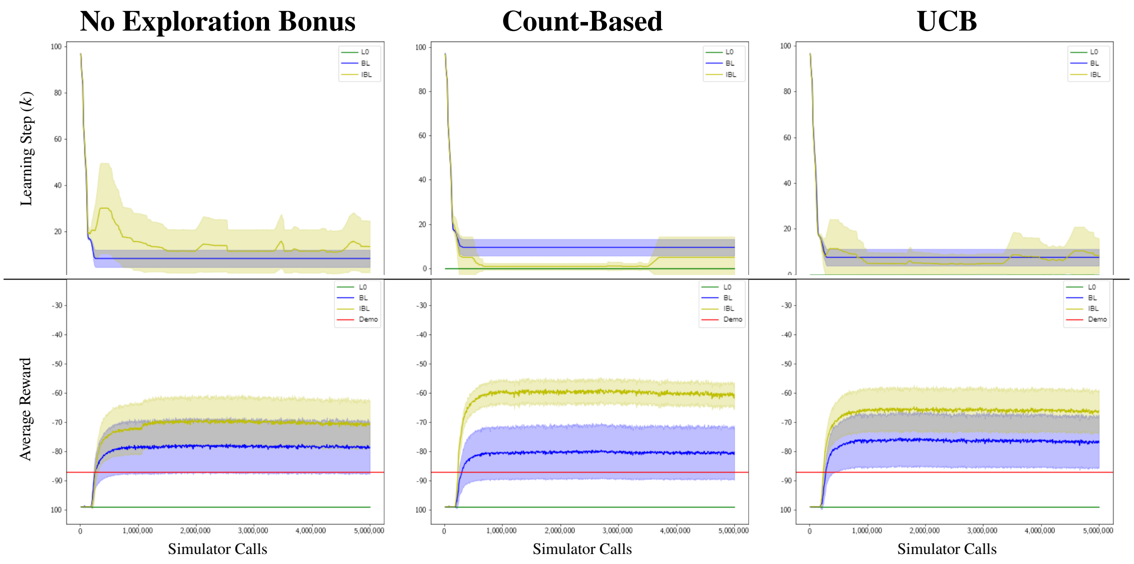


FIGURE 5.7: Mobile-Robot-Ob-Stay domain, with an evaluation threshold of 4/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. See Figure 5.4 caption for graph details.

learning methods using the $1/5$ versus $4/5$ threshold. It is clear that for the CTP with mines problem the performance of BL deteriorates much more than IBL when using the $1/5$ instead of the $4/5$ threshold. This trend continues for the MROS domains evidenced by figures 5.6 and 5.8, showing that IBL dominates BL for the evaluation threshold of $1/5$, and figures 5.7 and 5.9 showing that for the $4/5$ threshold, BL's 95% confidence intervals often overlap with IBL's intervals.

5.5.5 Evolution of learning schedule

In figures 5.4, 5.5, 5.7, 5.6, 5.9 and 5.8 the stage of the learning schedule k is shown in the top row of graphs. The results show that it is common for the BL method to not complete its learning schedule by progressing to $k = 0$ within the simulator budget for both CTP and MROS. All of IBL's trials also do not always reach the $k = 0$ stage of the learning schedule. However, on average in the majority of cases IBL's trials progress closer to the $k = 0$ stage of the learning schedule than BL. Note that these graphs show the average and 95% Confidence Intervals over 20 trials, such that it is possible 1 or more trials reach a learning stage of $k = 0$ even if $k = 0$ is not in the Confidence Interval. In the next section we discuss the common features of the demonstration states that we observed the BL method to get stuck at in its learning schedule.

5.5.6 Key takeaways

The results show that across the majority of settings we tested, with domains with both discrete and continuous actions and state spaces, IBL outperformed the BL method. However, due to the stochastic nature of the Backwards Learning evaluation and training procedures, no setting of IBL successfully learnt policies for every trial that could match or outperform the demonstration. We observed that the BL method is often unable to progress past particular demonstration states along its learning schedule. Through manually inspecting the states BL would get stuck at we observed that the states were often pivot points in the demonstration's behaviour. For example, for the CTP mine problem BL often does not progress beyond states of the demonstration that have a change in direction. The states of the demonstration for the CTP mine problem that have changes in direction are shown in Figure 5.1 as purple circles.

Additionally, the results suggest that for a majority of instances the exploration bonus terms we explored have a negative affect on the learning methods. The effect of the interaction between the exploration mechanisms of the entropy bonus term, state-normalisation, and the additional exploration bonus terms we explored for PPO with backwards learning methods remain as open questions. Ultimately our results suggest for the CTP and MROS domains including just the state-normalisation as a means for exploration through non-isotropic exploration for the Backwards Learning methods is enough to learn policies that can match or exceed the performance of the demonstration provided.

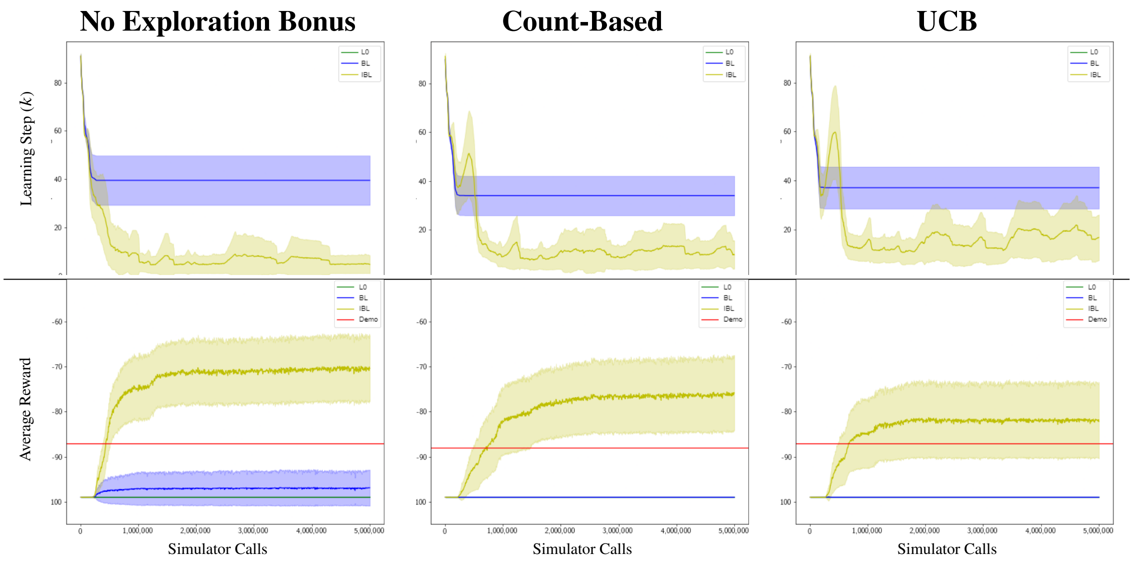


FIGURE 5.8: Mobile-Robot-Ob-Stay with Noise, with an evaluation threshold of 1/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. See Figure 5.4 caption for graph details.

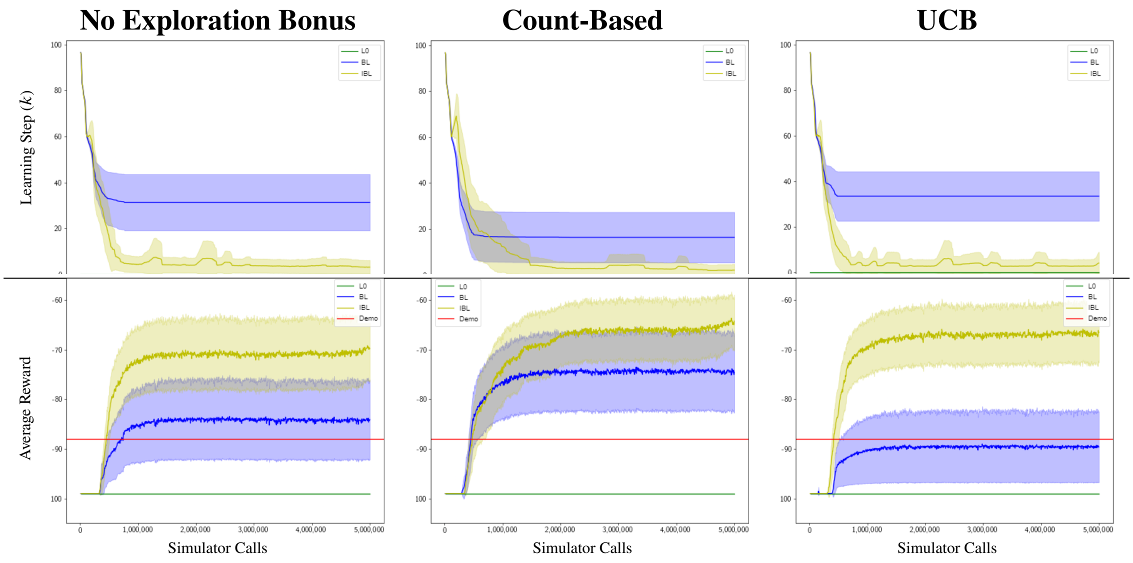


FIGURE 5.9: Mobile-Robot-Ob-Stay with Noise, with an evaluation threshold of 4/5. $\Delta = 1$, $c_2 = 0.01$ and $c_3 = 0.01$. See Figure 5.4 caption for graph details.

5.6 Conclusion

We introduced a new method for learning from a single demonstration which is based on the idea of learning from initial states that are progressively harder to solve. The Backwards Learning methods we introduced and explored showed that Backwards Learning can benefit from not only increasing the difficultly of initial states but also from decreasing the difficulty when the agent is unsuccessful in learning useful policies.

The Backward Learning method we explored in this chapter begins to address **RQ2** as the method regresses through a given demonstration, and has the ability to work with full, relaxed or partial demonstration trajectories. However, the methods would often get stuck in their learning schedule at positions in the demonstration where changes in the mode of behaviour are required from the agent. While the IBL method improved on BL to often progress past such pivot points in the demonstration, IBL sometimes also got stuck at the same points. In the next Chapter we explore a method, that instead of modifying the Backwards Learning schedule as we proposed in this chapter, uses Backwards Learning over piece-wise policies. The goal of learning over piece-wise policies is for the Backwards Learning method to better capture the different modes of behaviour required from the environment through learning separate policies that are most robust and generalisable than the policies learnt in this chapter.

Chapter 6

Imitation Learning via symbolic pre-imaging [1]

In this chapter we follow up from Chapter 5 by introducing a new learning from a single demonstration method which aims to be more robust over problems requiring different modes of behaviour. Ultimately, we finish addressing **RQ2** by showing that the method we introduce can successfully learn useful policies from complete, partial or relaxed demonstration trajectories, and that the policies learnt can generalise beyond the environment settings they were trained on.

6.1 Introduction

Deep Reinforcement Learning over continuous control spaces is a recent and exciting line of research in Machine Learning and Control. However, on problems requiring complex behaviour with reward functions containing vast plateaus, traditional Reinforcement Learning (RL) algorithms rarely converge to a solution or have a high sample complexity (Kober, Bagnell, and Peters 2013; Arulkumaran et al. 2017). One way to overcome this is to use Hierarchical Reinforcement Learning (Barto and Mahadevan 2003) where high level actions named *options* or *skills* are used in order to make the underlying problem easier to solve. However the options need to first be discovered which normally requires that meaningful reward feedback can be achieved through following random policies from the initial state in order to derive automatically the high-level actions (Konidaris and Barto 2009b; Bagaria and Konidaris 2019). Another way of solving

[1]This chapter is adapted from the pre-print article "Imitation Learning via Symbolic Pre-Imaging"

for complex domains is to provide one or multiple demonstrations of a solution (Hester et al. 2018). For example, Imitation Learning approaches aim to emulate the behaviour displayed by the demonstration without access to the environment's cost function (Bain and Sammut 1995; Ho and Ermon 2016; Torabi, Warnell, and Stone 2019). An alternative approach to Imitation Learning for learning from a demonstration is to learn backwards from the states in the demonstration (Salimans and Chen 2018; Resnick et al. 2018), that is start learning episodes from states near the end of a demonstration and progressively learning from states earlier in the demonstration. Learning backwards has been used in methods that achieve state-of-the-art results on problems where traditional RL methods have struggled, such as the Atari-2600 game Montezuma's Revenge (Ecoffet et al. 2019; Salimans and Chen 2018).

We build on the main ideas from Hierarchical Reinforcement Learning, learning backwards from a demonstration, and imitation learning. Our approach aims to solve Goal Markov Decision Problems (MDPs), and synthesise robust policies that generalise better than previous methods, particularly for domains requiring different modes of behaviour. The three main contributions of our method with respect to previous methods are: (1) a novel method to *partition the state space symbolically* into polytopic regions from which different sub-policies are learnt, (2) a *general domain-independent cost function* to define the problem solved by a sub-policy, and (3) an efficient method to *select starting states*, exploiting the compact representation of state regions as polytopes, for policy iteration.

Our method partitions the state space into convex regions defined through recursive *pre-imaging* of the goal set and represents them symbolically using polytopes. For each partition a unique MDP is defined and a policy is trained to solve the MDP. We show that this use of piecewise policies is better able to handle environments that require complex behaviour. This is similar to the concept of *options* in RL (Sutton, Precup, and Singh 1999) given that our method uses a number of unique sub-policies in order to solve MDPs. However, rather than having to design each option, or discover the options through episodes run from the initial state of the problem (Konidaris and Barto 2009b), our method elicits automatically through a given demonstration sub-problems and partitions the state space accordingly.

Existing backwards learning methods, like those explored in Chapter 5, optimise the environment's cost function directly. This makes them applicable only over instances where the cost function is available to the learner. Additionally, previous methods ignore any information from the demonstration besides using its states as starting states for policy iterations. Instead, our proposed cost function exploits not only the traversed states, but also encourages the agent towards regions of the state space visited by the demonstration. Importantly, doing so does not require the learner to know the cost function being optimized. Furthermore we show the benefit of learning from suitably chosen additional states instead of just learning from the states of the demonstration. Our method also differs from imitation learning algorithms like GAIL and BC, introduced in 2.8.4, as the methods we introduce do not require the demonstration's actions.

We evaluate our new approach, *Backwards Learning with Piecewise-defined Policies* (BLPP), on different control domains including the OpenAI's `gym` (Brockman et al. 2016) "classic control" benchmarks. BLPP learns policies that outperform those learnt from previous backwards learning, RL and imitation learning algorithms.

6.2 Comparison with Skill Chaining

The BLPP algorithm we introduce in this chapter is similar to Skill Chaining and more specifically Deep Skill Chaining (DSC), introduced in 2.8.6, in that it learns a set of unique policies by first learning policies which are executed near the goal set and then policies that are closer to the initial state of the problem. Where BLPP differs from DSC is that it decomposes the problem's Goal MDP into a set of Goal MDPs with different initial and goal states that are easier to solve then the original MDP. This results in BLPP not needing to rely upon random exploration from the initial state of the problem as DSC does to the find the goal set but instead uses a demonstration to guide where in the state-space to learn each piece-wise policy. For the Mobile-Robot problems we explore in this chapter the chance of random policies achieving the domains goal is extremely small and no methods in our experiments that rely upon random exploration from the initial state found successful trajectories. Additional differences between BLPP and DSC are their definition of the global policy, and how policy execution is defined. BLPP defines its policies over controllable sets, which are defined symbolically through polytopes regions of the state-space, while DSC defines the initialisation and termination sets of its options through a trained classifier. For the global policy, BLPP defines which policy to execute at every step of the environment according to controllable sets which are learnt for each different policy, while DSC learns a global policy to decide which option to execute.

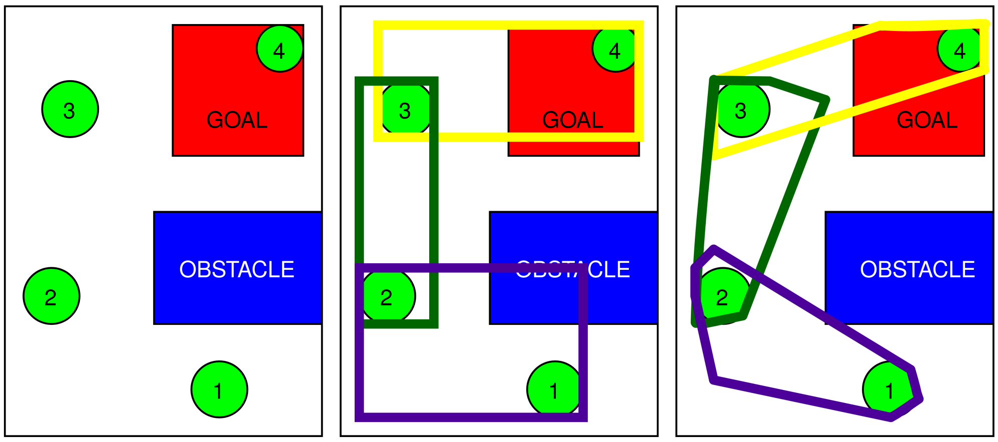


FIGURE 6.1: Overview of how BLPP obtains the approximate recursive pre-images of the goal set on a simple 2 dimensional problem. Left: shows the provided trajectory to imitate (green circles, numbers denote temporal order of demo states). Middle: Bounding boxes (hyperrectangles) used initially to bootstrap sub-policy learning. Right: Polytopes covering states in successful trajectories run from initial states within the bounding boxes. BLPP tightens first the approximate pre-image (polytope) closest to the goal set and proceeds backwards towards the first state in the trajectory provided (first polytope processed is yellow, second is green, last is purple).

6.3 Piecewise Backwards Learning

In this chapter we consider Goal MDP problems introduced in 2.1.2 with continuous state spaces, and assume we have access to settable black-box simulator (introduced in 2.2.2) that also returns if a resulting state is in the goal set $s' \in \mathcal{S}_G$.

We address the problem of finding a feasible policy π for a Goal-MDP $M^0 = (\mathcal{S}, s_0, \mathcal{S}_G, A(s), T(s, a, s'), c(s, a))$, given a successful *demonstration* trajectory, $\delta = s_0, \ldots, s_k, \ldots, s_{|\delta|-1}$, following an unknown policy π_δ. BLPP's high-level method is to recursively approximate the pre-image of the goal set through exploiting the information provided by the demonstration. Figure 6.1 provides an overview of how the approximate pre-images are computed. BLPP initialises a unique policy for each approximate pre-image. Once an approximate pre-image has been created, and if its corresponding policy is classified as successful, the next pre-image is computed as shown in Figure 6.1. Otherwise a training iteration of the policy is run and a new approximate pre-image is computed. This procedure is run until a successful policy is found for an approximate pre-image which includes the initial state of the demonstration.

We define successful trajectories as ones that reach the goal set and stay in it until the last state of the trajectory.

Definition 6.1 (Successful Trajectory)**.** A trajectory $\tau = s_0, \dots, s_n$ is successful for the goal set $\mathcal{S}_G$, denoted as $succ(\tau, \mathcal{S}_G)$, if there exists a state $s_i \in \tau$, such that $s_j \in \mathcal{S}_G$ for $j \geq i$, and $s_k \notin \mathcal{S}_G$ for $k < i$.

δ is assumed to be successful $succ(\delta, \mathcal{S}_G)$. Our aim is not to recover the policy π_δ from δ. We instead use δ to learn a feasible policy π for M^0. We assume that a state $s_k \in \delta$ is reachable with probability 1 within L steps from s_{k-1}, where $L \leq H$ is bounded by the problem horizon H. We observe that for a complete demonstration, that is $L = 1$, and $s_{|\delta|-1} \in \mathcal{S}_G$, $s_{|\delta|-2}$ is a member of the *pre-image controllable set* $\text{Pre}(\mathcal{S}_G)$.

Definition 6.2 (Pre-Image Controllable Set)**.** The pre-image controllable set of $\mathcal{S}_s \subseteq \mathcal{S}$ is defined as $\text{Pre}(\mathcal{S}_s) \triangleq \{s \in \mathcal{S} \mid \exists a \in A(s), \exists s' \in \mathcal{S}_s, T(s, a, s') > 0\}$.

That is, the controllable set for $\mathcal{S}_s$ contains any state that can reach a state in $\mathcal{S}_s$ within a single transition. Using the controllable set definition recursively we can define an N-step controllable set $\mathcal{K}_N(\mathcal{S}_s)$ for a target set $\mathcal{S}_s \in \mathcal{S}$.

Definition 6.3 (N-Step Controllable Set)**.** $\mathcal{K}_0(\mathcal{S}_s) = \mathcal{S}_s$ and $\mathcal{K}_N(\mathcal{S}_s) \triangleq Pre(\mathcal{K}_{j-1}(\mathcal{S}_s)) \cap \mathcal{S}, j \in \{1, \dots, N\}$.

The pre-image is a 1-step controllable set, $\mathcal{K}_1(\mathcal{S}_s)$.

Proposition 6.4 (Demonstration Membership of Controllable Sets of $\mathcal{S}_G$)**.** *Let s_k be the k-th state in the demonstration trajectory δ. For every $k = 0, \dots, |\delta| - 1$ and $L \geq 1$, $s_k \in \mathcal{K}_M(\mathcal{S}_G)$, where $M = L(|\delta| - 1 - k)$.*

Backwards Learning with Piecewise-defined Policies (BLPP) exploits Proposition 6.4, and aims to compute a *proper* policy π that maximizes expected reward, over an approximation of the set $\mathcal{K}_{L(|\delta|-1)}(\mathcal{S}_G)$ that contains all the states in δ. However, we need to approximate the controllable set $\mathcal{K}_{L(|\delta|-1)}(\mathcal{S}_G)$ as $T(s, a, s')$ is defined via a simulator and therefore we cannot compute the pre-images directly. Instead we use sampled trajectories from the simulator to create approximate controllable sets.

Definition 6.5 (Approximate N-Step Controllable Set)**.** Given a sample of m trajectories $\mathcal{T} = \{T_0, \dots, T_{m-1}\}$ generated by a policy π, with each trajectory T_i having N steps, $T_i = \{s_0, \dots, s_N\}$, $s_i \in \mathcal{S}$, we define the approximate N-step controllable set $\tilde{\mathcal{K}}_N$ of $\mathcal{S}_s \subseteq \mathcal{S}$ as

$$\tilde{\mathcal{K}}_N(\mathcal{S}_s) \triangleq \{x \in \mathcal{S} \mid x \in T_i, succ(T_i, \mathcal{S}_s), 0 \leq i \leq m - 1\} \tag{6.1}$$

6.3.1 Iterating over sequences of sub-policies

BLPP is a *policy iteration* algorithm that rather than iterating over *one single* policy π, it considers a set of $|\delta|-1$ sub-policies $\Pi_G = \{\,\pi^k \mid k = 0, \dots, |\delta|-2\,\}$. Each sub-policy π^k in BLPP is a valid policy for the Goal-MDP $\hat{M}^k = (\tilde{\mathcal{K}}_{\hat{h}^k}(G^k), s_k, G^k, A(s), T(s,a,s'), c^k(s,a,s'), \hat{h}^k)$, where the initial state is $s_k \in \delta$; $\hat{h}^k = L$ is the horizon; and the goal set $G^k = \bigcup_{i=k,\dots,|\delta|-3} \tilde{\mathcal{K}}_{\hat{h}^k}(G^{i+1})$ for all $k < |\delta| - 2$ is the union of the approximate controllable sets (Definition 6.5), with the exception of $G^{|\delta|-2} = \mathcal{S}_G$ which is set to the original goal states $\mathcal{S}_G$. The approximation of the controllable set $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ is computed, as a by-product of searching for $\pi^k \in \Pi_G$. That is, the sampled trajectories $\mathcal{T}$ used to create $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ (Definition 6.5) are generated by π^k.

Each $\hat{M}^k$ starts with a random policy π^k_0. Given π^k_0, BLPP creates an approximation of the controllable set $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)_0$. BLPP then solves $\hat{M}^k$ by iteratively constructing $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)_i$ and π^k_i until the policy is successful over states $s \in \tilde{\mathcal{K}}_{\hat{h}^k}(G^k)_i$ with a success rate greater than a given threshold R. Policy iterates π^k_i are constructed through an off-the-shelf policy optimization algorithm. In this work we use PPO (Schulman et al. 2017), but any direct policy optimization technique can be used. For every iteration i where π^k_i is computed to solve $\hat{M}^k$, $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)_i$ is updated with new trajectories generated by the policy.

Note that the goal set G^k of each Goal-MDP $\hat{M}^k$ is the union of the controllable sets $\tilde{\mathcal{K}}_{\hat{h}^k}(G^{k+1}), \dots, \tilde{\mathcal{K}}_{\hat{h}^k}(G^{|\delta|-2})$, hence $\hat{M}^{|\delta|-2}$ must be solved first. The MDP $\hat{M}^{|\delta|-2}$ is defined differently to the rest. First, $G^{|\delta|-2} = \mathcal{S}_G$, $\hat{h}^{|\delta|-2} = H$ and states within $G^{|\delta|-2}$ are not terminal unless they are terminal too in M^0. Therefore, $\pi^{|\delta|-2} \in \Pi_G$ is considered valid iff it reaches $\mathcal{S}_G$ within $l \leq H$ steps and stays within $\mathcal{S}_G$ for the remaining $H - l$ steps. This is akin to the definition of a successful trajectory (Def. 6.1). Once $\hat{M}^{|\delta|-2}$ is solved, we have computed $\langle \pi^{|\delta|-2}, \tilde{\mathcal{K}}_{\hat{h}^k}(G^{|\delta|-2}) \rangle$, and can solve $\hat{M}^k$ for $k = |\delta|-3, \dots, 0$ treating states in G^k as terminals.

In order to manipulate the sets $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)_i$ efficiently, we compute polytopes P^k_i in $\mathcal{H}$-representation (Def. 2.9) such that $P^k \supseteq \tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$. By representing $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ as P^k, we are assuming that any point within the convex hull of $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ is in the true L-step controllable set $\mathcal{K}_{\hat{h}^k}(G^k)$. That is, we assume any point in P^k reaches G^k within $\hat{h}^k$ steps when executing policy π^k. We define the approximate symbolic representation of $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ as:

$$P^k \triangleq conv(\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)) \tag{6.2}$$

From now on we use the notation P^k instead of $\tilde{\mathcal{K}}_{\hat{h}^k}(G^k)$ to emphasise that we represent the approximation of the controllable sets symbolically and implicitly in terms of the coefficients A and b that characterize P^k (Def. 2.8).

The cost function used for policy iterations of $\pi^{|\delta|-2}$ is,

$$c^k(s,a,s') = \begin{cases} -(\|s' - s_{|\delta|-1}\|_2 + \eta)^{-1} & \text{if } s' \in \mathcal{S}_G \\ \|s' - s_{|\delta|-1}\|_2 & \text{otherwise} \end{cases} \tag{6.3}$$

where $s_{|\delta|-1}$ refers to the last state in δ. This cost function rewards transitions into the goal set $\mathcal{S}_G$ and penalises ones away from the demonstration state. $\eta > 0$ is a constant added to avoid undefined values when $s' = s_{|\delta|-1}$. Otherwise when solving for $\hat{M}^0, \ldots, \hat{M}^{|\delta|-3}$ the cost function is,

$$c^k(s,a,s') = \begin{cases} -(\|s' - s_{k+1}\|_2 + \eta)^{-1} & \text{if } s' \in G^k \\ C + \|s' - s_{k+1}\|_2 & \text{if } s' \in \mathcal{S}_G \\ \|s' - s_{k+1}\|_2 & \text{otherwise} \end{cases} \tag{6.4}$$

where s_{k+1} is the next state in the demonstration δ, given that s_k is the associated demo state with $\hat{M}^k$. This cost function also penalises transitions away from the next demonstration state s_{k+1} and rewards transitions into the goal set G^k. The cost function applies a constant penalty C for transitions into $\mathcal{S}_G$ that are not in G^k, which is applied for $k < |\delta| - 2$ where $\hat{h}^k = L$. L is a hyperparameter and can be smaller than H. $\hat{M}^{|\delta|-2}$ uses $\hat{h}^{|\delta|-2} = H$ such that $\pi^{|\delta|-2}$ learns to get to and stay in $\mathcal{S}_G$ for H steps from $P^{|\delta|-2}$. Hence, we want π^k for $k < |\delta| - 2$ to get to a state in $P^{|\delta|-2}$ so that $\pi^{|\delta|-2}$ is executed to achieve a successful trajectory (Def.6.1). That is, we do not want a policy using a horizon $< H$ to avoid $P^{|\delta|-2}$ and get to $\mathcal{S}_G$ directly as it is not evaluated on being able to stay in the goal for H steps as $\pi^{|\delta|-2}$ is.

Given a demonstration δ for a Goal-MDP problem M, BLPP returns a set of policy-polytope pairs, Π_G. The execution of the global policy Π_G is defined as:

Definition 6.6 (Global Policy)**.** Given the set $\Pi_G = \{\langle \pi^0, P^0 \rangle, \ldots, \langle \pi^{|\delta|-2}, P^{|\delta|-2} \rangle\}$ and a state $s \in \mathcal{S}$, the global policy Π_G returns an action from the policy $\pi^j(s)$ closest to the goal $\mathcal{S}_G$, if $s \in P^j$ for any $j = [0, |\delta| - 2]$, otherwise Π_G returns $\pi^j(s)$ for

$$\underset{j}{\operatorname{argmin}} \|s - c(P^j)\|_2 \tag{6.5}$$

where $c(P^j)$ is the center of the Chebyshev Ball for polytope P^j, i.e., the center of the largest radius ball that fits within P^j. While $c(P^j)$ is not unique, computing the Chebyshev Ball for P^j is more efficient than other exact methods to find the interior points of a polytope (Borrelli, Bemporad, and Morari 2017).

Symbol	Description	Range
R	Evaluation success rate required.	$[0, 1]$
ϵ_0	Initial value for ϵ-bounding box.	$[0, \infty]$
m_V	Min. volume polytope required.	$[0, \infty]$
m_E	Min. trajectories for eval. and polytope creation.	$[1, \infty]$
H	Environment horizon	$[1, \infty]$
L	Horizon for each sub-policy	$[1, H]$
B_t, B_e, B_p	Simulator call budgets.	$[1, \infty]$
C	Constant used by cost function.	$[-\infty, \infty]$

TABLE 6.1: Hyperparameters for BLPP

6.3.2 Algorithm description

Algorithm 1 describes the BLPP algorithm, and a description of its hyperparameters are provided in Table 6.1. Algorithm 1 starts by initialising the parameters $k, G^k, \hat{h}^k$ in order to solve $\hat{M}^{|\delta|-2}$ (line 3). Where G^k is a set of polytopes to be treated as goal states in $\hat{M}^k$, for $k = 0, \ldots, |\delta| - 3$ and $\hat{h}^k$ is the horizon. We then proceed to solve each MDP, $\hat{M}^{|\delta|-2}, \ldots, \hat{M}^0$ in order (lines 4-23).

Solving $\hat{M}^k$ begins by initialising π^k, ϵ, E_i^k, b_p, and b_e (line 5-6). π^k is set to a random policy from the given set of possible policies Π. ϵ is used for ϵ-bounding box (Def. 2.12) operations. E_i^k is a set of extreme points from polytopes, $P_0^k, \ldots, P_{i-1}^k$, that have been created for $\hat{M}^k$:

$$E_i^k = \{e \mid (\exists j \in \mathbb{Z}^+)[e \in extremes(P_j^k), j < i]\} \tag{6.6}$$

The simulator budgets for creating a polytope, b_p, and for evaluating it, b_e, are set via hyperparameters m_e, B_p and B_e. The budgets are set to ensure that at least m_e trajectories (episodes) and at least B_p for b_p and B_e for b_e simulations are executed regardless the value of $\hat{h}^k$.

The policy iteration process for $\langle \pi^k, P^k \rangle$ on $\hat{M}^k$ (lines 7-20) is repeated until either the simulator budget is exhausted or the threshold requirements for the evaluation success rate R and the minimum polytope volume m_v are met. For each iteration the polytope P_i^k given π_i^k is computed. Initially, P_i^k is defined as β, an ϵ-bounding box (Def. 2.12) for the states s_k and s_{k+1} (line 8). Evaluation trajectories executing π_i^k on $\hat{M}^k$ are then simulated in the EVALB function (line 9).

Algorithm 4 Backwards Learning over Piecewise-defined Policies (BLPP)

```
1: Input: A demonstration $\delta = s_0, s_1, \ldots, s_m$
2: Output: $\Pi_G$ set of policy-polytope pairs
3: $\Pi_G \leftarrow \emptyset$, $k \leftarrow m - 1$, $G^k \leftarrow \emptyset$, $\hat{h}^k \leftarrow H$
4: repeat
5:   $\pi_0^k = sample(\Pi)$, $\epsilon \leftarrow \epsilon_0$, $E^k \leftarrow \{s_k, s_{k+1}\}$
6:   $r_d \leftarrow 0$ {also initialise budget vars $b_p, b_e$}
7:   for $i \leftarrow 0$ to $\infty$ do
8:     $\beta \leftarrow$ BOUNDING-BOX$(s_k, s_{k+1}, E^k, r_d, \epsilon)$
9:     $\tilde{S} \leftarrow$ EVALB$(\pi_i^k, s_k, \beta, G^k, \hat{h}^k, b_p)$
10:    $P_i^k \leftarrow conv(\tilde{S})$ {Create polytope}
11:    $r_p, r_d \leftarrow$ EVALP$(\pi_i^k, s_k, P_i^k, G^k, \hat{h}^k, b_e)$
12:    if $r_p \geq R$ then
13:      if VOLUME$(P_i^k) \geq m_v$ then
14:        break {Successful}
15:      end if
16:      $\epsilon \leftarrow 2 \times \epsilon$
17:    end if
18:    $E^k \leftarrow E^k \bigcup extremes(P_i^k)$
19:    $\pi_{i+1}^k =$ IMPROVE$(\pi_i^k, P_i^k, s_k, G^k, \hat{h}^k, B_t)$
20:  end for
21:  $G^{k-1} \leftarrow G^k \bigcup \{P_i^k\}$, $\Pi_G \leftarrow \Pi_G \bigcup \{\langle \pi_i^k, P_i^k \rangle\}$
22:  $\hat{h}^k \leftarrow L$, $k \leftarrow k - 1$ {Step back in demo}
23: until $k < 0$
```

EVALB runs the first trajectory starting from the demonstration state s_k, with the rest using starting states that are randomly sampled from the hyper-rectangle β. EVALB returns $\tilde{S}$ which is the set of states visited by successful trajectories (Def. 6.1). The convex hull of $\tilde{S}$ becomes the new polytope P_i^k (line 10). If $\tilde{S}$ is empty then P_i^k will also be empty. The convex hull is computed with the classic QuickHull algorithm (Barber, Dobkin, and Huhdanpaa 1996) [2]. We then evaluate the policy iterate $\langle \pi_i^k, P_i^k \rangle$ via simulation through the EVALP function (line 11). Half of these simulations start from the demonstration state s_k and the other half from the extremes of P_i^k. EVALP returns r_p and r_d. r_p is the success rate of all the trajectories that were run, i.e., both the trajectories starting from s_k and the extremes of P_i^k. r_d is the success rate of the trajectories that were run starting from s_k.

Lines 12-15 check if $\langle \pi_i^k, P_i^k \rangle$ are successful, defined as $r_p \geq R$ and VOLUME$(P_i^k) \geq m_v$. If $r_p \geq R$ but VOLUME$(P_i^k) < m_v$ we want the volume of P_i^k to increase in the future iterations therefore the ϵ used for the ϵ-Bounding Box operations is increased (line 16). Note that the VOLUME function uses a Monte Carlo method to estimate the volume of P_i^k with the implication that P_i^k may be misclassified as being below or above the volume threshold m_v

[2]We use github.com/tulip-control/polytope implementation

because of an approximation error. If successful, the policy iteration is stopped and BLPP steps back to the previous state in the demonstration (line 22). This continues until policies have been obtained for every state in the demonstration, and BLPP terminates returning Π_G. When stepping backwards to the previous state in the demonstration the goal set for the new Goal-MDP to be solved is updated (line 21) and the horizon $\hat{h}^k$ is set to L (line 22). Alternatively, if $\langle \pi_i^k, P_i^k \rangle$ do not meet the threshold requirements of R and m_v the set E^k is updated with the extreme points from P_i^k and IMPROVE, a policy improvement step, is run (lines 18-19). IMPROVE can be any direct policy optimization algorithm, and as noted earlier, in this work we use PPO. Half of the starting states used for the policy optimization algorithm are the demo state s_k and the other half are selected from the set $extremes(P_i^k)$.

The BOUNDING-BOX function returns the ϵ-bounding box for the set $\{x \in \mathbb{R}^d \mid x \in E_i^k\}$, unless $r_d = 0$ in which case the ϵ-bounding box for the demonstration states s_k and s_{k+1} is returned. As each iteration i only adds to the set E^k without ever removing any states, the bounding box of E^k will be monotonically increasing, such that as long as $r_d \neq 0$ the ϵ-bounding box β will also be monotonically increasing. When $r_d = 0$ we reduce the size of the bounding box so that the training effort of π^k can be focused on areas of the state space near the demonstration state s_k.

6.3.3 Single policy instance

We also propose a single policy version of BLPP, we name Backwards Learning using Polytopes (BLP). It works the same as BLPP but instead of creating a new policy when stepping backwards in the demonstration, BLP trains and evaluates a single policy. BLP does not use polytopes $P^1, \ldots, P^{|\delta|-2}$ as goal sets for the MDPs, instead it always has the final demonstration state $s_{|\delta-1|}$ as its goal state for the cost function. Hence BLP always uses the cost function previously defined in Equation 6.3. An evaluation trajectory for each step of BLP is only classified as successful if it reaches and stays in the goal set of the environment $\mathcal{S}_G$. Thus, BLP always uses the full environment horizon H, rather than L for the training and evaluation trajectories.

6.3.4 Complexity

The worst-case time complexity of BLPP is a function of the demonstration length $|\delta|$, the complexity of a call to IMPROVE denoted as ϕ, the maximum number of calls, N, to IMPROVE

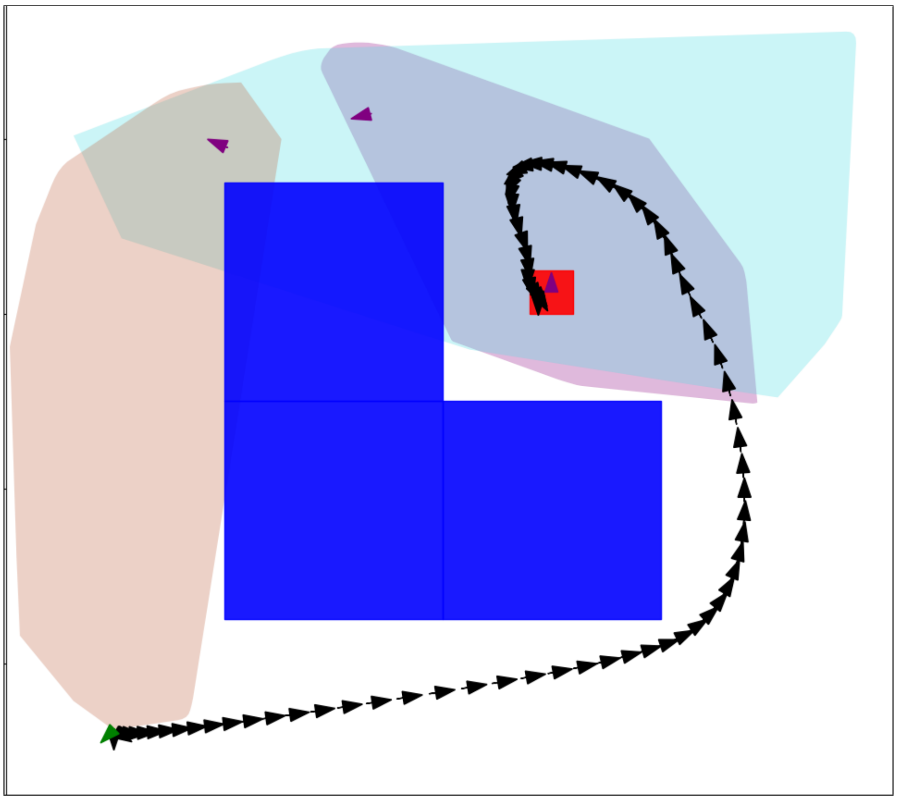


FIGURE 6.2: Mobile-Robot domain with an obstacle (L-shaped blue object) and the goal region (red square). The green arrow shows the initial starting state, where the direction of the arrow represents the initial velocity. The black arrows show a single human demonstration. The three purple arrows show a demonstration produced from RRT*. An example of the polytopes learnt by BLPP for the RRT* demonstration are projected from Mobile-Robot's 4-dimensional state space into their 2-dimensional representations (brown, light blue and purple blobs).

needed for each sub-policy π_i^k, and the dimension d of the state space $\mathcal{S}$. The worst-case time complexity for creating a convex hull and computing the extremes of a polytope is $O(n^{\lfloor d/2 \rfloor})$, where n is the number of states in the set the convex hull is being computed for (Sartipizadeh and Vincent 2016). In our case the set $\tilde{S}$ is used for creating the convex hull and $|\tilde{S}|$ is bounded by b_p which has a maximum value of $m = max(B_p, m_e \times H)$. Therefore the worst case time complexity of BLPP is $O(N|\delta|(\phi + m^{\lfloor d/2 \rfloor}))$ as BLPP needs to solve for $|\delta| - 1$ sub-policies with a maximum of N IMPROVE, $extreme$ and $conv$ function calls each. For a given Goal-MDP M, d is fixed and therefore the worst case complexity becomes $O(N|\delta|\phi)$.

In this work we test problems with up to 4 dimensions. For these problems the convex hull and extreme operations take 2%-30% of the compute time of BLPP. For higher dimensional problems an approximation method for the convex hull, like that introduced by Sartipizadeh and Vincent (2016), that is not exponential on the dimensions, may be required.

6.4 Experimental study

Here we compare BLPP and BLP with baseline algorithms on a number of different domains and demonstration types. As baselines we use a plain Backwards Learning (BL) algorithm as described by Salimans and Chen (2018), PPO, and the two imitation learning algorithms GAIL and BC [3]. We evaluate the algorithms across a number of goal-based control problems modelled as Goal MDPs.

Domains. We use the three instances of the Mobile-Robot which were introduced in 2.5.2. Additionally we use the OpenAI gym's *CartPole*, *MountainCar* and *MountainCarContinuous* domains. The only modification we make to these three environments is that we provide a function from which the environment returns if it is in a goal state. For the *CartPole* domain, the starting state is a goal position, that is the pole is balanced, and therefore for the agent to have a successful trajectory by our definition it needs to balance the pole for the whole time horizon which is 500 steps. The *MountainCar* problems require an agent to get to a goal state which is on top of a hill through either discrete accelerations, *MountainCar*, or continuous ones, *MountainCarContinuous*.

Demonstrations. We obtained human demonstrations for each of the domains providing manually the control for episodes until we had a successful trajectory.

We also test on algorithm-generated demonstrations. For the classic control domains we simply use demonstrations from the policies learnt from running PPO. For the *Mobile-Robot* domains however PPO does not produce policies that find any successful trajectories, instead we use the geometric planner RRT*(Karaman and Frazzoli 2011) as implemented in the OMPL library (Şucan, Moll, and Kavraki 2012). Holonomic RRT* assumes that any geometrically feasible trajectory is also dynamically feasible. One of the demonstrations generated by RRT* can be seen in Figure 6.2. It is clear when comparing the RRT* demonstration to the human one in Figure 6.2 that direct transitions between the states in the RRT* demonstration are impossible in a single time step and even having trajectories along a straight line between the states is infeasible. For instance, when the current state velocities are directed away from the next state in the geometric plan there is no single input that will cause the next state to be along the straight line between the two states. As the geometric plans provide only states and not actions and it is not possible to derive the actions from the plan as it can be dynamically infeasible, the GAIL and BC

[3]GAIL, BC and PPO are evaluated using the github.com/hill-a/stable-baselines implementations.

algorithms are not applicable. It should be noted that using versions of these algorithms that work only with observations, namely GAIfO (Torabi, Warnell, and Stone 2018b) and BCO (Torabi, Warnell, and Stone 2018a) is not possible as they aim to directly imitate the transitions s to s' of the demonstration, however, as explained these demonstrations provide state transitions that are not dynamically feasible. BL as described by Salimans and Chen (2018) is also not applicable to RRT* demonstrations as it relies on the accumulated cost of the demonstration for each demonstration state. BL uses the costs of the demonstration to test when the learnt policy is performing at the level of the demonstration so that it can change its training starting states to ones earlier in the demo. Hence, we create a modification of BL that measures the success of a trajectory as defined in Def. 6.1, as opposed to achieving an equal or better cost than the demonstration, we refer to this as BL*.

Evaluation and Testing Robustness. 5 unique seeded runs are used for each combination of algorithm, domain and demonstration. Once a run is complete its policy is evaluated with 100 episodes. Each domain has 3 human and 3 algorithmic demonstrations, except *Mobile-Robot* which has 1 algorithmic demonstration. Mobile-Robot has a single algorithmic demonstration as the geometric planner RRT* always produces the same plan which is simply the start and goal state as there are no obstacles between the two states. While we use the time horizon H associated with each domain for learning, for the evaluation of the policies we also use a time horizon of 4 times H. We use the longer horizon to evaluate how the learnt policies adapt to a change in episode time duration. For instance on environments that need to stay within the goal set, evaluating on the longer horizon tests if the policies have generalised to stay in the goal set for extended periods.

BLPP's sub-policy horizon L can be optimized using a grid search. However, due to computational constraints we set L intuitively depending upon the demonstration. For the human and PPO trajectories we use $L = 5$. We select this value as the human and PPO demonstrations provide complete dynamically feasible trajectories, that is for $s_k \in \delta$ we have evidence that $P(s_{k+1}|s_k) > 0$ hence we assume that 5 steps will suffice for reliably reaching the state space containing and surrounding s_{k+1}. For the RRT* demonstrations we know that in the worst case they will contain only the start and goal state so we simply use $L = H$. An example of the polytopes learnt by BLPP given $L = H$ can be seen projected down from 4 to 2 dimensions in Figure 6.2.

The learnt policies for the *Mobile-Robot* domains are also evaluated on versions of the domains

Domain	Type	BLPP	BLP	BL	BL*	GAIL	BC	PPO
M-Robot	H	5.2	0.8	3.5	1.6	18.6	<0.1	2.5
	R	0.3	0.7	NA	3.9	NA	NA	
M-Robot Ob	H	6.7	1.3	6.9	7.3	18.5	<0.1	2.5
	R	0.6	5.1	NA	7.3	NA	NA	
M-Robot-Ob-Stay	H	10.8	9.0	5.3	7.6	19.1	<0.1	2.5
	R	1.5	7.4	NA	7.4	NA	NA	
CartPole	H/P	8.3	5.9	1.8	2.6	15.1	<0.1	2.2
M.Car	H/P	0.6	0.4	5.3	5.5	14.8	<0.1	2.3
M.Car Cont.	H/P	0.8	1.5	2.1	0.2	14.6	<0.1	2.2

TABLE 6.2: Average training time (in hours) across different trials and demonstrations given the domain and demonstration types Human(H), RRT*(R) and PPO(P).

where random disturbances $w_x = \mathcal{N}(0, \sigma_x^2)$ and $w_y = \mathcal{N}(0, \sigma_y^2)$ are added to the velocities v_x and v_y respectively after the dynamics have been calculated at each time step. We do this to evaluate the robustness of the policies learnt and refer to these evaluation instances as domains with *noise*. For the evaluation of Mobile-Robot domains with noise we use $\sigma_x = \sigma_y = 0.1$. However, for the stay in goal with obstacles version we used $\sigma_x = \sigma_y = 0.05$ as there were no successful evaluation episodes with the longer horizon with $\sigma_x = \sigma_y = 0.1$ for any algorithm.

We evaluate each algorithm by executing the runs in parallel for each combination of trial, demonstration and domain on a computing instance of 80 Intel Xeon 2.10GHz processors with 720GB of RAM. We allow a maximum of 80 runs to be executed in parallel. The polytope operations and simulator calls execute as a single process but the PPO steps from the baselines repository execute across multiple processes. Each algorithm, domain and demonstration combination is run with 5 unique seeds. PPO runs do not use the demonstration so 15 unique seeded runs were used for each domain. Table 6.2 shows the average learning time for each algorithm across the different demonstrations and trials. Note that BC's average run time was around 30 seconds as it does not require any simulator calls.

6.4.1 Hyperparameters values for each algorithm

PPO, BLPP, BLP, BL, and BL* use the same policy network architecture of 2 fully connected hidden layers with 64 units each using tanh activation functions. BC and GAIL use the same network architecture but each layer has 100 units instead of 64 to match that used by Ho and Ermon (2016).

PPO

Number of steps per batch, B_t, = 3000, *entropy coefficient* = 0.0, *VF coefficient* = 0.5, *clipping parameter* = 0.2, *maximum value for the gradient clipping* = 0.5, *minibatch size* = 1, *learning rate* = 0.01, *GAE parameter* (λ) = 1.00, *number of epochs for optimising surrogate* = 5.

BLPP/BLP

For PPO steps we use the same hyperparameters as used for the PPO runs. $R = 0.99$, $\epsilon_0 = 0.01$, $m_v = 0.0001$, $m_E = 10$, $L = H$ for RRT* demonstrations and $L = 5$ for human and PPO demonstrations, $B_t = 3000$, $B_e = 100$, $B_p = 3000$, $C = 100$. We lack the computational resources to conduct a full grid search across all parameters and across the full benchmark set. Instead hyperparameters were tuned manually using a single run on a single demonstration (different to any demonstration used for the results) on the WalkBot domain, note that the selected values were used across the entire benchmark set. Table 6.3 shows the values tested, note that each value was tuned individually such that not every combination was tested.

Symbol	Values Tested
B_t, B_p	1000, 3000, 10000
m_V	0, 1e-4
m_E	0, 10, 100
B_e	10, 100, 1000, 3000
ϵ_0	0, 0.01
R	0.5, 0.8, 0.9, 0.99
Learning rate	0.0001, 0.001, 0.01, 0.1

TABLE 6.3: BLPP hyperparameters values that were tested.

BL/BL*

We use the same hyperparameters for the PPO steps as BLPP/BLP, that is the same hyperparameters used for the PPO runs. *Success rate of training episodes required to step back* = 0.99, *step size* = 1, *training epoch budget* = 3000.

BC

70% of the demonstration data is used for training and 30% for validation, in line with Ho and Ermon (2016). Training stops once the latest 50 epochs have a validation error higher than the previous 50 epochs we tuned this manually on a single run with a single demonstration using $[1, 10, 50, 100, 500]$ epochs. We also used the default *learning rate* = 10^{-4} and *adam epsilon* = 10^{-8} from the stable-baselines library (Hill et al. 2018).

GAIL

In line with Ho and Ermon(2016) we use, *the number of steps per batch* = 5000, *GAE parameter* (λ) = 0.995, γ=0.995, *entropy coefficient* = $1e-3$, adversarial network architecture is also a fully connected network with 2 layers with hidden units of 100 each and tanh activation functions.

Domain	Type	BLPP	BLP	BL	BL*	GAIL	PPO
M-Robot	H	4.1	**0.5**	5.2	3.8	10	10
	R	**0.6**	0.8	NA	10	NA	
M-Robot-Ob	H	5.3	**1.0**	9.9	9.5	10	10
	R	0.9	**0.8**	NA	10	NA	
M-Robot-Ob-Stay	H	**7.7**	9.9	**7.7**	9.2	10	10
	R	**1.9**	9.5	NA	10	NA	
CartPole	H/P	4.3	5.7	**3.5**	5.9	10	10
M.Car	H/P	0.9	**0.8**	9.5	10	10	10
M.Car Cont.	H/P	0.9	2.1	3.9	**0.3**	10	10

TABLE 6.4: Average simulator calls (in millions) across different trials and demonstrations given the domain and demonstration types Human(H), RRT*(R) and PPO(P). The maximum number of simulator calls allowed is 10 million.

6.4.2 Comparison with Chapter 5's experimental set up

While Chapter 5 also used the Mobile-Robot-Obstacles-Stay (referred to in Chapter 5 as MROS) there are a few key differences in the evaluation of the algorithms and the metrics reported in this chapter. Most importantly in this chapter we measure the percentage of successful trajectories (Definition 6.1), rather than the percentage of trajectories which match or exceed the demonstration's score. This means for the MROS domain in this chapter the agent once in the goal region must stay there until the problem horizon has been reached. In the previous Chapter as long as the agent was in the goal region for an equal or greater amount of steps than the demonstration trajectory it was counted as being successful. Additionally, for the MROS with noise evaluation, the previous Chapter also trained the policy on the MROS domain with noise while in this chapter we evaluate the policies trained on the MROS without noise domain to test the robustness of the learnt policies. As discussed in the previous section we also test the robustness of policies learnt by BLPP in this chapter by evaluating them with an increased problem horizon.

6.4.3 Results

Figure 6.3 summaries the results by providing the confidence intervals and averages over every combination of trail and domain for each algorithm. Additionally Appendix A provides figures that separate out the performance of each algorithm over the different demonstrations used. Figure 6.3 shows that, across the 6 domains, BLPP and BLP are the best overall performers when compared to BL, BL*, BC, GAIL and PPO. BLPP's performance is superior to that of BLP as the complexity of the behaviour required increases. This increase of complexity follows from

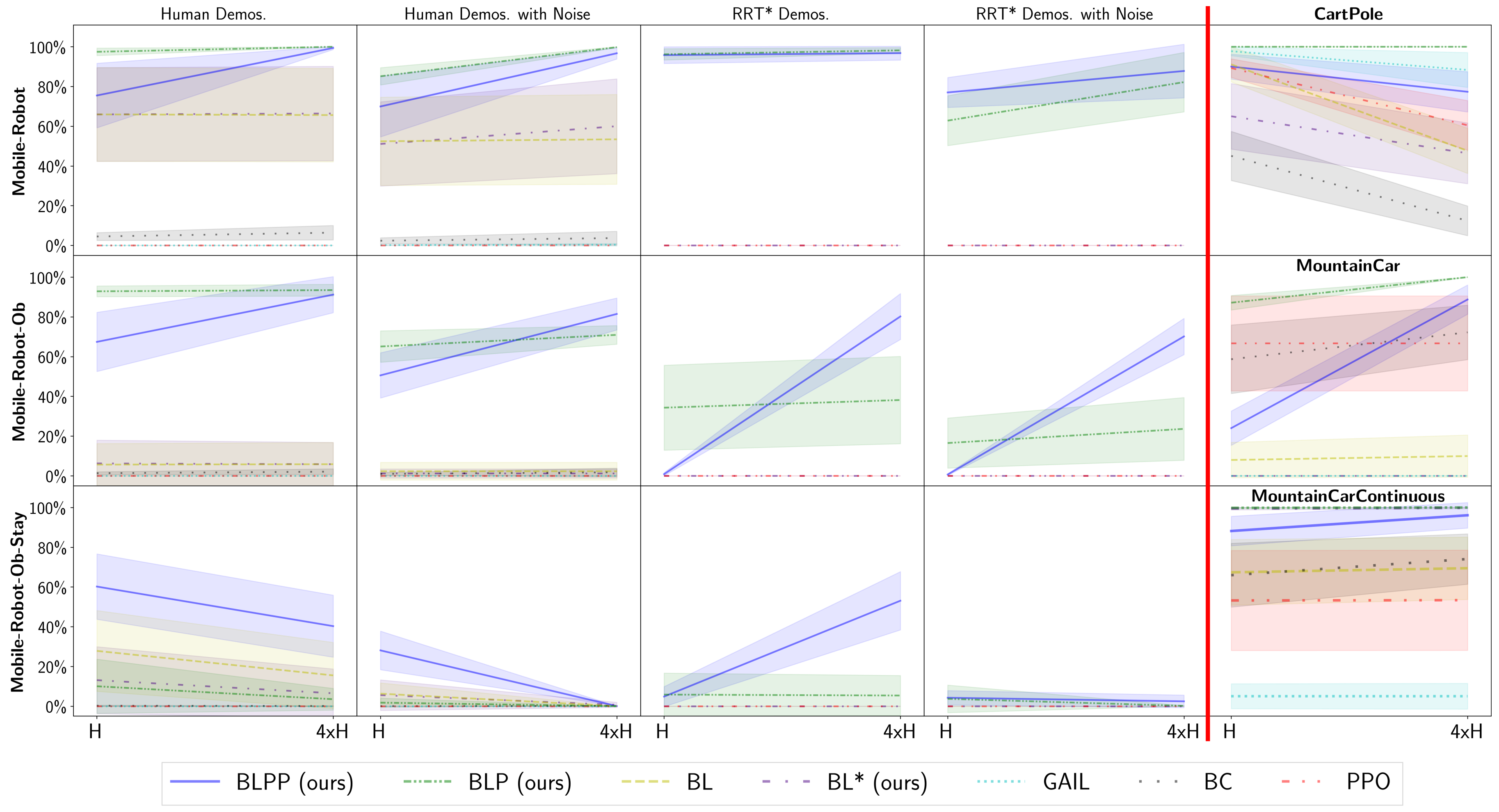


FIGURE 6.3: Success rate of evaluation episodes vs evaluation horizon, where H is the horizon used for demonstrations and learning. The Mobile-Robot domain results (left of red line) compare the average performance of human vs RRT* demos and evaluation with and without noise. The classic control domains (right) are the average values across all demos and trials. Shading represents 95% Confidence Intervals.

three causes. First, demonstrations can be incomplete, such as those provided by the holonomic RRT* planner. Second, obstacles require the policies to be defined over sets of states which are no longer convex. Third, requiring that the agent *both* reaches and stays within the goal requires the policy to deal effectively with stochastic perturbations. While the Classic Control benchmarks exhibit at least one of the three characteristics above, none of them features all three simultaneously. Hence we observe BLP to perform better than BLPP, as the effort to construct the decomposition does not pay off.

Table 6.4 shows the average sample complexity across the different domains for each algorithm. BLPP or BLP use the least simulator calls on average for every domain except two of the classic control domains, *CartPole* and *MountainCarContinuous*, where BL and BL* require fewer samples. BC is not shown in Table 6.4 as it does not require simulator calls. As shown in Table 6.2 BLPP on average requires about half the run-time consumed by GAIL and around double the time used for PPO.

6.5 Discussion

Our work has strong links to that from Rosolia and Borrelli (2017) where through Model Predictive Control (MPC) they iteratively design safe termination sets with associated termination costs. BLPP uses this idea of computing termination sets to use in future learning iterations but approximates them with polytopes due to the constraints on the availability of precise descriptions of dynamics and cost functions that are typical in RL literature.

The fact that BLPP works with partial plans means it could be used in a scenario where a demonstration is provided only for the difficult parts of a problem. Alternatively, an expert could simply select states that are key points of interest for BLPP to learn from. BLPP could also learn from demonstrations produced on abstractions of a domain that are easier to solve, like we demonstrated with BLPP learning from the RRT* demonstrations.

BLPP uses convex hull and vertex enumeration operations which are intractable as they can be exponential on the dimensions of the state space. Therefore BLPP may be infeasible for problems with high-dimension state spaces. Future work could investigate alternative representations of controllable sets with BLPP in order to feasibly handle high-dimension state spaces.

It is possible that BLPP can be used in a "learning from pixels" pipeline as the output layer of a Deep Generative Model (DGM) (2016) maps input images into a high-dimensional feature space, and the MDP M^0 can have its state space defined as a set of feature vectors. It would require further investigation to determine how to best bootstrap the DGM "pre-processing" step and to analyze the trade-offs between specializing the DGM on images that follow from demonstrations, or using a more general model.

Other avenues for future work include expanding BLPP to work with multiple demonstrations and exploring whether BLPP's learned sub-policies generalise to modified domains with different goal sets or obstacles but where some of the behaviour from the original domain is still required.

6.6 Conclusion

Through introducing and analysing a new approach for learning from a single demonstration by learning backwards we addressed **RQ2**. BLPP as opposed to previous methods works with complete and incomplete demonstrations while also not requiring access to the environment's cost function, the demonstration's actions or the demonstration's rewards. We have shown empirically the value of representing controllable sets symbolically through polytopes in order to segment the state space to define piecewise policies and to select starting states for policy iterations. We evaluated BLPP with human demonstrations as well as demonstrations generated by a planner on a relaxation of the environment. The relaxation resulted in demonstrations generated by the planner that are dynamically infeasible. Despite this, BLPP was able to successfully learn from both the human and planner generated demonstrations.

Part III

Learning heuristics through symbolic regression

Introduction to Part 3

This Part follows up from parts 1 and 2 to address **RQ3** by introducing a method that uses symbolic regression to learn cost-to-go approximations. Instead of being provided demonstrations, as done in Part 2, we introduce a regression-based planning method that generates demonstrations automatically that are then used to learn cost-to-go approximations. Once the cost-to-go approximations are learnt from the demonstration trajectories, similar to Part 1, we use the cost-to-go approximations within a GBFS algorithm to analyse their usefulness. Our experimental study shows that the symbolic regression method we introduce learns more useful approximations, in terms of the coverage achieved when used to guide a GBFS, than previous learning methods while using also using a fraction of the training compute.

Chapter 7

Learning heuristics via pre-images[1]

7.1 Introduction

Heuristics for automated planning can be formulated following a number of approaches. Heuristics such as Fast-Forward (FF) (Hoffmann and Nebel 2001), $h^{\max}$ and h^{add} (Bonet, Loerincs, and Geffner 1997) follow from analyzing the structure of many planning instances, and coming up with a mathematical framework to automatically compute functions that capture important structural information about instances from the symbolic descriptions of causal laws (actions) and domain constraints (Helmert 2006). Alternatively, Pattern database (Edelkamp 2014) and *merge-and-shrink* (Helmert et al. 2007) heuristics are defined as generic functions that evaluate states by projecting them onto many smaller sub-problems, that are solved optimally, and combining their solutions in some specific way. These are chosen on a per-instance basis, and typically involve solving a discrete optimization problem to select which projection is deemed to be most informative according to some suitably defined criterion. A third approach has attracted attention recently, where heuristic functions are searched for in a family of functions described by a NN (Ferber et al. 2021; Ferber, Helmert, and Hoffmann 2020; Shen, Trevizan, and Thiébaux 2020). These efforts are mainly driven by the suggestive results in Computer Vision and RL of novel, general, and scalable stochastic algorithms for convex optimization to select NN parameters that minimize a suitably defined notion of *empirical risk* (Kingma and Ba 2015; Goodfellow, Bengio, and Courville 2016; Hardt and Recht 2021). Currently, the most successful methods for

[1]This chapter is adapted from the article "Sampling from Pre-Images to Learn Heuristic Functions for Classical Planning" published as an extended abstract in the proceedings of Fifteenth International Symposium on Combinatorial Search 2022.

learning NN heuristics for classical planning problems require vast amounts of computational resources for training and are usually outperformed by heuristics that do not require any offline training time (Ferber et al. 2021).

In this chapter, we are concerned with learning per-instance NN defined heuristic functions. The idea of per-instance learning is introduced in 2.9.5 and we note that the learning of per-instance heuristics is equivalent to the set up of the per-instance pre-computation required by pattern databases (Sievers, Ortlieb, and Helmert 2012), and cost partitioning heuristics (Seipp, Keller, and Helmert 2017). The new method we introduce for per-instance NN defined heuristic function learning is *Regression based Supervised Learning* (RSL). Like other methods (Ferber et al. 2021; Yu, Kuroiwa, and Fukunaga 2020) do, RSL selects a set of *regressions*, trajectories of *sets of states*, or *pre-images*, found via the application of well-known and efficient pre-imaging operators (Rintanen 2008) which rely on symbolic action descriptions. These trajectories found along a given regression always start from the set of goal states of an instance and then training states are sampled from each set along the trajectories. Our method takes many samples from each pre-image found in a regression, instead of performing many trajectories or longer regressions to increase the number of training states for the NN, using the observed goal distances for each pre-image to label the sampled states.

Through a sensitivity analysis over RSL's hyper-parameters we explore the impact of its key mechanisms. We also benchmark the NN heuristics learnt by RSL against existing methods to provide insight into the ability of RSL to learn effective heuristics. The Chapter's contributions are threefold: (1) introducing RSL, a new method for training NN defined heuristic functions, (2) introducing a new novelty measure for a regression based search, and (3) an experimental analysis of RSL's hyper-parameters and components.

7.2 Background

In this chapter we explore learning Heuristics for classical planning problems defined through the STRIPS formulation introduced in 2.2.1. In particular, we focus on per-instance heuristics which are introduced in 2.9.5.

The objective of the heuristics we introduce in this chapter is to approximate the optimal value function V^* as described through the Bellman optimality equation (Bellman 1957),

$$V^*(s) = \min_{a \in A(s)} \left[c(a,s) + V^*(f(a,s)) \right] \tag{7.1}$$

for all states $s \notin S_G$ and $V^*(s) = 0$ for $s \in S_G$. Similar to many of the existing heuristics such as h^{FF} the heuristics introduced in this chapter are not guaranteed to be lower or upper bounds on the optimal value function.

As explained in 2.2.1 for the progression state-transition model every atom not in a state is false, while for the regression state-transition model the atoms not in a state are simply undefined. From this point forward we refer to such "partial states" as *pre-images* and denote them as x to distinguish them from complete truth-assignment state denoted as s.

In this chapter we take advantage of the pre-images explored by a regression search in order to sample a set of training states labelled with approximations $\tilde{V}$ of V^* (Equation 2.2), to learn a heuristic defined through a NN using a Supervised Learning (SL) algorithm.

The SL approach is introduced in 2.4, and in this chapter we use SL to find a unique set of NN parameters $\theta \in \mathcal{H}$, where $\mathcal{H}$ is a set of NN parameters using states s as training examples $\bar{x}_i$ and cost-to-go estimates $\tilde{V}$ between s and S_G as their labels y_i. That is, our objective in this work is to efficiently, in terms of computation time, create a data sample of states labelled with $\tilde{V}$ such that an optimisation of the NN parameters θ using Equation 2.4 generalises to the population set of states s_0^i reachable from s_0 .

7.3 Comparisons with Related work

The algorithm we introduce in this work differs from both TSL and SING that were introduced in 2.9.5 in a number of ways. First, RSL samples training states through a regression over partially assigned states starting from the goal rather than a DFS starting at a fully assigned goal state (SING) or a forward search from the instance's initial state (TSL). Second, RSL additionally samples random states that are added to the training set. Third, for goal distance estimates TSL uses a teacher planner and SING uses the depth at which the state was visited while RSL uses the tightest upper bound found for the state's goal distance derived through the pre-images visited by a number of regressions. Last, as RSL searches and labels goal distance estimates over pre-images, and samples many different training states from each pre-image.

Ferber et al. (2021) introduced three RL inspired NN heuristics, h^{Boot}, h^{BExp}, and h^{AVI} heuristics which are described in 2.9.5. Ferber et al.'s motivation for these approaches versus SL ones is that SL is limited to instances small enough for training data generation. However, we show through the Experimental Study in this chapter that our SL approach scales better to larger instances than these approaches, while using 2 orders of magnitude less training time, including the time required for data generation.

Beyond learning per-instance NN defined heuristics, as discussed in 2.9.5 Shen et al.'s (2020) STRIPS Hypergraphs Networks (HGNs) learns per-domain and even domain independent NN defined heuristics. In this work we do not focus on domain independent NN defined heuristic functions, but we do compare against a STRIPS-HGN heuristic trained in a per-domain framework as described by Ferber et al. (2021).

In this chapter we also explore a version of the RSL algorithm that aims to maximise the structural diversity of the states in its selected sample. Width-based planning introduced in 2.7.5 is one method that has been particularly effective in increasing the structural diversity of states considered during search (Lipovetzky and Geffner 2012; Lipovetzky, Ramírez, and Geffner 2015; Frances et al. 2017; Katz et al. 2017). Inspired by the success of the width-based novelty algorithms in classical planning, both in progression and regression (Lei and Lipovetzky 2021), we create and experiment with a new novelty based regression algorithm.

7.4 Regression based Supervised Learning

Given a planning problem $\Pi = \langle F, O, I, G\rangle$, RSL produces a training set $\mathcal{D} = \{(s_1, h_1), \ldots, (s_N, h_N)\}$ which is a set of states $s \in S$ paired with goal distance estimates h. To produce $\mathcal{D}$ RSL performs N_r rollouts, each starting at the goal G and applying L times the classical planning regression operator. Each rollout from G is a sequence of actions $\pi^j = (a_i^j)_{i=0}^{L-1}$, for $j = 1, \ldots, N_r$. Each π^j produces a sequence of pre-images $x_0^j, x_1^j, \ldots, x_L^j$, where $x_0^j = G$ and $x_i^j \subseteq F$. The sequence of pre-images denotes a sequence of sets of states $\mathcal{R}^j = X_0^j, X_1^j, \ldots, X_L^j$, where $X_i^j = \{s \mid x_i^j \subseteq s, s \in S\}$. See Figure 7.1 for an example of this mapping of a pre-image represented by a partial truth assignment into its corresponding set of fully assigned states. By the definition of the regression operator, X_{i-1}^j corresponds with the *pre-image*, conditioned on

a^j_{i-1}, of X^j_i. Therefore any state within X^j_i can be reached from X^j_{i-1} by applying action a^j_{i-1}. It follows that any state within X^j_i can reach X^j_0 in at most i transitions. Using this observation, RSL labels each state s within its training set of states, T_s, with

$$d(s) = \min(i \mid s \in X^j_i, j \in \{1, \dots, N_r\}) \tag{7.2}$$

that is, the smallest goal distance estimate of any of the state sets visited by the regressions which s is also a member of.

```
Algorithm 5 Overview of the RSL Algorithm
Input: $\Pi$
Parameter: $P_r, L, N_r, N_t, \mu$
Output: $h^{\text{RSL}}$
1: $\mathcal{R} \leftarrow \text{REGRESSION}(\Pi, L, N_r, \mu)$
2: $T_s \leftarrow \text{SAMPLE\_STATES}(\mathcal{R}, P_r, N_t)$
3: $\mathcal{D} \leftarrow \text{LABEL}(\mathcal{R}, \mathcal{T}_f)$
4: $h^{\text{RSL}} \leftarrow \text{SUPERVISED\_LEARNING}(\mathcal{D})$
```

Algorithm 5 provides an overview of RSL. The hyper-parameters of RSL are the length of each regression L, the number of regressions to perform N_r, the number of training states to use N_t, the percentage of training states that are randomly sampled from the entire state space P_r, and a function that maps state regression trajectories into novelty levels μ. RSL has three distinct steps, 1: extracting sets of state sets, $\mathcal{R} = \bigcup_{j=1}^{N_r} \mathcal{R}^j$, through performing regressions from the goal set (Alg. 5 line 1), 2: sampling training states T_s and labeling them with goal distance estimates using (7.2) (Alg. 5 lines 2-3), and 3: training the NN defined heuristic function (Alg. 5 line 4) using empirical risk minimisation (2.4).

7.4.1 Extracting state sets through regression

As previously explained RSL performs N_r regressions to extract the set of state sets $\mathcal{R}$ over which the training set $\mathcal{D}$ is defined. At step i of a rollout with a pre-image x^j_i corresponding to the set of states X^j_i, as previously defined, the actions we consider valid for pre-imaging X^j_i with are $a \in \nu(x^j_i)$, defined as follows

$$\begin{aligned} \nu(x^j_i) = \{a \mid\ & a \in \text{REACHABLE}(O, I), \\ & x^j_i \cap \text{e-Del}(a) = \emptyset, \\ & x^j_i \cap \text{Del}(a) = \emptyset, x^j_i \cap \text{Add(a)} \neq \emptyset\} \end{aligned} \tag{7.3}$$

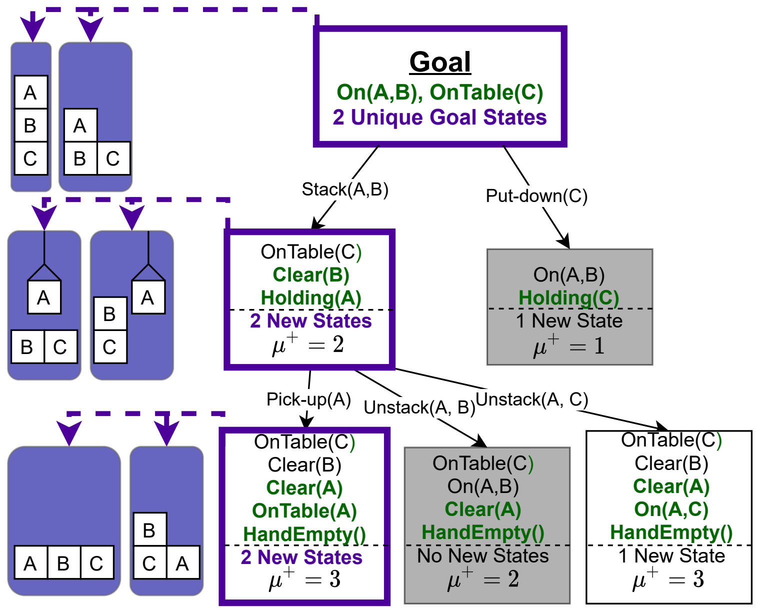


FIGURE 7.1: An example rollout performed by N-RSL with $L = 2$ on a 3 block Blocksword problem. Each square represents a per-image through a partially assigned state with its assigned atoms written within the square. The purple squares represent the example rollout trajectory selected by N-RSL and the possible Blocksworld states for each pre-image along the trajectory are shown on the left. Green atoms are atoms assigned for the first time in the trajectory, and the μ^+ value is calculated using Equation 7.6.

and e-Del(a) is

$$\begin{aligned} \text{e-Del}(a) = \{\, q \mid q \in F \setminus Add(a), \\ \exists p \in Pre(a) : \textsc{mutex}(p,q)\}. \end{aligned} \tag{7.4}$$

REACHABLE(O, I) maps the operator set O and initial state I to the set of actions with reachable preconditions in the delete relaxation of Π (Bonet and Geffner 2001) given the initial state of the progression state-transition model I. The MUTEX(p, q) function in (7.4) maps the pair of atoms (p, q) to true if p and q are mutually exclusive, that is, it is impossible for p and q to both be true in any state $s \in S$ that can be reached from I. Note that the Ferber et al. (2021) algorithms that use regression also filter the valid actions in the same way through using the mutex groups and applicable operations found by the Fast Downward (FD) (Helmert 2006) Translator[2].

The baseline option for performing the rollout, and the method used by Ferber et al. (2021), is to randomly select actions a for which applying the regression operator is valid. In addition to testing RSL using random action selection we instantiate a version of RSL we name Novelty

[2]See the IS_VALID_WALK parameter in line 848 in code/src/search/task_utils/sampling_technique.cc in Ferber et al.'s (2021) Supplementary Material available at `https://zenodo.org/record/5345958`

guided Regression based Supervised Learning (N-RSL) that aims to increase the structural diversity of operators selected in its regression.

Preferring actions with novel preconditions

For N-RSL at step i of a rollout with a pre-image x_i^j, and a state trajectory of $\tau = G, x_1^j, \ldots, x_i^j$, N-RSL uses the regression policy μ to select the action a_i to pre-image X_i^j.

$$a_i = \text{argmax}_a \{\mu(a, \tau) \mid a \in \nu(x_i^j)\} \tag{7.5}$$

where ties are broken randomly and $\nu(x_i^j)$ is a set of valid actions as described above. Note that for the plain RSL algorithm, which uses random action selection, $\mu = \mu(a, \tau) = 0$ for any $a \in O$ and any state trajectory τ.

In order to increase the structural diversity of the states in sets X_i^j in $\mathcal{R}^j$ we consider how the pre-images x_i^j evolve as we repeatedly apply the regression operator. At each step i of the regression we have a set of valid operators $\nu(x_i^j)$ and a state set X_i^j derived from the pre-image $x_i^j \subseteq F$, where $x_0^j = G$. We propose increasing the structural diversity of the sets extracted by counting the number of atoms an action will assign as true in a pre-image for the first time in the current trajectory. In order to apply this criterion, μ in (7.5) is replaced by μ^+ below,

$$\mu^+(a, \tau) = |Pre(a) \setminus \bigcup_{x_i^j \in \tau} x_i^j| \tag{7.6}$$

This μ^+ measure means N-RSL prefers actions with pre-conditions which contain atoms that are not specified in the goal G and are not a member of the pre-condition set of any of the actions executed in the trajectory up until that point. Note that μ^+ relies only on the current trajectory and is independent of any previously executed regression trajectories. Figure 7.1 shows an example of a rollout performed on a simple Blocksworld problem with the one-step lookahead method described by (7.5) using the novelty-based measure μ^+ defined in (7.6).

7.4.2 Sampling and labeling training data

The training data for RSL is sampled from the sets of states $\mathcal{R}$. States are sampled from each set $X_i^j \in \mathcal{R}$, and sampled states that contain mutex atom pairs (p, q) are modified by removing either the p or q atom from the state. Note that if a sampled state from the set X_i^j has a mutex pair (p, q) and $p \in x_i^j$, q will be removed from the state, that is, an atom that is a member of the partial state x_i^j that a state is sampled from will never be removed from the state. As previously described, the labelled heuristic value for a sampled state is given by $d(s)$, which returns the distance of the closest state set in $\mathcal{R}$ to the goal according to the regression.

As we report later, some domains benefited from adding randomly sampled states from S. The random states have mutexes enforced using the same method as above, and are also labeled with $d(s)$. In the case that the state is not a member of any of the sets of states in $\mathcal{R}$, we define $d(s)$ to equal $L + 1$.

Theorem 2. Given a problem $\Pi = \langle F, O, I, G \rangle$, N-RSL obtains a training set $\mathcal{D}$ in $O(|F|(|O|N_r L + N_t L N_r))$ time and $O(|F|(N_r L + N_t))$ space.

Proof sketch. RSL performs N_r rollouts and by definition the number of state transitions in each π^j is L. For each state transition the one step lookahead (Equation 7.5) considers a maximum of $|O|$ actions and $|O|$ pre-images. Therefore the total number of pre-images considered by RSL is $|O|N_r L$. Note that both L and N_r are set as constant hyper-parameters and for each pre-image a maximum of $|F|$ atoms need to be considered by the novelty definition μ^+, therefore the REGRESSION algorithm has a time complexity of $O(|F||O|N_r L)$. Also for each pre-image a maximum of $|F|$ atoms need to be assigned, hence REGRESSION runs in $O(|F|N_r L)$ space. Sampling states requires the assignment of at most $|F|N_t$ atoms and checking a state's membership requires at most $|F|N_t L N_r$ comparisons meaning both the SAMPLE_STATES and LABEL algorithms run in $O(|F||O|N_r L)$ time and $O(|F|N_t)$ space. □

Given the training set, $\mathcal{D}$, any suitable off-the-shelf SL algorithm can be used to obtain a heuristic estimator through optimising the SL objective function (2.4). We discuss the specifics of one such algorithm in our Experimental Study.

7.5 Experimental study

Our experimental study of RSL has two objectives: (1) evaluate the effect of the different mechanisms within the RSL algorithm, and (2) provide a direct comparison to existing NN and model-based heuristic functions.

	$h^{\textbf{Boot}}$			$h^{\textbf{BExp}}$			$h^{\textbf{AVI}}$			$h^{\textbf{TSL}}$		
	Ave	**Min**	**Max**	**Ave**	**Min**	**Max**	**Ave**	**Min**	**Max**	**Ave**	**Min**	**Max**
blocks	1.2	0.7	5.0	-	-	-	-	-	-	2.3	0.4	6.8
depot	1.2	0.8	1.4	1.2	1.0	1.5	1.2	0.9	2.5	3.9	0.7	9.9
grid	3.1	0.6	43.8	1.3	0.5	12.7	15.8	0.7	79.8	13.2	4.2	80.5
npuzzle	1.1	0.8	2.3	-	-	-	1.5	1.0	2.5	-	-	-
pipesworld	1.4	0.3	8.0	1.4	0.7	10.8	1.4	0.3	19.7	7.8	0.8	82.6
rovers	14.6	0.9	70.1	15.3	0.9	72.3	16.1	0.9	85.6	13.7	0.3	43.4
scanalyzer	1.1	0.1	23.2	0.3	0.03	0.9	1.2	0.1	9.5	8.6	0.2	45.8
storage	1.5	0.8	4.5	1.3	0.8	2.0	1.2	0.8	1.6	3.3	1.0	6.4
transport	1.3	0.8	3.8	1.1	0.9	1.7	1.0	0.7	2.9	5.0	0.6	11.2
visitall	0.9	0.4	1.5	-	-	-	-	-	-	-	-	-

TABLE 7.1: Ratios of evaluations per second of our h^{RSL} versus the baseline methods run by Ferber et al. (2021) for commonly solved instances.

7.5.1 Methodology

Our benchmark set of domains, instances and initial states is the same as those used by Ferber et al. (2021). As we are learning per-instance heuristics, a unique heuristic is trained for each problem instance and then evaluated over a set of 50 different initial states. The 50 initial states were produced by Ferber et al. (2020; 2021), through performing 50 200-step random-walks from the original initial state of the instance. The benchmark instances have also been separated by Ferber et al. (2020; 2021) into "Moderate Tasks", which are solved by GBFS guided by h^{FF} in less than 900 seconds but more than 1 second, and "Hard Tasks" which are not solved within 900 seconds.

We evaluate h^{RSL} heuristic using the same method as Ferber et al. (2021). Each heuristic is evaluated over 50 different initial states guiding GBFS implemented in FD (Helmert 2006). It is known that a lack of accuracy in heuristics can lead to blow ups in the size of search trees (Helmert, Röger et al. 2008), therefore we can test the relative accuracy of the heuristics by using them in the GBFS and comparing their coverage. The coverage of a planner is defined as the percent of initial states for which a solution path is found within the given planning budget. Ferber et al. (2021) report observing that in general the coverage superiority between the different NN heuristics tested did not vary over time. That is, the planning time used and the relative coverage superiority between the algorithms were not correlated. Given this observation and constraints on our available compute resources we reduce the overall 10 hour planning time-limit that Ferber et al. used by 99% down to just 6 minutes, and compare with the NN defined heuristic baseline algorithms' results as provided by Ferber et al. (2021) which use the 10 hour planning time-limit.

As our experiments are run on different hardware and Ferber et al.'s algorithms use the Keras (Chollet et al. 2015) and the TensorFlow (Abadi et al. 2015) libraries to train and evaluate their NN while we use PyTorch (Paszke et al. 2019), we preformed a comparison of the evaluations per second used for commonly solved instances between Ferber et al.'s (2021) methods and $h^{\text{N-RSL}}$. Ferber et al.'s algorithms use the same NN architecture meaning that for the same instance, and assuming the relationship between evaluations and planning time is linear, the evaluations per second should be similar. The data logs provided by Ferber et al. only provide the number of evaluations completed by FD if the problem is solved, therefore we can only compare the ratios of commonly solved instances. Table 7.1 shows the evaluations per second ratios between our h^{RSL} FastDownward runs and the baseline algorithms run by Ferber et al. (2021). Table 7.1 shows that depending on the domain the ratios averaged between 0.3 to 16.1, meaning h^{RSL}'s evaluations per second were sometimes less but could also be up to 16.1 times higher. We also benchmark on our hardware an implementation of the SING (Yu, Kuroiwa, and Fukunaga 2020) algorithm which Ferber et al. (2021) did not test and has not previously been applied to this benchmark set. We implemented the configuration C4 of the SING algorithm (Yu, Kuroiwa, and Fukunaga 2020) as described by Yu et al. but using the same NN architecture, loss function and training hyper-parameters as used for RSL. 10^5 samples were used for training, which were collected using a budget of 500 samples per DFS resulting in 200 total DFS rollouts. Finally, we ran the baselines that do not require offline training, h^{FF} and LAMA, on our hardware with the 6 minute time budget.

We define each h^{RSL} heuristic through the same NN architecture used by Ferber et al. (2021). The NN is a *residual network* (He et al. 2016) made up of two dense 250 neuron layers followed by a single residual block with two dense 250 neuron layers followed by a single neuron output. Each neuron in the NN uses the rectified linear activation function. The inputs of the NN are full states of $S(\Pi)$ represented by a Boolean vector $\{0, 1\}^{|F|}$. As Ferber et al. do, we set the $loss$ function of the SL optimization problem (2.4) to be the *mean squared error* (MSE). We use the Adam (Kingma and Ba 2015) stochastic optimization algorithm to find locally optimal solutions of (2.4), using the following hyper-parameters: learning rate is 10^{-4}, initial decay rates of $\beta^1 = 0.9$ and $\beta^2 = 0.999$, $\epsilon = 10^{-8}$, batch size is 64, and maximum number of epochs is set to $1,000$. Additionally, we use the *early stopping* heuristic (Duvenaud, Maclaurin, and Adams 2016) setting the *patience* parameter to 2. We split each instance data set into training and validations sets, taking the former 80% and the latter 20% of available samples.

Note that Ferber et al.'s TSL method trains 10 heuristics functions and selects the best performing heuristic on a set of validation states. We follow this method by running RSL 10 times with different random seeds on each instance to produce 10 heuristic functions. We then test each heuristic over a set of validation states, which are collected using the same method as Ferber et al. (2021), to select the best performing heuristic for each instance. We report both the average coverage of the 10 heuristics learnt and the best heuristic found via the validation method. Due to the amount of compute required in testing an algorithm setting over the benchmark we do not tune the architecture or any training parameters of the NN.

Training is run simultaneously for 80 instances over 40 Intel®Xeon®Gold 6138 CPU @ 2.00GHz processor cores with 720GB of shared RAM, limiting each training instance to run on a single vCPU (each CPU core has 2 vCPUs). For training the h^{RSL} heuristic, the Tarski framework (Francés and Ramírez 2021) is used to ground each problem instance in order to get the F,O, and G sets used for the regressions. In line with the h^{Boot}, h^{BExp}, h^{AVI} and h^{SL} heuristics, the FD translator is used for identifying a set of state mutexes to enforce, for both the state sampling and valid actions for the regression operator as described in the previous section. For evaluation we use the same hardware, along with Downward lab (Seipp et al. 2017) to run 80 FD searches at once limiting each FD search to run on a single vCPU with a maximum of 3.8 GB of memory. The PyTorch library (Paszke et al. 2019) is used for the training and evaluations of RSL's NN.

7.5.2 Results

First we present the results from a grid search over the hyper-parameters of RSL. To maximise the number of hyper-parameter combinations we could test with our limited computational resources, we evaluated each trained heuristic from the grid search on 10 out of the 50 initial states for each of the 143 instances in the benchmark set. Figure 7.2 summarises the impact of each of the hyper-parameter values tested on the average coverage of h^{RSL} across the benchmark set. For the hyper-parameter grid search we tested 16 different settings over the different combinations of N_t =10,000 or 100,000, P_r = 0 or 50, N_r = 1 or 5, and L = 50 or 500. Table 7.2 details the different configurations tested in the hyper-parameter grid search along with their average coverage over each domain. Figure 7.2 shows that the value of P_r has the biggest influence on the average coverage over the benchmark set of h^{RSL} used in GBFS. This observation shows the importance of including states in the training set that are not within any of the sets of states visited by the rollouts from the goal set. It is common practise in Supervised Learning problems to have

N_t	10,000								100,000							
P_r	0				50				0				50			
N_r	1		5		1		5		1		5		1		5	
L	50	500	50	500	50	500	50	500	50	500	50	500	50	500	50	**500**
Moderate Tasks																
blocks	0.0	20.0	0.0	20.0	**100**	76.0	80.0	48.0	0.0	0.0	0.0	60.0	54.0	68.0	62.0	76.0
depot	0.0	0.0	3.3	5.0	31.7	71.7	**76.7**	48.3	0.0	0.0	0.0	6.7	45.0	41.7	66.7	43.3
grid	0.0	0.0	0.0	0.0	25.0	60.0	10.0	50.0	0.0	0.0	0.0	0.0	20.0	20.0	50.0	**95.0**
npuzzle	0.0	0.0	0.0	0.0	0.0	3.8	0.0	**32.5**	0.0	0.0	0.0	0.0	0.0	1.2	0.0	16.2
pipesworld	16.0	15.0	13.0	17.0	11.3	40.0	44.0	50.0	12.0	15.0	28.0	16.0	47.0	53.0	67.0	**70.0**
rovers	7.5	7.5	6.2	10.0	11.3	12.5	10.0	11.2	7.5	7.5	7.5	8.8	12.5	11.2	**13.8**	8.8
scanalyzer	83.3	83.3	66.7	**100**	66.7	**100**	**100**	66.7	83.3	83.3	66.7	83.3	**100**	**100**	**100**	66.7
storage	0.0	0.0	0.0	0.0	27.5	15.0	2.5	0.0	0.0	0.0	0.0	0.0	40.0	0.0	**32.5**	25.0
transport	0.0	0.0	12.5	5.0	62.5	75.0	62.5	70.0	0.0	0.0	0.0	13.8	31.2	52.5	57.5	**77.5**
visitall	0.0	0.0	1.7	20.0	48.3	83.3	73.3	**100**	0.0	0.0	3.3	30.0	5.0	93.3	21.7	96.7
Average	10.7	12.6	10.3	17.7	39.8	53.7	45.9	47.7	10.3	10.6	10.6	21.9	35.5	44.1	47.1	**57.5**
Hard Tasks																
blocks	0.0	0.0	0.0	0.0	**40.0**	24.0	6.0	20.0	0.0	0.0	0.0	0.0	20.0	12.0	2.0	26.0
depot	0.0	0.0	0.0	0.0	**17.1**	5.7	12.9	7.1	0.0	0.0	0.0	0.0	4.3	4.3	1.4	0.0
grid	0.0	0.0	0.0	0.0	0.0	12.5	26.2	7.5	0.0	0.0	0.0	0.0	1.2	13.8	2.5	**41.2**
npuzzle	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pipesworld	0.0	0.0	0.0	0.0	6.0	7.0	5.5	5.5	3.0	0.0	6.0	0.0	9.0	12.0	18.5	**28.0**
rovers	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
scanalyzer	43.3	13.3	63.3	23.3	**100**	**100**	66.7	66.7	30.0	33.3	73.3	66.7	**100**	86.7	**100**	66.7
storage	0.0	0.0	0.0	0.0	1.2	3.8	6.2	**10.0**	0.0	0.0	1.2	0.0	**10.0**	0.0	8.8	5.0
transport	0.0	0.0	0.0	0.0	0.0	**12.0**	6.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
visitall	0.0	0.0	0.0	0.0	0.0	20.0	0.0	**100**	0.0	0.0	0.0	0.0	0.0	90.0	0.0	**100**
Average	4.3	1.3	6.3	2.3	16.4	18.5	13.0	21.7	3.3	3.3	8.1	6.7	14.5	21.9	13.3	**26.7**

TABLE 7.2: Comparison of the coverage given a 6 minute planning time budget of h^{RSL} using a range of different hyper-parameter values. Each run uses only a single trial and is evaluated over 10 of the 50 initial states for each instance in the benchmark set.

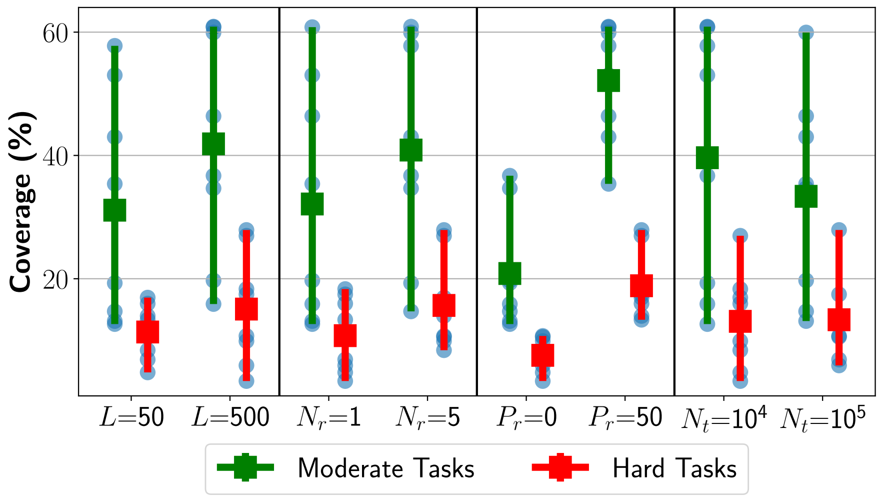


FIGURE 7.2: A summary of the average coverage over the “Moderate" and “Hard" tasks for 16 different configurations of RSL shown in Table 7.2. The 4 boxes each show a different segmentation of the 16 configurations according to the value of a single hyper-parameter. Each configuration’s coverage is marked with a blue dot and the lines show the range of coverage over the configurations with the mean marked with a square.

positive and negative examples. Negative examples in a planning context can be interpreted as states that are unlikely to allow for shorter plans to the goal, which may explain the benefit of sampling states outside of the sets of states visited in the rollouts. Figure 7.2 also suggests that RSL benefits from using more than one rollout for extracting training states and considering 500

Planning Budget	**Ave. over Trials**		**Methods using Validation**										
	6 minutes		6 minutes		600 minutes*					6 minutes		
	h^{RSL}	$h^{\text{N-RSL}}$	h^{RSL}	$h^{\text{N-RSL}}$	h^{Boot}	h^{BExp}	h^{AVI}	h^{TSL}	h^{HGN}	h^{SING}	h^{FF}	LAMA
						Moderate Tasks							
blocks	80.0	91.5	99.2	**100**	18.0	0.0	0.0	80.4	**100**	17.6	100	100
depot	48.7	58.8	77.0	89.0	60.3	32.7	54.7	**90.3**	0.0	0.0	93.7	99.3
grid	71.0	60.3	**100**	97.0	**100**	**100**	51.0	93.0	0.0	0.0	90.0	100
npuzzle	15.3	18.9	25.2	**46.8**	28.0	0.0	1.0	0.0	0.3	0.0	92.8	97.8
pipesworld	70.7	69.6	85.8	82.6	57.8	68.4	50.2	**92.2**	7.6	13.8	63.0	98.6
rovers	12.6	12.5	15.0	15.8	**48.2**	21.8	45.0	26.0	14.0	6.2	79.5	100
scanalyzer	87.4	94.1	**100**	**100**	33.3	70.7	67.3	82.7	11.0	16.7	91.7	100
storage	16.1	16.4	17.0	18.0	**89.0**	57.5	69.5	24.5	0.0	0.0	27.5	34.0
transport	74.8	70.8	**100**	93.8	**100**	**100**	87.5	99.2	94.7	0.0	100	100
visitall	93.6	95.4	99.7	98.0	55.3	0.0	0.0	0.0	**100**	1.3	89.0	100
Average	57.0	58.8	71.9	**74.1**	59.0	45.1	42.6	58.8	32.8	5.6	82.7	93.0
Average Train Time per Inst. (Hours)	0.37	0.38	3.68	3.75	112*				13.1*	1.6	-	
						Hard Tasks							
blocks	18.5	45.3	36.4	**68.0**	0.0	0.0	0.0	0.0	50.0	0.8	46.4	96.0
depot	4.0	3.3	14.6	12.6	8.3	4.3	12.9	**35.4**	0.0	0.0	9.4	69.4
grid	21.9	32.1	67.2	69.0	87.8	**95.0**	70.5	60.2	0.0	0.0	31.0	100
npuzzle	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.8	12.8
pipesworld	27.4	25.0	33.8	33.1	23.4	19.1	8.0	**48.7**	0.0	0.1	20.4	68.0
rovers	0.0	0.1	0.1	0.1	2.8	0.8	**6.5**	1.5	0.3	0.0	8.3	100
scanalyzer	86.4	89.7	**100**	**100**	3.3	0.0	60.7	60.0	0.0	0.0	83.3	100
storage	3.7	2.5	1.8	0.2	**27.2**	13.2	15.8	0.0	0.0	1.0	9.2	9.0
transport	0.0	1.2	0.0	**8.8**	0.0	0.0	2.4	0.0	0.0	0.0	0.0	59.6
visitall	82.4	91.8	**100**	**100**	28.0	0.0	0.0	0.0	**100**	0.0	40.0	100
Average	24.4	29.1	35.4	**39.2**	18.1	13.3	17.7	20.6	15.0	0.2	25.4	71.5
Average Train Time per Inst. (Hours)	0.69	0.69	6.88	6.88	112*				10.4*	2.6	-	

TABLE 7.3: Comparison of the coverage of h^{RSL} with other Neural Network defined heuristics functions introduced by Ferber et al. (2021) h^{Boot}, h^{BExp}, h^{AVI}, as well as the h^{TSL} introduced by Ferber et al. (2020), and h^{HGN} from Shen et al. (2020). The table also shows the coverage of h^{SING}(Yu, Kuroiwa, and Fukunaga 2020) (using a single trial), h^{ff}(Hoffmann and Nebel 2001) and LAMA (Richter and Westphal 2010) (run on same hardware as RSL). Note that h^{HGN} trains one heuristic per domain not instance with training budgets between 14.7 and 112 CPU hours. The bold numbers indicate the highest coverage among the Neural Network methods. *Note that the baseline learning methods that use a 600 minute planning budget are the values as reported by Ferber et al. (2021). According to standard single thread CPU benchmarks, our vCPU can be 20% faster. We provide a discussion of the impact of discrepancies in hardware and software in the Methodology Section.

applications of the regression operator instead of 50. Interestingly, Figure 7.2 shows that training with 10,000 or 100,000 states does not influence the performance in an obvious way. However, overall the best performing set of hyper-parameters over the “Hard Tasks" which we use for all experiments from this point forth are N_t =100,000, P_r =50, N_r=5, and L =500.

Table 7.3 shows a comparison of existing methods with respect to RSL and N-RSL, using the best performing configuration from the hyper-parameter grid search. As can be seen in Table 7.3 the results of our SING implementation are very poor. In addition to the results shown in the table we also performed individual runs of instances using SING with the same NN architecture, loss function, number of training samples, and number of samples per DFS as described by Yu et al. (2020), however the performance observed was still poor. Yu et al. mention that they perform manual tuning of hyper-parameters but do not provide the selected parameters used by the NN training algorithm, which could be the reason for the poor performance observed in our results. Due to SING's poor performance we omit it from consideration in our results analysis going forward.

The first notable difference in Table 7.3 is the average training times used by each algorithm. h^{RSL} without validation uses less than 1% of the training CPU time used by all the other per-instance NN defined heuristics functions for both the "Moderate" and "Hard" task sets. Even when using the validation method which requires 10 RSL executions the training time used is less than 7% of the per-instance NN methods. The overall average coverage over the benchmark set shows that h^{RSL} and $h^{\text{N-RSL}}$ outperform the other NN defined heuristic functions for both the "Moderate" and "Hard Tasks" with and without using the validation method. The model-based method LAMA clearly dominates all other methods. The model-based heuristic h^{FF} also outperforms h^{RSL} and $h^{\text{N-RSL}}$ on the "Moderate Tasks", however on the "Hard Tasks" h^{RSL} with validation and $h^{\text{N-RSL}}$ with and without validation have better overall performance. While $h^{\text{N-RSL}}$ has better average coverage than h^{RSL} the difference is small with a 1.8% and 4.7% average improvement over the "Moderate" and "Hard Tasks" respectively when no validation is used.

Figure 7.3 provides additional insights into the performance of $h^{\text{N-RSL}}$ compared to the best performing baseline algorithms. For commonly solved tasks, h^{TSL} on average produces higher quality plans than $h^{\text{N-RSL}}$. This observation could be a result of h^{TSL} having higher quality labels used in training. h^{TSL} uses h^{FF} with GBFS to label sampled states in the train set, while $h^{\text{N-RSL}}$ labels states with goal distance estimates derived from the state sets visited by N-RSL's regressions. The objective of h^{FF} with GBFS is to find the shortest path from a state to the goal state, while the objective of N-RSL's regression is to maximise a defined novelty measure (Equation 7.5) with the aim of finding the shortest path that visits the most number of reachable atoms $a \in F$. This trade-off between exploration and exploitation shows through in the results where $h^{\text{N-RSL}}$ has a higher coverage than h^{TSL} over a diverse set of initial states but produces lower quality plans in terms of plan length among the commonly solved tasks. This behaviour is also displayed when comparing directly to the h^{FF} heuristic, while LAMA dominates $h^{\text{N-RSL}}$ in both coverage and plan quality.

Figure 7.4 shows that for smaller problem instances the model-based methods h^{FF}, and LAMA are able to compute a higher number of evaluations per second than h^{RSL}. However, h^{RSL}'s evaluations per second stays relatively constant while the model-based methods evaluations per second decreases as problem instances become larger.

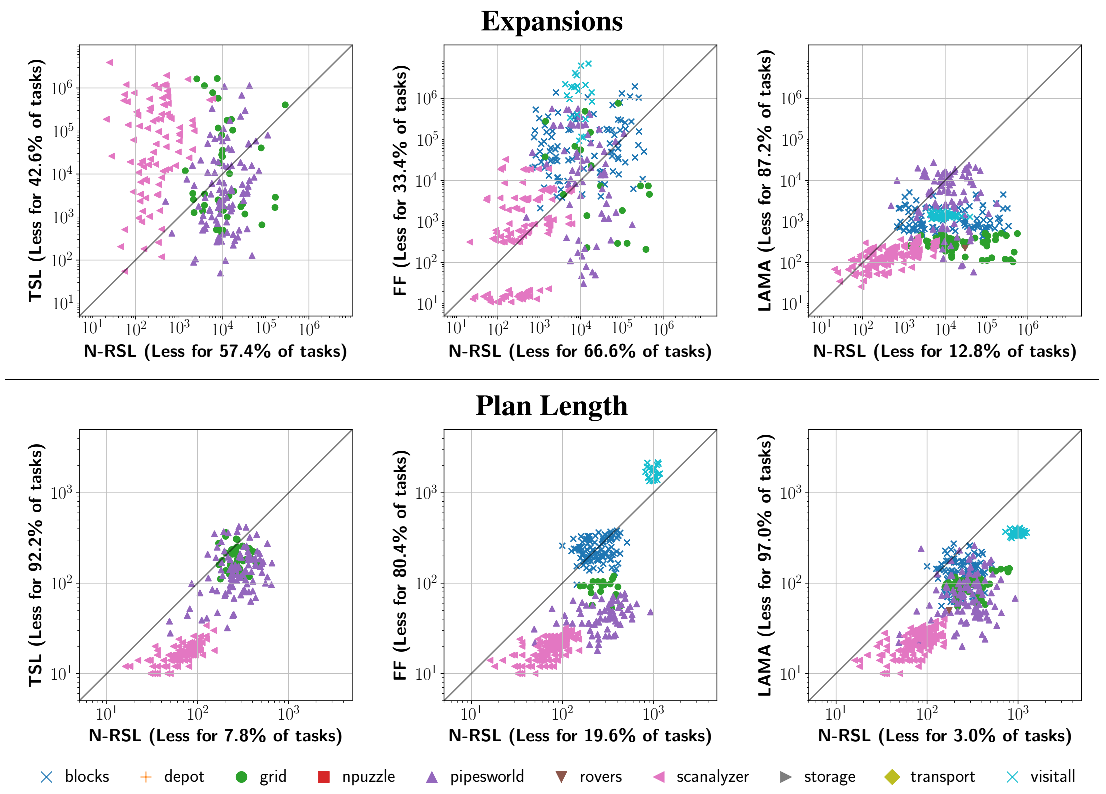


FIGURE 7.3: Pairwise comparisons over commonly solved tasks between $h^{\text{N-RSL}}$ using the validation method, and the best performing baseline NN based heuristic function h^{TSL} and model-based methods h^{FF} and LAMA over the "Hard Tasks" benchmark set. The top row shows the scatter plot of the number of expansions used by each search algorithm while the bottom row shows the plan length found by each algorithm. The percentage value, shown in the axis titles, is the percentage of the commonly solved tasks for which the relevant algorithm required fewer expansions or discovered a shorter plan than the algorithm it is being compared to. Note that tasks which are not solved by at least one of the algorithms in the pair are not included in these graphs. Graphs were generated using Downward Lab (Seipp et al. 2017).

7.6 Initial results of linear defined heuristics using N-RSL

In this chapter we have presented the N-RSL algorithm for learning NN-defined heuristic functions. We defined the heuristic function that was learnt as a NN which matched recent works that have explored learning such heuristic functions for classical planning. NN training relies upon stochastic gradient descent optimisation techniques with no guarantees on the optimally of the NN parameters found through the optimisation. Alternatively, through defining the heuristic function as a linear weighting of the boolean facts, we can define a linear programming problem. Using linear programming we can find an exact solution to the optimisation problem, that is, the problem of

$$\min_{\theta \in \mathcal{H}} g(\mathbf{y} - z(\mathbf{X}, \theta)), \tag{7.7}$$

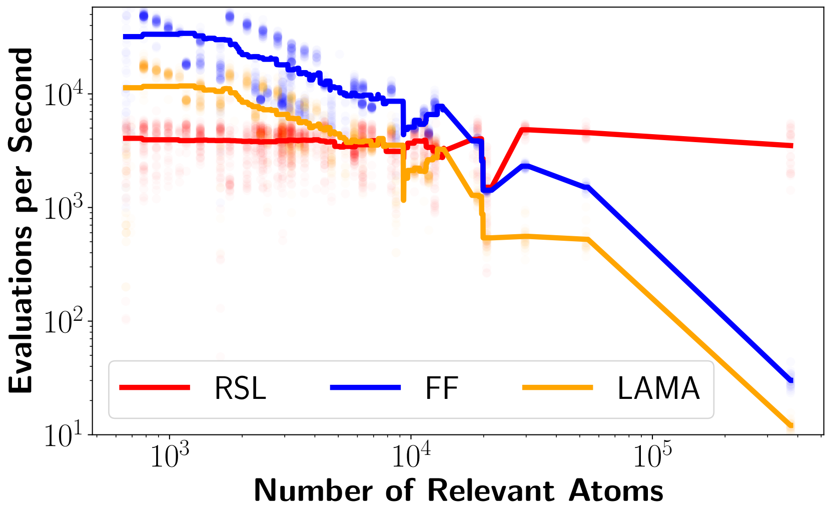


FIGURE 7.4: The number of heuristic evaluations per second versus the number of relevant atoms as determined by FD translator's reachability analysis for a set of commonly solved problem instances between h^{RSL}, h^{FF} and LAMA. Each dot represents the values for a single problem instance and the line represents the moving average with a window size of 2000 atoms.

	Moderate Tasks			Hard Tasks		
	Linear		NN	Linear		NN
	L = 50	L = 500	L = 500	L = 50	L = 500	L = 500
blocks	100	0.0	91.5	41.2	0.0	45.3
depot	81.0	49.0	58.8	0.0	7.1	3.3
grid	51.0	2.0	60.3	39.8	4.5	32.1
npuzzle	0.0	0.0	18.9	0.0	0.0	0.0
pipesworld	0.2	0.2	69.6	0.0	0.0	25.0
rovers	2.0	3.5	12.5	0.0	0.0	0.1
scanalyzer	82.7	50.7	94.1	9.3	0.0	89.7
storage	2.0	0.0	16.4	1.0	0.0	2.5
transport	100	25.2	70.8	0.0	0.0	1.2
visitall	16.0	76.0	95.4	0.0	100.0	91.8
Average	43.5	20.7	58.8	9.1	11.2	29.1

TABLE 7.4: Initial results of a comparison between linear defined heuristics functions versus NNs. Note that the NN results are averaged over 10 trials while the linear results only use a single trial.

becomes

$$\min_{\theta \in \mathcal{H}} g(\mathbf{y} - \mathbf{X}\theta), \tag{7.8}$$

where z represents the non-linear NN function, $\mathbf{X}$ is the training states represented as boolean facts, $\mathbf{y}$ is the cost-to-go estimates for $\mathbf{X}$ computed by the RSL method, θ is the parameters of the heuristic function, $\mathcal{H}$ is the set of all valid heuristic function parameters, and g is the loss function.

Heuristics defined as a linear combination of facts in this way are known as potential heuristics (Pommerening et al. 2015). Potential heuristics can be optimised using linear programming with constraints that ensure the heuristics are consistent and goal aware (Pommerening et al. 2015). Here we present some initial results using the linear programming package `CVXPY` (Diamond and Boyd 2016) to solve equation 7.8. Additionally we add a normalisation term $h(a)$, that is scaled by the constant parameter λ such that our objective function becomes,

$$\min_{\theta \in \mathcal{H}} g(\mathbf{y} - \mathbf{X}\theta) + \lambda h(\theta) \tag{7.9}$$

For these initial experiments we do not impose any constraints on the linear program and use the Lasso (Tibshirani 1996) method where $g(a) = ||a||_2^2$ and $h(a) = ||a||_1$. We select the λ value for each optimisation by trialing 10 values from 0.001 to 10 and selecting the best performing value according to a validation set. In Table 7.4 we present the linear programming results using a single trial. It is clear from Table 7.4 that the non-linear NN heuristic definition pays off for the N-RSL algorithm, however, there are a few interesting observations. It is clear that the L parameter of N-RSL has a great effect on the accuracy of the linear heuristics. For the visitall domain, optimal solution trajectories are often long as they are required to visit each position in the problem to achieve the goal. We can see that for visitall the linear N-RSL using the larger L value increases its performance greatly. In contrast, the optimal solution paths for blocksworld in the moderate and hard tasks are shorter than those of visitall. For blocksworld N-RSL with the smaller L performs much better. There are a number of research directions that could be explored given these initial results. First, it would be interesting to compare the results of linear heuristics that incorporate the constraints used by related potential heuristics work (Pommerening et al. 2015). Second, a more rigorous investigation into the effects of the hyper-parameters of N-RSL, such as L and P_r, on the performance of the linear heuristics may provide interesting insights into the N-RSL method.

7.7 Discussion

The results showed that using the novelty-based regression policy μ^+ improved the heuristic that RSL learned. A promising future research direction is to explore different types of novelty-based regression policies that can be used in RSL. For example, instead of defining μ^+ to be independent of previous regression trajectories it may be beneficial to define it to count only novel preconditions that have not been observed in any of previously performed trajectories.

The heuristic values from the NN defined heuristic functions we report in this chapter are calculated by taking the linear combination of the outputs of the last hidden layer of the NN. We note that the NN heuristics discussed in this chapter and those of Ferber et al. (2020; 2021) are an application of *state aggregation* (Bertsekas 2018), a well-known technique in Approximate Dynamic Programming. For each of the different problem instances we explored, the size of the last hidden layer was fixed at 250 neurons. It is possible that for harder problem instances a feature vector larger than 250 is required in order to produce more useful heuristics. Ferber et al. (2020) have performed an investigation into effects of the NN architecture in terms of the activation functions used and the number of hidden layers, however they did not explore the number of neurons used within the hidden layers. One avenue of future work is to investigate the impact of scaling the number of neurons used within the hidden layers of the NN so the capacity (Goodfellow, Bengio, and Courville 2016) of the NN scales too with the size of the planning instance.

One advantage we observed of the NN heuristic functions over the model-based heuristics in Figure 7.4 is that the gap in number of evaluated states per second narrows until it eventually disappears, so the NN heuristic becomes significantly faster in comparison (and less informed too, probably) as the problem size increases. This is because only the input layer size changes according to $|F|$ for the NN architecture used in this work, while the rest of the neurons and edges remain constant. That is, as the input layer has $|F|$ neurons and the first hidden layer has 250 neurons there are $250|F|$ edges joining the two layers resulting in the NN evaluations being in $O(|F|)$ time. Conversely, h^{FF} requires completing a search over the problem with the delete relaxation which has time-complexity that is polynomial over the number of actions and reachable atoms (Hoffmann and Nebel 2001).

Future work should investigate NN heuristics like h^{RSL} over problems that are too large for h^{FF} or LAMA to produce enough evaluations per second to be useful. It is possible that a NN defined heuristic function will be able to produce a large enough number of evaluations per second to be useful over these large domains, even if the heuristic is less informed than the model-based heuristics. Another domain-type that NN-heuristics will likely be useful for are problems that only have a regression simulator available and not a full action model, and hence heuristics such as LAMA and h^{FF} can not be used. Last, we have shown that h^{RSL} and $h^{\text{N-RSL}}$ have good performance while requiring significantly smaller resources than the NN-based baselines considered in this chapter. We think that this encouraging result supports the assumption that RSL and follow-up methods can be used to obtain useful heuristic functions for large instances of challenging combinatorial optimization problems.

7.8 Conclusion

We introduced a new algorithm, N-RSL, for learning per-instance NN defined heuristic functions over classical planning problems. N-RSL uses a novelty measure in order to select what regression operations to apply on the goal with the aim of maximising the diversity of the partial states visited by the rollouts. These rollouts label a sets of states with an upper bound of their distance from the goal. Using this N-RSL trains a NN heuristic using SL. We ultimately address **RQ3** by showing that N-RSL generalises better than previous NN defined heuristic functions in terms of coverage while using a fraction of the training compute. While N-RSL performs better than model-based heuristic function h^{FF} on harder tasks it was still dominated by the state-of-the-art model-based LAMA algorithm. We also provided a number of suggestions for how future work could address some of the open questions about NN defined heuristic functions. We hope the addition of N-RSL to the catalogue of learning methods for NN defined heuristics, will enable the discovery of new heuristic methods and benchmarks that highlight the potential of NN heuristics.

Chapter 8

Conclusion

To conclude we will summarise the main contributions this thesis has presented and to what extent they addressed the Research Questions set out in Chapter 1. Finally, we discuss future lines of research to follow up and extend upon the work presented in this thesis.

8.1 Contributions

8.1.1 Cost-to-go approximation for model-free planning

Part 1 focused on the research question of "How can planning and learning interact to trade-off exploration and exploitation for model-free simulator-based problems?", which we explored through new planning and learning algorithms relying on mechanisms such as novelty pruning, which encourages exploration, and learnt cost-to-go approximations, which encourages the exploitation of previously gained knowledge. In Chapter 3 we identified a limitation of the state-of-the-art model-free simulator-based planner RIW on SSP problems and showed that it can be addressed using cost-to-go approximations. In Chapter 4 we followed up on the results of Chapter 3 to propose and evaluate a method for learning cost-to-go approximations for a RIW-based planner. Additionally, Chapter 4 incorporated learnt action policies into the RIW-based planner to further improve its performance. In summary the key contributions of Part 1 included,

- extending width-based algorithms to perform well on SSP problems by introducing the RIW(1)-λ algorithm with cost-to-go approximations,

- defining a methodical learning schedule for planning and learning methods,
- introducing the N-CPL and CPL planning and learning algorithms,
- identifying characteristics of MDP problems that influence the performance of different planning and learning approaches.

8.1.2 Imitation Learning via regression

Part 2 examined the research question "To what extent can symbolic regression given a full, relaxed or partial demonstration trajectory assist learning?". Chapter 5 investigated learning methods which use demonstration trajectories to iteratively learn a policy to reach the problem's goal from states of the demonstration that are increasingly further away from the goal state. Chapter 5 showed that the learning methods investigated can fail to reliably learn successful policies for domains which require the agent to change its mode of behaviour multiple times in order to achieve the goal. We follow up in Chapter 6 by introducing a method that learns piece-wise policies that can more reliably learn to solve problems that require multiple modes of agent behaviour. This new method, BLPP, is particularly useful for the application of learning from trajectories produced by a planner as it can learn from demonstration trajectories that are partial or generated on an relaxed version of the environment, and does not require the demonstration's actions, rather it only requires the demonstration's states. In summary, the key contributions of Part 2 include,

- an investigation of the generality of backwards learning methods, including the effects of exploration bonus terms,
- introducing a novel method to partition the state space symbolically into polytopic regions from which different sub-policies are learnt,
- defining a general domain-independent cost function to define the problem solved by each sub-policy,
- introducing an efficient method to select starting states, exploiting the compact representation of state regions as polytopes, for policy iteration.

8.1.3 Learning heuristics through symbolic regression

Part 3 investigated the research question of "To what extent can learning with symbolic regression produce useful cost-to-go approximations?". Chapter 7 introduced the N-RSL method that uses a regression search to teach a SL algorithm to learn cost-to-go approximations. Our experimental analysis showed that these N-RSL learnt cost-to-go approximations can provide useful signal to forward-based search planners. In summary the contributions of Part 3 include,

- defining a new novelty measure for regression based search,
- introducing a new method for training NN defined heuristic functions,
- investigating exact versus approximate methods for learning heuristic functions.

8.2 Future research directions

Here we detail a few future work directions that arise from the work presented in this thesis.

8.2.1 Extension of Critical Path Learning to stochastic settings

While our work in Chapter 3 extended width-based planning algorithms to a stochastic setting, we only presented results of the width-based planning and learning method, N-CPL, in Chapter 4 on a deterministic setting. Open questions remain about the ability of width-based planning and learning methods like N-CPL to generalise to stochastic settings.

8.2.2 Connections between RL exploration and width-based planning methods

In Chapter 5 we explored and discussed a number of exploration methods that are readily used by RL algorithms. Additionally, we discussed across Chapters 3, 4 and 7 the ability of width-based novelty planning methods to encourage exploration through aiming to maximise the structural diversity of transitions visited. The novelty measures described in 4.3.1 are similar to the intrinsic reward exploration bonuses for RL discussed in 5.3 in that they use information about the states previously visited by the agent in order to encourage exploration. The relationship between these measures has not yet been defined or explored and in doing so may provide meaningful insights into the interface between planning and learning methods.

8.2.3 Potential heuristics approach for RSL

In Chapter 7 we presented and discussed the initial results of the N-RSL method using a simplified heuristic function representation of the weighted sum of boolean facts instead of a NN. One advantage of the simplified heuristic function representation is that its parameters can be optimised using linear programming methods rather than the approximate approaches required for NN functions. There are a number of different research directions to extend upon this initial work, perhaps the most promising is to incorporate the constraints that previous work have defined (Pommerening et al. 2015), which can be used within the linear program to ensure the learnt heuristic is consistent and goal-aware.

8.2.4 N-RSL follow-ups and extensions

In Chapter 7 we showed that the introduced RSL and N-RSL methods can learn useful heuristics for the benchmark set of problem instances within an average training time of around 30 minutes. Given that the IPC satisficing track (discussed in 2.5.4) uses a time budget of 30 minutes, it may be insightful to benchmark the N-RSL method using the IPC satisficing track evaluation protocols. For example, N-RSL could use 24 minutes to train a heuristic function for the given instance and then the remaining 6 minutes for planning with the learnt heuristic.

While the heuristics learnt by the RSL methods generalise over different initial states of an instance they do not generalise over different goals. One way RSL could be extended to generalise over different goals is to learn a unique heuristic function for each possible single atom goal. Once a heuristic is learnt for each single atom goal, the relevant heuristic function for an instance can be calculated from the single atom goal heuristics, using approximations for multiple atom goals similar to the approximations used by the $h^{\max}$ or h^{add} (Bonet, Loerincs, and Geffner 1997) functions. We hypothesise that the influence of the novelty mechanism in N-RSL will also have a greater influence on the success of RSL learning useful heuristics for single atom goals, as Lipovetzky and Geffner (2012) showed that width-based forward searches can quite effectively achieve single atom goals.

Appendix A

Imitation learning results separated out over demonstrations

In Chapter 6 we report the figures with the averages and 95% confidence intervals over every combination of trial and demonstration for each domain. Here we show the results using the same figures but separated out for each demonstration in Figures A.1, A.2, A.3, A.4, A.5 and A.6.

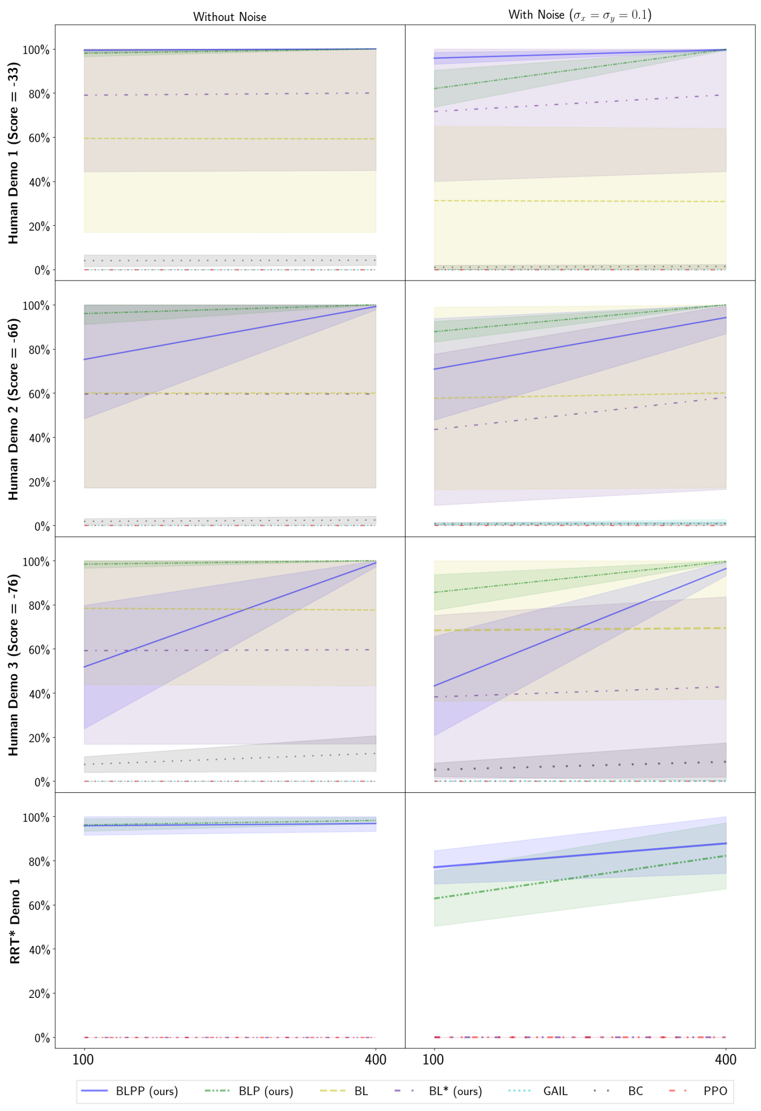


FIGURE A.1: Mobile-Robot results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

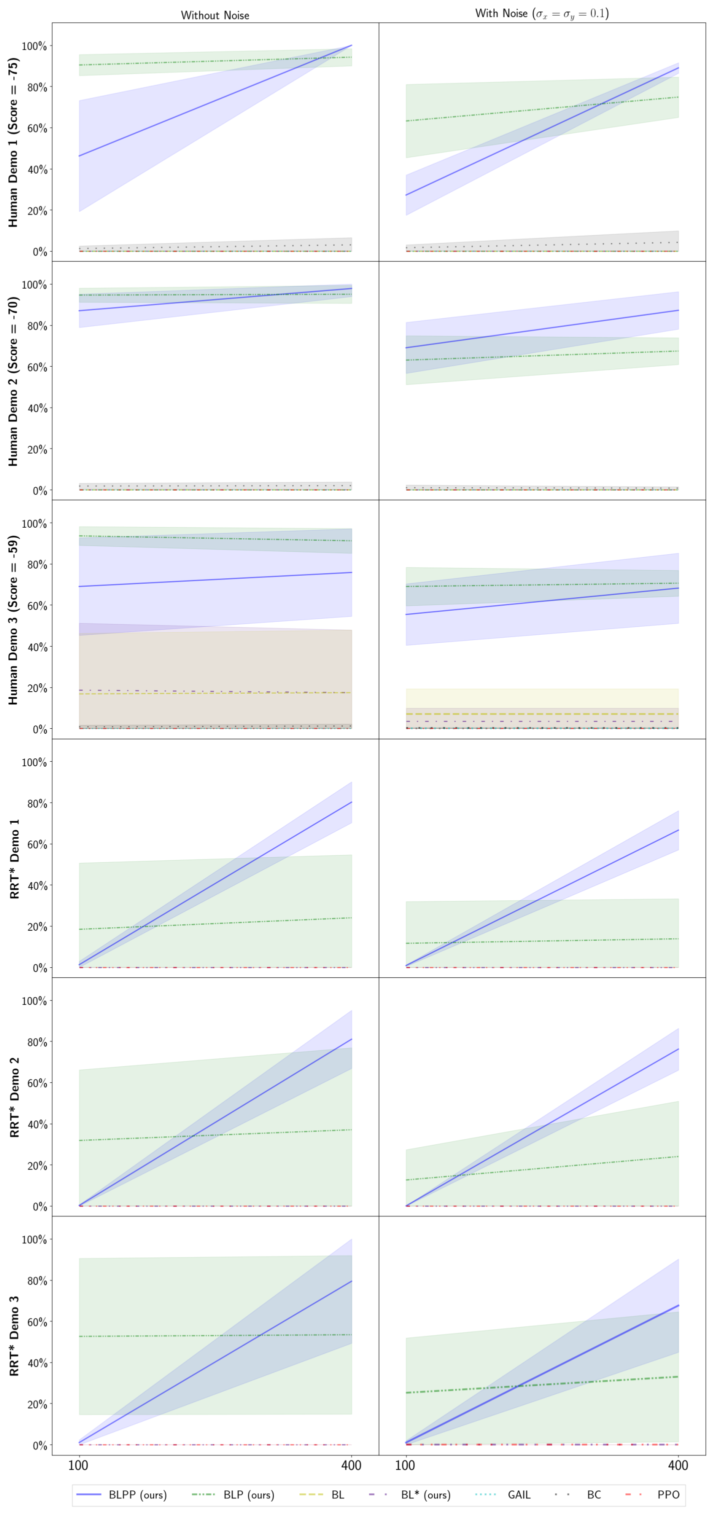


FIGURE A.2: Mobile-Robot Obstacles results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

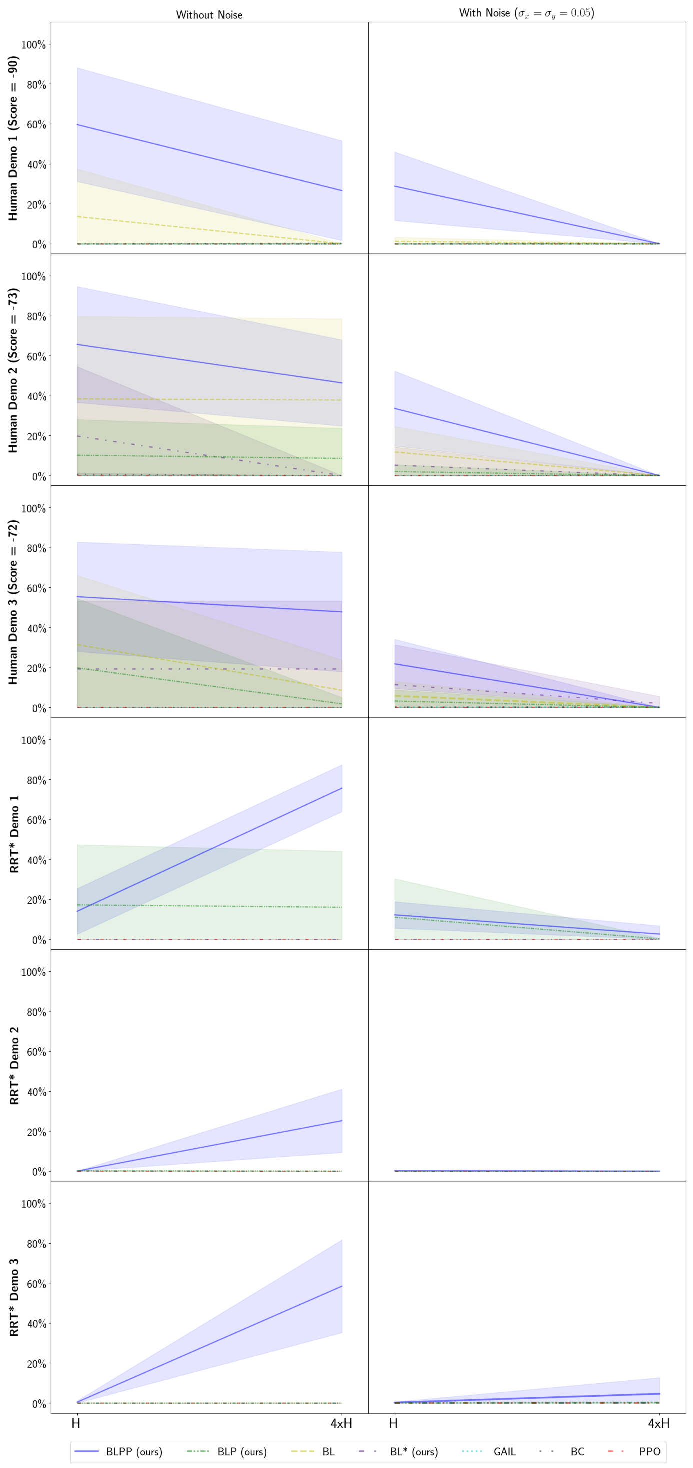


FIGURE A.3: Mobile-Robot Obstacles Stay in Goal results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

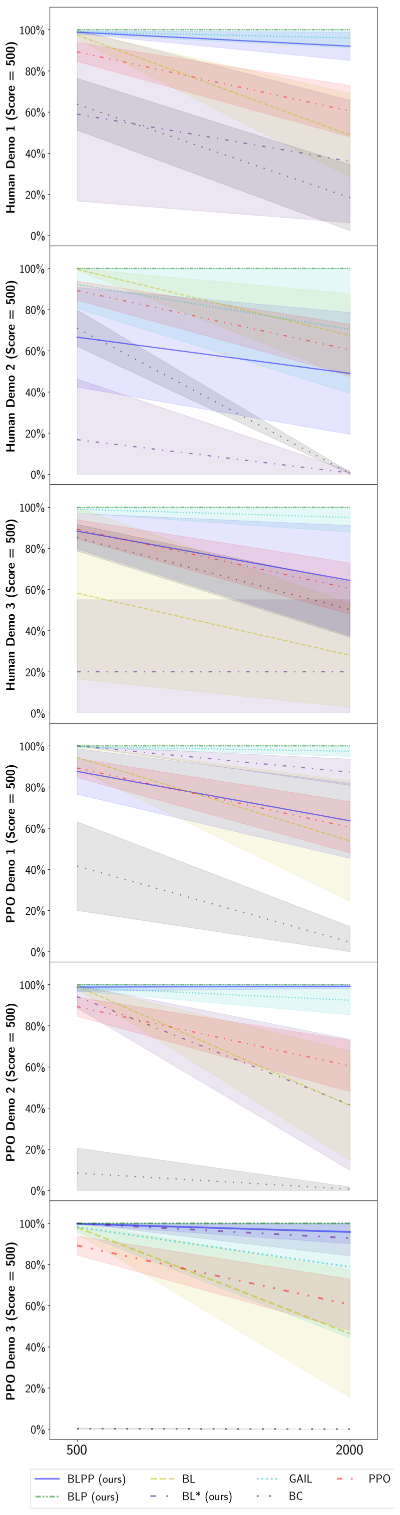


FIGURE A.4: CartPole results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

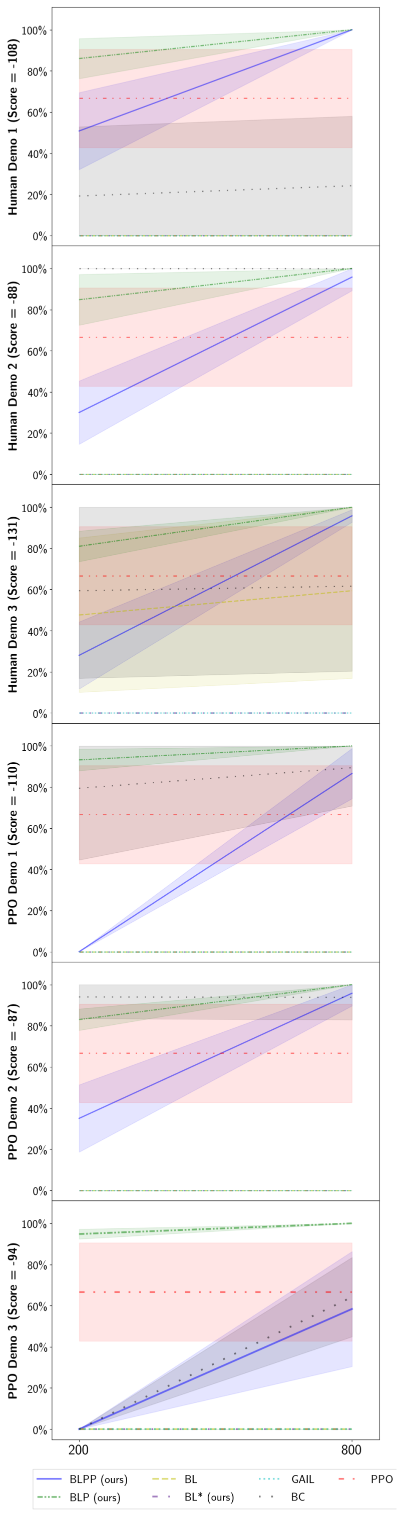


FIGURE A.5: Mountain Car results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

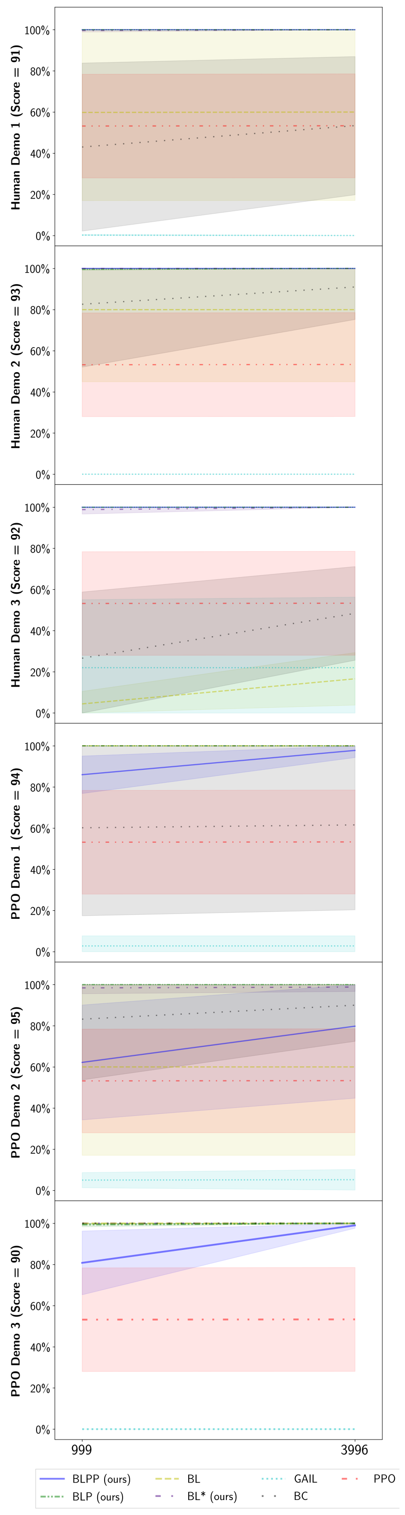


FIGURE A.6: Mountain Car Continuous results for each demonstration, percentage of evaluation episodes successful verses the evaluation horizon.

Bibliography

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

Abbeel, P.; and Ng, A. 2004. Apprenticeship learning via inverse reinforcement learning. *Proceedings of the twenty-first international conference on Machine learning*.

Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I. D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D. W.; Barrett, A.; Christianson, D.; et al. 1998. PDDL| The Planning Domain Definition Language. *Technical Report, Tech. Rep.*

Anthony, T. W.; Tian, Z.; and Barber, D. 2017. Thinking Fast and Slow with Deep Learning and Tree Search. In *NIPS*.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17): 2075–2098.

Argall, B. D.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5): 469–483.

Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; and Bharath, A. A. 2017. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6): 26–38.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite–Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, (47): 235–256.

Bagaria, A.; and Konidaris, G. 2019. Option discovery using deep skill chaining. In *International Conference on Learning Representations*.

Bagaria, A.; and Konidaris, G. D. 2020. Option Discovery using Deep Skill Chaining. In *ICLR*.

Bain, M.; and Sammut, C. 1995. A Framework for Behavioural Cloning. In *Machine Intelligence 15*, 103–129.

Bandres, W.; Bonet, B.; and Geffner, H. 2018. Planning with pixels in (almost) real time. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Barber, C. B.; Dobkin, D. P.; and Huhdanpaa, H. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4): 469–483.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Real–Time Learning and Control Using Asynchronous Dynamic Programming. *Artificial Intelligence*, 72: 81–138.

Barto, A. G.; and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1): 41–77.

Bellemare, M.; Srinivasan, S.; Ostrovski, G.; Schaul, T.; Saxton, D.; and Munos, R. 2016a. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, 1471–1479.

Bellemare, M.; Srinivasan, S.; Ostrovski, G.; Schaul, T.; Saxton, D.; and Munos, R. 2016b. Unifying Count-Based Exploration and Intrinsic Motivation. In Lee, D. D.; Sugiyama, M.; Luxburg, U. V.; Guyon, I.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 29*, 1471–1479. Curran Associates, Inc.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47: 253–279.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P. 2017. *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition.

Bertsekas, D. P. 2018. Feature-based aggregation and deep reinforcement learning: A survey and some new implementations. *IEEE/CAA Journal of Automatica Sinica*, 6(1): 1–31.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1): 5–33.

Bonet, B.; and Geffner, H. 2003. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 1233–1238.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *AAAI/IAAI*, 714–719.

Borrelli, F.; Bemporad, A.; and Morari, M. 2017. *Predictive control for linear and hybrid systems*. Cambridge University Press.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym.

Chiappa, S.; Racaniere, S.; Wierstra, D.; and Mohamed, S. 2017. Recurrent environment simulators. In *ICLR*. ICLR.

Chollet, F.; et al. 2015. Keras. `https://keras.io`.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games (ICCG)*, 72–83.

Dhariwal, P.; Hesse, C.; Klimov, O.; Nichol, A.; Plappert, M.; Radford, A.; Schulman, J.; Sidor, S.; Wu, Y.; and Zhokhov, P. 2017. OpenAI Baselines. `https://github.com/openai/baselines`.

Diamond, S.; and Boyd, S. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83): 1–5.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1): 269–271.

Dittadi, A.; Drachmann, F. K.; and Bolander, T. 2021. Planning from Pixels in Atari with Learned Symbolic Representations. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, volume 35, 4941–4949.

Duvenaud, D.; Maclaurin, D.; and Adams, R. 2016. Early stopping as nonparametric variational inference. In *Artificial Intelligence and Statistics (AISTATS)*, 1070–1077.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2019. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*.

Edelkamp, S. 2014. Planning with pattern databases. In *Sixth European Conference on Planning*, 84–90.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-Quality Policies for the Canadian Traveler's Problem. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 51–58.

Ferber, P.; Geiber, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2021. Neural Network Heuristic Functions for Classical Planning: Reinforcement Learning and Comparison to Other Methods. In *2nd PRL Workshop, International Conference on Automated Planning and Scheduling*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural network heuristics for classical planning: A study of hyperparameter space. In *European Conference on Artificial Intelligence*, 2346–2353.

Fern, A.; Koul, A.; Issakkimuthu, M.; and Tadepalli, P. 2018. A2C-Plan: A Reinforcement Learning Planner. *IPPC-6 planner abstracts*.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Florensa, C.; Held, D.; Wulfmeier, M.; Zhang, M.; and Abbeel, P. 2017. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300*.

Francés, G.; and Ramírez, M. 2021. Tarski - An AI Planning Modeling Framework. https://github.com/aig-upf/tarski.

Frances, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence*, 4294–4301.

Geffner, H. 2018. Model-free, model-based, and general intelligence. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 10–17.

Geffner, H.; and Bonet, B. 2013. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1): 1–141.

Geffner, T.; and Geffner, H. 2015. Width-based Planning for General Video-Game Playing. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference*, 23–29.

Geißer, F.; and Speck, D. 2018. Prost-DD–utilizing symbolic classical planning in THTS. *IPPC-6 planner abstracts*.

Geißer, F.; Speck, D.; and Keller, T. 2019. An Analysis of the Probabilistic Track of the IPC 2018. *Workshop on the International Planning Competition (WSIPC)*.

Ginsberg, M. L. 1999. GIB: Steps toward an expert-level bridge-playing program. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 584–593.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep learning*. MIT press.

Guo, X.; Singh, S.; Lee, H.; Lewis, R. L.; and Wang, X. 2014. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. In *NIPS*.

Hardt, M.; and Recht, B. 2021. *Patterns, predictions, and actions: A story about machine learning*. https://mlstory.org.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2): 1–187.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *International Conference on Automated Planning and Scheduling*, 176–183.

Helmert, M.; Röger, G.; et al. 2008. How Good is Almost Perfect?. In *AAAI*, volume 8, 944–949.

Hessel, M.; Modayil, J.; Van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.; and Silver, D. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 3215–3222.

Hester, T.; Vecerík, M.; Pietquin, O.; Lanctot, M.; Schaul, T.; Piot, B.; Horgan, D.; Quan, J.; Sendonaris, A.; Osband, I.; Dulac-Arnold, G.; Agapiou, J.; Leibo, J. Z.; and Gruslys, A. 2018. Deep Q-learning From Demonstrations. In *AAAI*.

Hill, A.; Raffin, A.; Ernestus, M.; Gleave, A.; Kanervisto, A.; Traore, R.; Dhariwal, P.; Hesse, C.; Klimov, O.; Nichol, A.; Plappert, M.; Radford, A.; Schulman, J.; Sidor, S.; and Wu, Y. 2018. Stable Baselines. https://github.com/hill-a/stable-baselines.

Ho, J.; and Ermon, S. 2016. Generative adversarial imitation learning. *Advances in Neural Information Processing Systems (NIPS)*, 4565–4573.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Hosu, I.-A.; and Rebedea, T. 2016. Playing Atari Games with Deep Reinforcement Learning and Human Checkpoint Replay.

Jinnai, Y.; and Fukunaga, A. 2017. Learning to prune dominated action sequences in online black-box planning. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, volume 31, 839–845.

Junyent, M.; Gómez, V.; and Jonsson, A. 2021. Hierarchical Width-Based Planning and Learning. In *Proc. of the Int'l Conf. in Automated Planning and Scheduling (ICAPS)*, volume 31, 519–527.

Junyent, M.; Jonsson, A.; and Gomez, V. 2019. Deep Policies for Width–Based Planning. In *Proc. of the Int'l Conf. in Automated Planning and Scheduling (ICAPS)*.

Karaman, S.; and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research (IJRR)*, 30(7): 846–894.

Katz, M.; Lipovetzky, N.; Moshkovich, D.; and Tuisov, A. 2017. Adapting novelty to classical planning as heuristic search. In *International Conference on Automated Planning and Scheduling*, 172–180.

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic planning based on UCT. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

Keller, T.; and Helmert, M. 2013. Trial-based heuristic tree search for finite horizon MDPs. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Keyder, E.; and Geffner, H. 2008. The HMDPP planner for planning with probabilities. *Sixth International Planning Competition at ICAPS*, 8.

Kingma, D. P.; and Ba, J. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Knuth, D. E. 1975. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129): 122–136.

Kober, J.; Bagnell, J. A.; and Peters, J. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research (IJRR)*, 32(11): 1238–1274.

Kocsis, L.; and Szepevari, C. 2006. Bandit Based Monte Carlo Planning. In *Proc. of European Conference in Machine Learning (ECML)*, 282–293.

Konidaris, G.; and Barto, A. 2009a. Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining. In Bengio, Y.; Schuurmans, D.; Lafferty, J.; Williams, C.; and Culotta, A., eds., *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc.

Konidaris, G.; and Barto, A. 2009b. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in neural information processing systems*, 22: 1015–1023.

Lei, C.; and Lipovetzky, N. 2021. Width-Based Backward Search. In *International Conference on Automated Planning and Scheduling*, 219–224.

Liang, Y.; Machado, M. C.; Talvitie, E.; and Bowling, M. H. 2016. State of the Art Control of Atari Games Using Shallow Reinforcement Learning. In *Proc. of Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 485–493. ACM.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *European Conference on Artificial Intelligence*, 540–545.

Lipovetzky, N.; and Geffner, H. 2017. Best-first width search: Exploration and exploitation in classical planning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Lipovetzky, N.; Ramírez, M.; and Geffner, H. 2015. Classical planning with simulators: results on the Atari video games. In *International Joint Conference on Artificial Intelligence*, 1610–1616.

Machado, M. C.; Bellemare, M. G.; Talvitie, E.; Veness, J.; Hausknecht, M.; and Bowling, M. 2018. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61: 523–562.

Mania, H.; Guy, A.; and Recht, B. 2018. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, 1800–1809.

Martin, J.; Narayanan, S.; Everitt, T.; and Hutter, M. 2017. Count-Based Exploration in Feature Space for Reinforcement Learning. IJCAI, AAAI Press.

McDermott, D. M. 2000. The 1998 AI planning systems competition. *AI magazine*, 21(2): 35–35.

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: 529.

Mundhenk, M.; Goldsmith, J.; Lusena, C.; and Allender, E. 2000. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM (JACM)*, 47(4): 681–720.

Nelson, M. J. 2021. Estimates for the Branching Factors of Atari Games. In *Proc. of the IEEE Conference on Games.*

Ng, A. Y.; and Russell, S. J. 2000. Algorithms for inverse reinforcement learning. *International Conference on Machine Learning (ICML)*, 1: 2.

Oh, J.; Guo, X.; Lee, H.; Lewis, R. L.; and Singh, S. 2015. Action-conditional video prediction using deep networks in atari games. In *NIPS*, 2863–2871. MIT Press.

Papadimitriou, C. H.; and Yannakakis, M. 1991. Shortest paths without a map. *Theoretical Computer Science*, 84(1): 127–150.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, 8024–8035.

Pearl, J. 1984. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Reif, J. H. 1979. Complexity of the mover's problem and generalizations. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 421–427.

Resnick, C.; Raileanu, R.; Kapoor, S.; Peysakhovich, A.; Cho, K.; and Bruna, J. 2018. Backplay:" Man muss immer umkehren". *arXiv preprint arXiv:1807.06919*.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Rintanen, J. 2008. Regression for classical and nondeterministic planning. In *European Conference on Artificial Intelligence*, 568–572.

Rosolia, U.; and Borrelli, F. 2017. Learning model predictive control for iterative tasks. a data-driven control framework. *IEEE Transactions on Automatic Control*, 63(7): 1883–1896.

Rubinstein, R. Y.; and Kroese, D. P. 2017. *Simulation and the Monte Carlo Method*. Wiley & Sons.

Salimans, T.; and Chen, R. 2018. Learning Montezuma's Revenge from a Single Demonstration. *Deep RL Workshop, NeurIPS*.

Sanner, S.; et al. 2010. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32: 27.

Sartipizadeh, H.; and Vincent, T. L. 2016. Computing the approximate convex hull in high dimensions. *arXiv preprint arXiv:1603.04422*.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Seipp, J.; Keller, T.; and Helmert, M. 2017. A comparison of cost partitioning algorithms for optimal classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, 259–268.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning domain-independent planning heuristics with hypergraph networks. In *International Conference on Automated Planning and Scheduling*, 574–584.

Shleyfman, A.; Tuisov, A.; and Domshlak, C. 2016. Blind search for atari-like online planning revisited. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 3251–3257.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In *International Symposium on Combinatorial Search*, volume 3.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Driessche, G. V. D.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; and others. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; and others. 2017. Mastering the game of Go without human knowledge. *Nature*, 550(7676): 354.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4): 72–82.

Sun, W.; Gordon, G. J.; Boots, B.; and Bagnell, J. 2018. Dual policy iteration. *Advances in Neural Information Processing Systems*, 31.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1): 9–44.

Sutton, R. S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, 216–224. Elsevier.

Sutton, R. S. 1991. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4): 160–163.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press, 2nd edition.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211.

Tang, H.; Houthooft, R.; Foote, D.; Stooke, A.; Chen, O. X.; Duan, Y.; Schulman, J.; DeTurck, F.; and Abbeel, P. 2017. # Exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in neural information processing systems*, 2753–2762.

Tibshirani, R. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1): 267–288.

Torabi, F.; Warnell, G.; and Stone, P. 2018a. Behavioral cloning from observation. *International Joint Conference on Artificial Intelligence (IJCAI)*, 4950–4957.

Torabi, F.; Warnell, G.; and Stone, P. 2018b. Generative adversarial imitation from observation. *arXiv preprint arXiv:1807.06158*.

Torabi, F.; Warnell, G.; and Stone, P. 2019. Recent advances in imitation learning from observation. *International Joint Conference on Artificial Intelligence (IJCAI)*, 6325–6331.

Trevizan, F.; Thiébaux, S.; and Haslum, P. 2017. Occupation measure heuristics for probabilistic planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Tuisov, A.; and Katz, M. 2021. The Fewer the Merrier: Pruning Preferred Operators with Novelty. In *IJCAI*.

Welch, B. L. 1947. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2): 28–35.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A Baseline for Probabilistic Planning. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 352–359.

Yu, L.; Kuroiwa, R.; and Fukunaga, A. 2020. Learning Search-Space Specific Heuristics Using Neural Network. *12th HSDIP Workshop, International Conference on Automated Planning and Scheduling*, 1–8.