Imperial College London
Department of Computing

# Learning Heuristic Functions and Search Policies for Classical Planning

Paweł Krzysztof Gomoluch

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, February 2020

## Declaration of Originality

I, Paweł Gomoluch, declare that this is my original work, except where explicitly stated, and that I have appropriately referenced the work of others.

## Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

## Abstract

Automated planning deals with the problem of composing sequences of actions, which, when executed from a given initial state, lead to the satisfaction of a given goal. Classical planning, in particular, relies on a known and deterministic model of the environment. In this thesis, we consider learning from the planner's experience as a way of improving its performance.

Modern classical planners are typically based on heuristic forward search. Such planners include two key components: a search routine, such as A* or greedy best-first search, and a heuristic function, which guides the search by evaluating any given state of the environment. A domain-independent heuristic function can be applied to problems from any planning domain.

The first part of the thesis investigates the use of machine learning techniques to obtain heuristic functions. The existing work on learning heuristics has focused on learning domain-specific functions. We propose a domain-independent approach, which allows the heuristics to be learned from data representing multiple domains and then deployed on any domain, thus preserving the generality of the planner. The learning task takes the form of supervised learning: solutions to a number of training problems are used to learn a heuristic function, which can then be applied for unseen problems and domains. We evaluate the approach experimentally and indicate its potential strengths and weaknesses.

In the second part of the thesis, we turn our attention to the search routine. The search routines used in classical planning are typically based on best-first search and remain fixed throughout the process of solving a problem. In this thesis, we construct search routines capable of changing the search approach while solving a problem. This allows us to formulate the task of learning search policies. The purpose of a search policy is to indicate the most suitable way of continuing the search given its current state. We consider two distinct approaches. The first one relies on a fixed set of search subroutines. The policy search problem translates to reinforcement learning with a discrete action space. The second one introduces a novel parametrised search algorithm, combining a variety of search techniques in a single routine. In this case, the search policy is responsible for setting the values of the search algorithm's parameters. This results in a learning task akin to reinforcement learning in continuous action spaces. Experimental evaluation shows that the learners are capable of discovering efficient search policies tailored to given distributions of planning problems.

## Acknowledgements

First, I would like to express my gratitude to my supervisors: Alessandra Russo, Dalal Alrajeh and Antonio Bucchiarone. This thesis would not be possible without their continual guidance and support.

I would also like to thank my colleagues and friends from offices 558b and former 572, for our illuminating discussions, steady encouragement during all the highs and lows of a PhD, and for all the laughs.

Last by not least, I would like to thank my family, especially my parents, for their consistent support throughout all stages of my education and studies.

# Contents

**Abstract** **3**

**Acknowledgements** **7**

**1 Introduction** **17**

1.1 Motivation . . . . . 19

1.2 Contributions . . . . . 20

1.2.1 Learning Heuristic Functions . . . . . 21

1.2.2 Learning Search Policies with a Finite Action Space . . . . . 21

1.2.3 Learning Search Policies with an Infinite Action Space . . . . . 21

1.3 Publications . . . . . 22

**2 Background** **23**

2.1 Classical Planning . . . . . 23

2.1.1 The Planning Domain Definition Language . . . . . 25

2.1.2 Planning as Forward Search . . . . . 29

2.1.3 Planning Heuristics . . . . . 30

2.1.4 Evaluating Planners' Performance . . . . . 34

2.1.5 Evaluation Domains . . . . . 35

2.2 Supervised Learning . . . . . 37

2.2.1 Ranking Support Vector Machines . . . . . 38

2.3 Reinforcement Learning . . . . . 39

2.3.1 Policy Gradient . . . . . 41

2.4 Evolution Strategies . . . . . 42

2.4.1 The Cross-Entropy Method . . . . . 42

**3 Related Work** **44**

3.1 Learning in Classical Planning . . . . . 44

3.2 Learning in Search . . . . . 46

3.3 Planning with Multiple Techniques . . . . . 46

3.4 Model-Based Reinforcement Learning . . . . . 48

3.5 Summary . . . . . 49

**4 Learning Heuristic Functions** **50**

4.1 Motivation . . . . . 51

4.2 Related Work . . . . . 52

4.3 Feature Representation . . . . . 52

4.4 Learning Algorithm . . . . . 55

4.5 Training Data . . . . . 56

4.6 Evaluation . . . . . 57

4.7 Summary . . . . . 61

**5 Learning Search Policies with a Finite Action Space 63**

5.1 Related Work . . . . . 64

5.2 Planning with Alternating Search Routines . . . . . 65

5.3 The Learning Framework . . . . . 67

5.4 Framework Instantiation . . . . . 70

5.4.1 Representation of the Planner's State . . . . . 70

5.4.2 Search Routines . . . . . 72

5.4.3 Reward Computation . . . . . 73

5.5 Evaluation . . . . . 76

5.5.1 Experimental Setup . . . . . 76

5.5.2 Learned Policies . . . . . 77

5.5.3 Results . . . . . 81

5.6 Summary . . . . . 86

**6 Learning Search Policies with an Infinite Action Space 90**

6.1 Relation to Finite Action Spaces . . . . . 91

6.2 Parameterized Search Routine . . . . . 92

6.3 Neural Search Policies . . . . . 95

6.3.1 State Representation . . . . . 95

6.3.2 Model Architecture . . . . . 96

6.4 Learning Search Policies . . . . . 98

6.4.1 Learning the Search Parameters Directly . . . . . 99

6.5 Evaluation . . . . . 100

6.5.1 Training . . . . . 101

6.5.2 Quantitative Evaluation . . . . . 103

6.5.3 Qualitative Evaluation . . . . . 105

6.6 Summary . . . . . 109

**7 Conclusion** **110**

7.1 Summary of Thesis Achievements . . . . . 110

7.2 Future Work . . . . . 111

**Bibliography** **112**

# List of Tables


4.1 IPC scores for heuristics learned with FF features (small problems) . . . . . 58

4.2 IPC scores for heuristics learned with FF features (IPC problems) . . . . . 60

4.3 IPC scores for heuristics learned with CEA features (small problems) . . . . . 60

4.4 IPC scores for heuristics learned with CEA features (IPC problems) . . . . . 61

5.1 The policies learned for the *Transport* problems . . . . . 77

5.2 The policies learned for the *Parking* problems . . . . . 78

5.3 The policies learned for the *Elevators* problems . . . . . 79

5.4 The policies learned for the *No-mystery* problems . . . . . 80

5.5 The policies learned for the *Floortile* problems . . . . . 80

5.6 Performance on test problems . . . . . 85

5.7 Performance on IPC problems . . . . . 87

5.8 Performance on IPC problems, 60s time limit . . . . . 88

6.1 IPC scores for IPC problem sets . . . . . 105

6.2 IPC scores for randomly generated test problems . . . . . 106

6.3 The optimized search parameters . . . . . 107

# List of Figures

1.1 Planning and learning: an overview . . . . . 18

2.1 The initial state of an example planning problem . . . . . 26

2.2 The relaxed plan graph constructed by FF . . . . . 32

2.3 Example *Parking* problem . . . . . 36

5.1 Training on *Transport*, IPC reward, *near early* state . . . . . 81

5.2 Training on *Transport*, IPC reward, *far early* state . . . . . 82

5.3 Training on *Parking*, IPC reward, *near early* state . . . . . 82

5.4 Training on *Parking*, time reward, *near late* state . . . . . 83

5.5 Training on *Elevators*, IPC reward, *near early* state . . . . . 83

5.6 Training on *No-mystery*, time reward, *near late* state . . . . . 84

6.1 Average IPC scores on training problems . . . . . 102

# Chapter 1

# Introduction

Automated planning is concerned with finding sequences of actions which, when executed from a given initial state, lead to satisfaction of a given goal. Classical planning, in particular, assumes a known, static and deterministic model of the environment. A critical property of classical planning is the ability to solve problems from multiple, unrelated domains, provided that they are expressed in some common formalism, such as the *Planning Domain Definition Language* (PDDL) [MGH+98]. The generality of planning comes at the cost of its computational complexity. In fact, the problem of finding a plan is PSPACE-complete [Byl94]. A paramount objective of planning research has been to find approaches suitable for efficiently solving problems representing a wide range of planning domains. In this thesis, we consider learning from the planner's experience as a way of improving its performance.

A dominant paradigm in forward search planning is forward search in the space of possible environment states. A forward search planner starts at the initial state and iteratively explores the state space by applying available actions from the states reached so far. If the problem is solvable and the planner is complete, the search continues until a goal state is found or the planner runs out of resources, such as time or memory.

At a very high level, the two key components of a forward search planner are the search routine and a heuristic function. The search routine, for example greedy best-first search or A*, uses the heuristic function to determine the order in which the environment states are explored. The

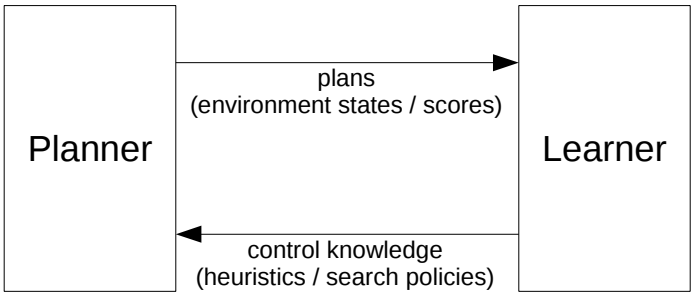



Figure 1.1: Planning and learning: an overview.

heuristic itself is a function mapping any given environment state to a scalar value, which is often interpreted as an estimate of the distance from the evaluated state to the goal. Heuristic functions help guide the planner towards more promising states and, eventually, towards the goal.

In the first part of the thesis, we investigate the use of machine learning to automatically acquire heuristic functions for classical planning. Then, in the second part, we focus on using learning in the search component of the planner. In particular, we introduce the task of learning *search policies* in action spaces comprising multiple forward search techniques, such as greedy search, local search and random walks. We present two variants of the task, along with reinforcement learning approaches for solving them.

Figure 1.1 is a high-level depiction of our learning planning systems. In all cases, the results of planner runs are used to learn control knowledge that can be used while solving new problems. When learning the heuristic functions, the learning process takes the form of a supervised learning task. Plans for a number of training problems are computed using existing planners. Environment states lying along the plan trajectories are used to learn a heuristic function. The learned function can then be used for evaluating unseen states, encountered by a planner while solving new problems. For learning the search policies, we adopt a reinforcement learning setting, in which the training process involves multiple runs of the planner using the current

search policy (or samples from the current distribution over search policies). The results of the runs, such as information about whether the problem was solved and the cost of the plan, are used to update the search policy (or a distribution over policies).

## 1.1 Motivation

All methods introduced in this thesis share the objective of learning from the planner's experience in order to improve its future performance. The first approach we consider is learning heuristic functions. Heuristic functions can be learned using examples of environment states for which the distance to the goal is known, for example states extracted from plans computed using existing planners. This idea has been explored in the literature [YFG08, XFY10, VBA13, GKLP16]. However, in these approaches, learning has been confined to domain-specific settings. While learning in a domain-specific setting can improve the performance of the planner in the given domain, in many ways it is at odds with the general nature of domain-independent planners, which are expected to be able to handle problems from any domain. For example, when a planner is used as a problem-solving component of a dynamic system, the planning domains it faces may constantly change [BSM+16, BMPR17]. In such settings, tying the learned knowledge to a particular domain would severely restrict its usefulness. In this thesis, we explore the problem of learning *domain-independent* heuristic functions. In line with the state of the art in learning domain-dependent heuristics, our learning approach uses features extracted from graphs computed by existing heuristics. In a sense, the learning objective in this setting is the correction of the deficiencies in the original heuristic. While existing work does it in a per-domain setting, we introduce a domain-independent approach, in which the learned heuristic can be applied to unseen domains.

The second part of the thesis is focused on learning search policies, whose task is to choose the most suitable search approach, given the state of the search. This is a novel learning task, based on the idea of combining multiple search techniques within a single search run and switching between them as necessary. The existing work on search alternating between multiple

techniques is typically limited to hand-crafted approaches, in which the primary search mode (such as greedy search) is supported with a single exploratory mechanism (such as random walks) when a specific, human-defined criterion is met [XMH14, LG17].

In contrast, our approaches include multiple search techniques in the planner without manually selecting the primary and exploratory technique. Rather than that, the task of choosing the most suitable one is handled by the learned policy. The first variant of our approach uses a discrete action space comprising a number of *fixed* search routines. We then take the idea further and construct a parametrized search routine template, which combines multiple techniques in a manner defined by the values of its parameters. For specific values of the parameters, the routine can resolve to, for example, standard best-first search or iterated local search. For intermediate values of the parameters, various combinations of the above are possible, optionally supported by random walks and randomized order of node expansion. One of the key advantages of this setting is that it allows the learning process to optimize quantities such as the length of random walks and the span of the local search, rather than just choose between fixed, predefined routines. This enables the learning process to explore a wide range of search approaches in order to find a policy best suited to a given problem distribution. Learning search policies tailored to a given problem distribution and performance objective can be seen as bridging the gap between general planners, designed to work for a large variety of problems, and domain-specific solvers, handcrafted by human experts for a particular problem type.

## 1.2 Contributions

In this section, we outline the contributions of this thesis, organized by chapters. Each of the contributions is accompanied by an open source implementation based on the Fast Downward planning system [Hel06], which is available online[1].

[1] https://github.com/pgomoluch/fd-learn

### 1.2.1 Learning Heuristic Functions

To the best of our knowledge, we present the first attempt to learn domain-independent heuristic functions. Our approach builds on the state-of-the-art approach to learn domain-dependent heuristics [YFG08, GKLP16]: we base our feature representation on the computation of existing relaxed-plan heuristics and employ a ranking approach for learning the heuristic function. Our original contributions include: construction of domain-independent features characterizing the environment state, building a training data set, and empirical evaluation of the approach, in which we show its potential strengths and weaknesses.

### 1.2.2 Learning Search Policies with a Finite Action Space

Our first approach to learning search policies is based on a discrete action space, comprising multiple search routines. The contributions of this chapter include a novel search algorithm capable of switching between multiple search routines and a policy gradient framework for learning stochastic search policies, which choose the search routine given the state of the search. We provide a simple instantiation of the framework comprising a two-dimensional search state representation and five different search routines. The experimental evaluation shows that the approach is capable of learning various search policies which match the performance of the best performing baselines in each of the domains and outperform them on aggregate.

### 1.2.3 Learning Search Policies with an Infinite Action Space

Finally, we present a different way of learning search policies, based on a parametrized template of a search routine. Depending on the values of the parameters, the parametrized search routine can perform best-first search, iterated local search or various combinations of the two, supported with random walks and randomized order of node expansion. In this setting, the search policies take the form of a function mapping the state of the search to values of the routine's parameters. We present a *cross-entropy method* approach to learning search policies

tailored to a given distribution of the planning problems. We show that the same evolutionary approach can also be used to learn the values of the search parameters directly, without a state-dependent policy. We provide an empirical evaluation of both methods and show that they are capable of finding effective search policies, outperforming the relevant baselines.

## 1.3 Publications

Many of the ideas and results presented in this thesis have been published in the following papers:

- Paweł Gomoluch, Dalal Alrajeh, Alessandra Russo and Antonio Bucchiarone. *Towards learning domain-independent planning heuristics.* The Workshop on Architectures for Generality and Autonomy at the International Joint Conference on Artificial Intelligence (IJCAI) 2017. [GARB17]
- Paweł Gomoluch, Dalal Alrajeh, Alessandra Russo. *Learning Classical Planning Strategies with Policy Gradient.* International Conference on Automated Planning and Scheduling (ICAPS) 2019. [GAR19]
- Paweł Gomoluch, Dalal Alrajeh, Alessandra Russo and Antonio Bucchiarone. *Learning Neural Search Policies for Classical Planning. (to appear).* Accepted for the International Conference on Automated Planning and Scheduling (ICAPS) 2020. [GARB20]

# Chapter 2

# Background

In this chapter, we introduce the relevant planning and machine learning terminology, to which we refer throughout this thesis. The chapter is structured as follows. We first present background on classical planning, including a formal definition of a classical planning problem and some existing algorithms used to solve it, with a particular focus on the techniques we build upon in this thesis. We then describe the relevant machine learning problems and techniques. We finally present the evolutionary optimization algorithms, which we employ in Chapter 6.

## 2.1 Classical Planning

Planning is the act of reasoning about future actions prior to executing any of them. The aim of planning is to construct *plans*, which indicate, action by action, what steps need to be taken in order to achieve a given goal from a given initial state. Classical planning, in particular, assumes a known, static and deterministic model of the environment, described in terms of discrete variables. The planning approaches considered in this thesis rely on forward search through the space of possible environment states. The plans constructed by forward-search planners are linear sequences of actions (also referred to as *operators*), which, executed in order, lead from a given initial state to a state in which the goal is satisfied.

One of the first formalizations of the classical planning task was the propositional STRIPS problem definition, associated with a problem-solving program by the same name [FN71]. A similar propositional representation is used by the *Planning Domain Definition Language* (PDDL) [MGH+98], which is the standard way of describing a planning task. PDDL is presented in more detail in Section 2.1.1. For the purpose of this thesis, we formally define the classical planning problem using the *multi-valued planning task* (MPT) formalism [BN95, Hel06]. Before we formally define the classical planning problem (Definition 2.2), we introduce the necessary concepts in Definition 2.1.

**Definition 2.1** *Assume that $\mathcal{V}$ is a set of variables, where each variable $v \in \mathcal{V}$ is associated with a domain $D_v$. A **partial assignment** over $\mathcal{V}$ is a function $s$ defined on some subset $\mathcal{V}_s$ of $\mathcal{V}$, such that, for every $v \in \mathcal{V}_s$, $s(v) \in D_v$. A partial assignment $s$ over $\mathcal{V}$ is an **assignment** (also referred to as a **state**) over $\mathcal{V}$ if and only if $\mathcal{V}_s = \mathcal{V}$. A partial assignment $s$ over $\mathcal{V}$ **satisfies** a partial assignment $s'$ over $\mathcal{V}$ if and only if $\mathcal{V}_{s'} \subseteq \mathcal{V}_s$ and $s(v) = s'(v)$ for every $v \in \mathcal{V}_{s'}$.*

**Definition 2.2** *A **classical planning problem** is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, g \rangle$, whose components are defined as follows:*

- *$\mathcal{V}$ is a finite set of variables; each variable $v \in \mathcal{V}$ is associated with a finite domain $D_v$;*
- *$\mathcal{O}$ is a set of operators, each operator $o \in \mathcal{O}$ is defined by its preconditions $\mathrm{pre}(o)$ and effects $\mathrm{eff}(o)$, which are partial assignments over $\mathcal{V}$; an operator $o \in \mathcal{O}$ can be applied in a state $s$ if and only if $s$ satisfies $\mathrm{pre}(o)$; the state resulting from applying operator $o$ in state $s$, denoted as $s' = o(s)$ is an assignment over $\mathcal{V}$ such that $s'(v) = \mathrm{eff}(o)(v)$ for every $v \in \mathcal{V}_{\mathrm{eff}(o)}$ and $s'(v) = s(v)$ for every $v \in \mathcal{V} \setminus \mathcal{V}_{\mathrm{eff}(o)}$*
- *$s_0$ is the initial state, which is an assignment over $\mathcal{V}$;*
- *$g$ is the goal, which is a partial assignment over $\mathcal{V}$.*

In classical planning with action costs, the problem definition is augmented to include a cost function: $\Pi_c = \langle \mathcal{V}, \mathcal{O}, s_0, g, c \rangle$, where $c : \mathcal{O} \to \mathbb{R}_+ \cup 0$ is a function assigning a nonnegative scalar cost to each operator $o \in \mathcal{O}$.

**Definition 2.3** *A **plan** is a sequence of operators $o_0, o_1 ... o_n \in \mathcal{O}$, such that applying the operators in order, starting from the initial state $s_0$, leads to a state that satisfies the goal $g$:*

$$g \subseteq o_n(o_{n-1}(...o_0(s_0)...))$$

*A plan is a solution of a classical planning problem. In planning with action costs, the cost of a plan is the sum of the costs of the operators used in the plan $\sum_{i=0}^{n} c(o_i)$. A plan is optimal if and only if its cost is minimal among all the plans solving the problem. In planning without action costs, a plan is optimal if and only if its length is minimal among all the plans solving the problem.*

While a propositional representation using binary variables is automatically an MPT one (where all variables are binary), using variables that can take more than two values often allows for a more practical representation of a planning task. For this reason, the multi-valued representation lies at the core of many modern planners, including the Fast Downward planning system [Hel06]. A binary representation of a planning task can be efficiently translated to a more compact multi-valued one by using invariant synthesis to detect groups of propositional variables which can be replaced with a single variable with larger domain [Hel06].

A planning setting that requires finding optimal plans is referred to as *optimal* planning. Planning without the optimality requirement is known as *suboptimal* or *satisficing* (from *satisfy* and *suffice*) planning. In this thesis, we focus on satisficing planning.

### 2.1.1 The Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) [MGH+98] is the standard language for describing classical planning problems. PDDL is a declarative language allowing for the defi-

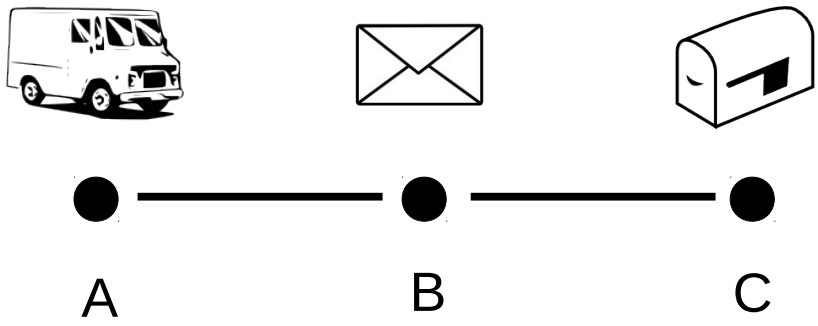



Figure 2.1: The initial state of an example planning problem.

nition of predicates and typed objects. Predicates with variables are used to define operator schemata. Operators are obtained from the operator schemata by grounding, i.e., substituting objects for the variables in the schema definition. A PDDL specification typically comes in two parts referred to as the planning *domain* and the *problem*. The planning domain specifies the predicates and operator schemata. The *problem* is an instance of the planning domain. The problem description defines the initial state, the goal and the objects existing in the environment.

As an example, let us consider the planning problem illustrated in Figure 2.1. The problem represents a simplified version of the *Transport* domain from the *International Planning Competition* (IPC). The task is to deliver the package (initially at location B) to its destination C. The package can be transported using a truck, initially located at A.

Listing 2.1 shows a PDDL description of the domain. The description starts by stating the features of PDDL used in the specification. In this case, the only requirement is *typing* which allows for building object type hierarchies. The object types are used to restrict the types of predicate arguments. In our example, there are two types (*location* and *locatable*) derived directly from *object*. The *locatable* type is used as a base type for the *vehicle* and *package* types.

The definition of types is followed by definitions of predicates. In this case, there are three predicates (*road*, *at* and *in*), each with two arguments. The $road(l_1, l_2)$ predicate is used to

**Listing 2.1** A PDDL domain example.

```
(define (domain transport)
  (:requirements :typing)
  (:types
        location locatable - object
        vehicle package - locatable
  )
  (:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?l - location)
     (in ?p - package ?v - vehicle)
  )

  (:action drive
    :parameters (?v - vehicle ?l1 ?l2 - location)
    :precondition (and
        (at ?v ?l1)
        (road ?l1 ?l2)
      )
    :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)
      )
  )
 (:action pick-up
    :parameters (?v - vehicle ?l - location ?p - package)
    :precondition (and
        (at ?v ?l)
        (at ?p ?l)
      )
    :effect (and
        (not (at ?p ?l))
        (in ?p ?v)
      )
  )
  (:action drop
    :parameters (?v - vehicle ?l - location ?p - package)
    :precondition (and
        (at ?v ?l)
        (in ?p ?v)
      )
    :effect (and
        (not (in ?p ?v))
        (at ?p ?l)
      )
  )
)
```

state that there is a road between locations $l_1$ and $l_2$. The $at(x, l)$ predicate can express the fact that a *locatable* object $x$ (such as a vehicle or a package) is at location $l$. The $in(p, v)$ predicate states that the package $p$ is in the vehicle $v$.

The predicates are used to specify the operator schemata (*actions*) in terms of their parameters, preconditions and effects. The *drive* action has three arguments specifying the vehicle and the locations between which it is driven. The preconditions of the action are that the vehicle is present at the initial location and that there is a road connecting the initial location with the destination. The effect of the action is that the vehicle disappears from its initial location and appears at the destination. The *pick-up* action is used to load the package to a vehicle when both are present at the same location. The action *drop* unloads a package from a vehicle.

The problem itself is described in Listing 2.2. It defines three locations (*a*, *b* and *c*), as well as one vehicle (*truck1*) and one package (*package1*). The initial state lists the roads between *a* and *b*, as well as *b* and *c* (in both directions). The package is initially located at *b* and the truck is located at *a*. The goal is to get the package to location *c*.

**Listing 2.2** A PDDL problem example.

```
(define (problem transport1)
 (:domain transport)
 (:objects
  a - location
  b - location
  c - location
  truck1 - vehicle
  package1 - package
 )
 (:init
  (road a b)
  (road b a)
  (road b c)
  (road c a)
  (at package1 b)
  (at truck1 a)
 )
 (:goal (and
  (at package1 c)
 ))
)
```

### 2.1.2 Planning as Forward Search

A common paradigm in classical planning is forward search. A forward search planner starts at the initial state $s_0$ and iteratively examines states reachable by applying the operators applicable in the visited states, until a state which satisfies the goal $g$ is reached. High-level pseudocode of a forward search planner is presented in Algorithm 1.

The inputs of the algorithm are the initial state $s_0$, the goal $g$ and the set of operators $\mathcal{O}$. The algorithm starts by initializing the *open list* with the initial state (line 2). The main loop of the algorithm (lines 3-14) consists of removing a state from the open list (line 4) and using all the operators applicable in the state (line 5) to reach descendant states which are then added to the open list (line 11). The algorithm terminates when a state satisfying the goal is reached (lines 7-9). If the open list becomes empty without reaching the goal, the search fails (line 15). Note that the search algorithm is presented here in a simplified form. We omit detail relating to detecting duplicate states and storing information about how each of the states was reached (i.e., the ancestor state and the operator applied). One way to read the algorithm is to assume that the open list is actually a richer data structure that stores the ancestor information and is capable of detecting and possibly rejecting the duplicate states.

**Algorithm 1** Planning by forward search

```
 1: function PLAN(s_0, g, O)
 2:     open_list ← [s_0]
 3:     while queue is not empty do
 4:         s_c ← next(open_list)
 5:         for all o ∈ O applicable in s_c do
 6:             s ← o(s_c)
 7:             if s ⊨ g then
 8:                 plan ← extract-ancestors(s)
 9:                 return plan
10:             else
11:                 insert(open_list, s)
12:             end if
13:         end for
14:     end while
15:     return failure
16: end function
```

The effective behaviour of a planner following Algorithm 1 depends on the ordering of states

in the open list. If the list is a FIFO queue, the search becomes breadth-first search. If the list is a LIFO queue (a stack), the search is depth-first search (DFS). In best-first search (BFS), the list is ordered using a heuristic function, which maps a state to a scalar value, with lower values assigned to states which are deemed closer to the goal. Search using a list ordered solely by increasing values of the heuristic is called *greedy* best-first search (GBFS). If the ordering instead uses a sum of the heuristic and the cost of the path from the initial state to the evaluated state, the resulting search algorithm is known as A*. In the following section, we discuss the concept of a heuristic function in more detail.

### 2.1.3 Planning Heuristics

In planning, a *heuristic* is a function that maps an environment state to a scalar value: $h : \mathcal{S}_\mathcal{V} \to \mathbb{R}$, where $\mathcal{S}_\mathcal{V}$ is the set of states in the given planning problem (a set of possible assignments over $\mathcal{V}$). The purpose of a heuristic function is to guide the search towards the more promising states, for which the heuristic is generally expected to return lower values. The value of the heuristic function is often interpreted as the estimated distance from the evaluated state to the goal. Indeed, an oracle knowing the exact distance to the goal from any given state would be a perfect heuristic, linearly guiding the search to the goal (every expanded state would lie on a direct path to the goal). Designating the distance to the goal is generally as hard as solving the planning problem itself. For this reason, the heuristic functions used in practice need to trade off the quality of the distance estimate with the time required to compute it.

A very simple example of a heuristic function is the *goal count* heuristic, which returns the number of variables whose value in the evaluated state is different from the value required by the goal $g$. For the example problem from Figure 2.1, the value of goal count heuristic is 0 for states in which proposition *at(package1,c)* holds. For all other states the value of the heuristic is 1. In a larger problem, with more packages to deliver, the value of the heuristic would be equal to the number of packages not yet delivered.

Classical planning heuristics have been an area of active research and many effective heuristic functions have been proposed [HN01, Hel06, HG08, RW10, DHK15]. In the following subsection,

we describe the *Fast Forward* heuristic, one the most successful heuristic functions in recent years. The Fast Forward heuristic is used extensively in the evaluation of the contributions presented in this thesis.

### The Fast Forward Heuristic

The *Fast Forward* heuristic (FF) [HN01] is a key component of a successful planner by the same name. The FF heuristic belongs to the family of *relaxation* heuristics. The key idea behind relaxation heuristics is to efficiently compute a solution to a simplified version of the planning problem at hand (i.e., the problem where the initial state is the state under evaluation and the goal is the same as the original goal $g$). The cost or length of the plan computed for the reduced problem is then used as the heuristic value for the evaluated state.

The design of a relaxation heuristic involves two key steps. The first is to decide how to simplify the planning problem. The second is what algorithm to use to efficiently solve the reduced problem. The FF heuristic is an instance of a *delete-relaxation* heuristic. The reduced problem is obtained by removing the negative effects (also known as the *delete* effects) of the operators from the propositional representation of the original planning task. For instance, the action *drive(truck1,a,b)* from the example problem *adds* the proposition $at(truck1, b)$ and *deletes* $at(truck1, a)$. With the *delete* effect removed, the action only adds the truck at location $b$, which means that after applying the action the truck exists in both locations simultaneously. Intuitively, this makes solving the problem much easier, since the set of true propositions grows monotonically with every operator application, eventually leading to a state in which each of the goal propositions holds. In the case of FF, the relaxed plan is solved using a variant of the *Graphplan* algorithm [BF97]. Solving the relaxed planning task is possible in polynomial time [HN01].

**Example** To illustrate the computation of the Fast Forward heuristic, let us consider the evaluation of the initial state of the example problem from Figure 2.1. The *delete-relaxed* version of the problem is obtained by removing the *delete* effects from all the operators. The

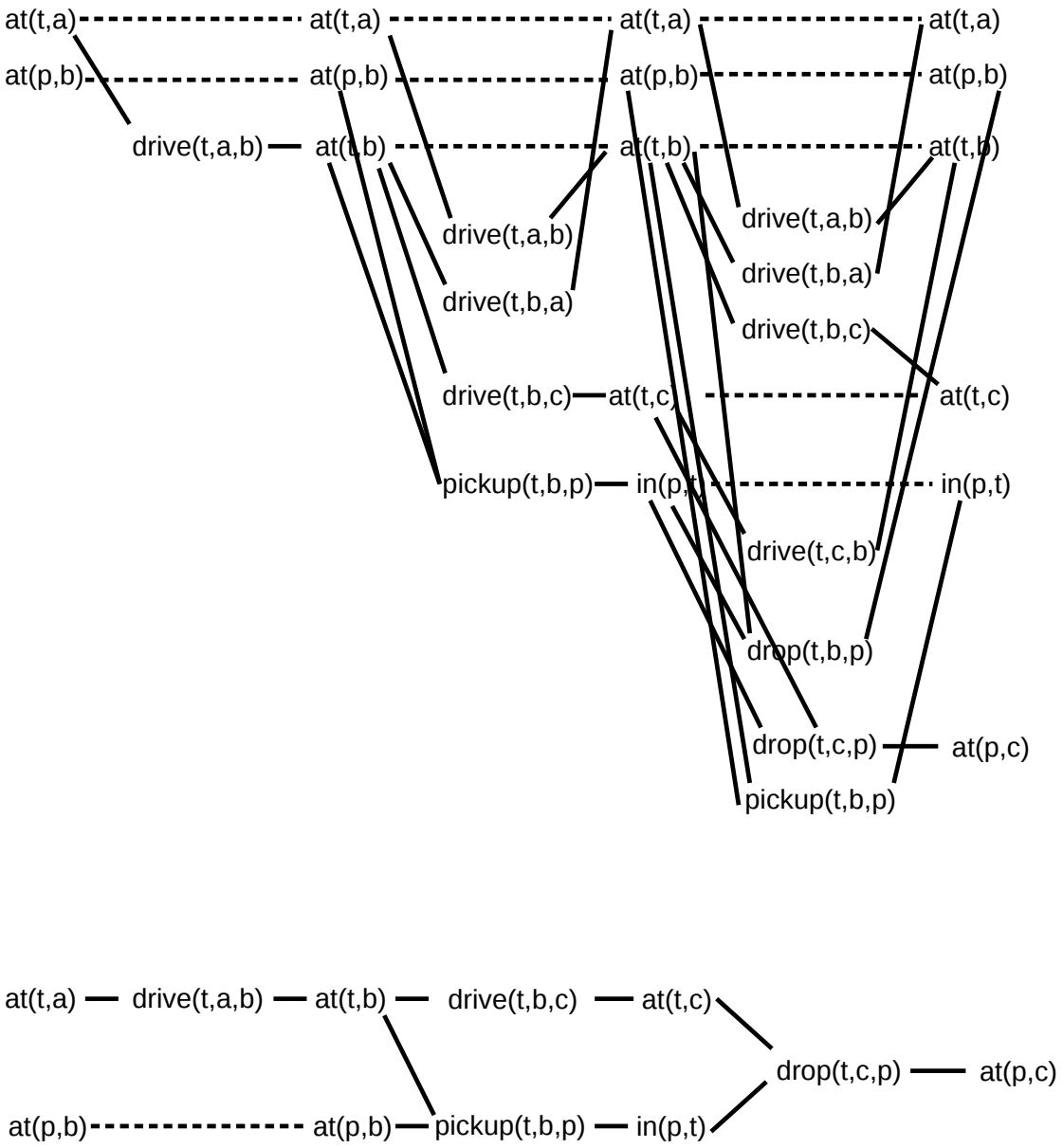



Figure 2.2: The relaxed plan graph constructed by FF.

relaxed plan is then solved using the Graphplan algorithm. The graph constructed to compute the heuristic value is shown in Figure 2.2. The top of the figure depicts the full graph constructed for the relaxed problem. The graph is organized in layers, interleaving layers of propositions with layers of operators. The solid lines connect each of the operators with its preconditions and effects. The dashed lines represent additional *no-op* operators, modelling the persistence of propositions between layers. The graph is constructed left to right. The leftmost proposition layer represents the initial state. Every action layer includes all the operators applicable in the state given by the preceding proposition layer. The next proposition layer augments the previous one with propositions achieved by at least one of the applicable operators. The process is repeated until a layer containing all the goal propositions is reached or until the proposition layer stops growing. If the final layer contains all the goal propositions, the relaxed plan is extracted by starting at the final layer and traversing the graph backwards, selecting operators that achieved each of the goal propositions, then examining their preconditions to add the actions achieving each of them and so on, until the initial layer is reached. The graph at the bottom of Figure 2.1 shows the subgraph of the full graph, including all the selected operators. The subgraph represents a partially ordered relaxed plan: the operators in the same layer can be executed in any order. Note that in this case, the plan allows driving from $b$ to $c$ before picking the package at $b$. This is possible in the relaxed problem because of the *delete relaxation*: the truck does not disappear from $b$ after moving to $c$, so the preconditions of the *pickup* action still hold. The number of operators in the relaxed plan (respectively: the sum of their costs) is the heuristic estimate returned for the evaluated state. In this case, the heuristic value is 4, which is the exact distance from the initial state to the goal.

In the case of an MPT representation of the planning task, the *delete-relaxation* amounts to allowing each of the multi-valued variables to take multiple values simultaneously [Hel06]. For example, consider a variable *truck1-location* for the example problem from figure 2.1. The variable can take three values: $a$, $b$ or $c$. In the relaxed plan, the result of applying action *drive(truck1,a,b)* is that the set of values associated with variable *truck1-location* grows from $\{a\}$ to $\{a, b\}$.

### 2.1.4 Evaluating Planners' Performance

The planner's performance can be evaluated using a number of metrics. One common approach is to use *coverage*, i.e., the number of problems solved by the planner within a time limit, out of a number of problems in the evaluation set. The time limit is usually per-problem, i.e., for every problem the planner receives a fixed time period, for example 30 minutes. Coverage alone is very informative in the case of optimal planning, in which all the planners solving the problem return a plan of equal (optimal) cost.

On the other hand, in suboptimal planning, the planners can return plans of varying cost. In this case, using coverage to measure the planner's performance would amount to ignoring the plan cost. While optimality is not required in this setting, plans of lower cost are typically preferred. The *International Planning Competition* (IPC) 2008[1] introduced a scoring approach which, for every problem, awards the planner a score defined as the ratio between the cost of the best known plan and the cost of the plan returned by the evaluated planner.

For a failed problem, the planner receives a score of 0. For a solved problem, the score $\psi$ is defined as:

$$\psi = \frac{c_{\min}}{c}$$

where $c_{\min}$ is the lowest known cost of a plan solving the problem and $c$ is the cost of the plan returned by the evaluated planner. In practice, $c_{\min}$ is often designated as the minimum over the costs of plans returned by all the compared planners.

For example, a planner which returns a plan of cost $c = 20$ for a problem for which a plan with a cost of $c_{\min} = 10$ is known, will receive a score of $\frac{10}{20} = 0.5$.

The planner's score for a set of problems is the sum of scores for individual problems:

$$\Psi = \sum_{p} \psi_p$$

Throughout the thesis, we use the IPC score as the primary way of assessing the learned

[1] http://icaps-conference.org/ipc2008/deterministic/

planner's performance. In the reinforcement learning setting of Chapters 5 and 6, the IPC score is also used to derive the reward functions for the training process.

### 2.1.5 Evaluation Domains

In this section, we provide a brief description of the IPC domains used for evaluating the learning approaches introduced in this thesis. The domains were used in the learning track of IPC 2014[2]. Previously, they also appeared in the deterministic part of IPC 2011 [LCO15]. Their PDDL definitions are also available in the online repository of planning domains[3].

#### Elevators

The *Elevators* domain is concerned with planning the routes of elevators in a multi-storey building, so that each of the involved passengers can reach their destination floor. Buildings feature two types of elevators. The *fast* elevators serve the full height of the building but can only stop on selected floors, skipping a number of floors in between. The *slow* elevators can stop on every floor but only serve some segment of the building (for example, floors 6-12). The actions of the planner correspond to the elevators moving up and down, as well as the passengers boarding and leaving the elevators.

#### Floortile

The *Floortile* domain involves planning for a set of robots whose task is to paint the floor in a certain pattern. The floor is represented as a grid of tiles. Each tail can be painted in a certain colour by a robot. The difficulty is that the robots are not allowed to step on a tile which has already been painted. This results in possible dead-end states of the environment. The planning agent controls all of the robots. The actions correspond to individual robots picking up paint of certain colour, painting tiles and moving around.

[2] `http://www.cs.colostate.edu/~ipc2014/`
[3] `https://github.com/potassco/pddl-instances/`

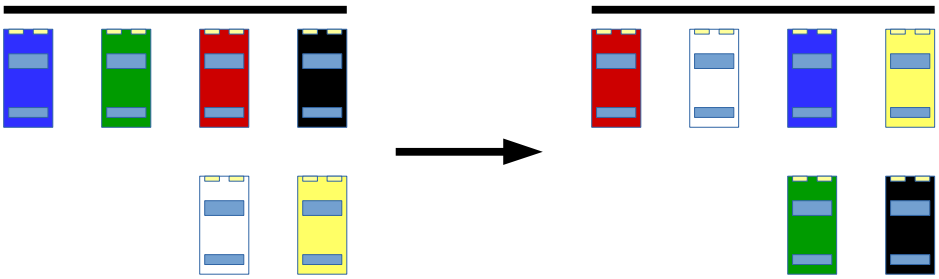


Figure 2.3: An example of a *Parking* problem. Six cars are parked in a total of eight parking slots next to four curbs. The left part of the figure is the initial state. The right part is the goal state.

### No-mystery

In the *No-mystery* domain, the task is to drive a truck between a number of locations in order to pick up packages and deliver them to their respective destinations. The truck has a limited amount of fuel, which is consumed by every driving action. The initial amount of fuel is determined during problem generation by invoking a domain-specific solver to find the length of the optimal route. The truck receives the amount of fuel necessary to cover the optimal route times a cost factor $c$, greater or equal 1 (for example, across IPC 2011 problems, $c$ varied between 1.1 and 1.5). This constraint ensures that the computed plans are at most $c$ times more expensive than an optimal one.

### Parking

In *Parking* problems, a number of cars is parked next to a number of curbs. Cars can be double parked but not triple parked. Moving a car that is situated between a curb and another car requires removing the other car first. The task is to rearrange cars from their initial locations to a certain target arrangement by moving them between valid parking slots. An example of a *Parking* problem is illustrated in Figure 2.3.

**Transport**

In the *Transport* domain, the task is to deliver a number of packages to their destinations using a number of trucks. Each truck has limited capacity. The actions correspond to moving a selected truck between two locations connected by a road, as well as loading and unloading packages. Unlike in the *No-mystery* domain, there is no limit on the distances the trucks can cover. As a result, greedy search can return plans much longer than the optimal ones.

## 2.2 Supervised Learning

Supervised machine learning is the type of machine learning concerned with learning to predict the value of a label (dependent variable) given the values of the features (independent variables). Supervised learning models are trained on labelled data, consisting of examples for which the values of both features and labels are known. The trained model is expected to correctly predict the values of the label for examples unseen during the training. The two most common types of supervised learning are *classification* and *regression*. In classification, the labels are categorical values, assigning each of the examples to a particular class. An example of classification is the detection of a particular type of artifact in images, i.e., classifying the images into two classes: the ones which contain the artifact and the ones which do not. On the other hand, in regression, the label values are numerical. An example of a regression problem is predicting the house prices based on features such as the floor area, distance to underground station, crime rating etc.

Another type of supervised learning is *learning to rank*. The objective here is to provide the correct ranking of a number of examples, e.g., to order a list of recommendations presented to a customer of an online service. Learning to rank is not necessarily disjoint from classification and regression. For example, an intermediate product of a ranking mechanism can be a scalar value for each of the examples. The value can then be used to sort them in ascending order. In such a case, the key difference between ranking and regression is the training objective: in regression, the model is expected to predict some real-world quantity as accurately as possible,

while, with a ranking objective, the predicted numbers are irrelevant, as long as they lead to a correct ordering of the samples. In many ways, a ranking setting more closely reflects the purpose of a heuristic function, which is only used to indicate that some environment states are preferable to others [GKLP16]. We follow the ranking approach in our work on learning heuristic functions, presented in Chapter 4.

### 2.2.1 Ranking Support Vector Machines

Support Vector Machines (SVMs) [BGV92] are among the most common supervised learning algorithms. When used for classification, SVMs work by constructing a separation plane which correctly divides the feature space into two classes while maintaining as large margin between the data points and the separation hyperplane as possible. Ranking Support Vector Machines (RSVMs) [Joa02] are a particular type of SVMs designed for learning to rank. The input to an RSVM is a sequence of samples $x_1, ..., x_n$, ordered according to the ground-truth ranking. Assuming a linear model, the learning process corresponds to solving the following optimization problem:

$$\min_{w} \quad ||w||^2 + C \sum_{i=1}^{n} \sum_{j=i+1}^{n} \xi_{ij}$$

subject to: $x_i^\top w \geq x_j^\top w + 1 - \xi_{ij} \quad \forall i, j : i < j$

$\xi_{ij} \geq 0 \quad \forall i, j$

where $w$ is the vector of trainable weights (one for each of the features), $C$ is a constant hyperparameter and $\xi_{ij}$ is the slack variable for the sample pair $\langle x_i, x_j \rangle$. The constraints state that for all pairs of samples $\langle x_i, x_j \rangle$, such that $i < j$ (i.e., $x_i$ is ranked higher than $x_j$), the value of the dot product $x_i^\top w$ should be higher than that of $x_j^\top w$ by at least one. This is to ensure that the value of the expression $x^\top w$ will correctly order the samples. The slack variables $\xi_{ij}$ in the constraints cater for cases when the separation criterion is not satisfied, i.e., $x_i^\top < x_j^\top + 1$, i.e., the margin between the examples is less than 1 or even negative. The optimization objective minimizing the sum over $\xi_{ij}$, ensures that such misranking cases are minimized.

By rearranging the constraints, we can obtain a problem corresponding to traditional formula-

tion of SVM for linear classification [Joa02]:

$(x_i - x_j)^\top w \geq 1 - \xi_{ij} \quad \forall i, j : i < j$

$\xi_{ij} \geq 0 \quad \forall i, j$

In this formulation, the examples for classification are pairs $\langle x_i, x_j \rangle$ and the task is to classify them as correctly or incorrectly ranked.

A trained RSVM model can provide rank an unseen example $x$ by computing $x^\top w$.

## 2.3 Reinforcement Learning

Reinforcement learning is a type of machine learning, in which an agent uses its own experience to learn from how to best interact with its environment. The agent observes how the state of the environment changes in response to the agent's actions. At every step, it also receives a scalar *reward* signal, informing the agent how well it is doing. A reinforcement learning agent aims to maximize the (possibly discounted) sum of rewards it is going to observe. The reinforcement learning setting is usually defined using Markov Decision Processes (MDPs).

**Definition 2.4** *A Markov Decision Process (MDP) is a tuple* $\langle \mathcal{S}, \mathcal{A}, p(s, a, s'), r(s, a), \gamma \rangle$*, whose components are defined as follows:*

- $\mathcal{S}$ *is the set of states of the environment;*
- $\mathcal{A}$ *is the set of actions which the learning agent can take;*
- $p(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *is the state transition function: the value of* $p(s, a, s')$ *is the probability of transitioning to state* $s'$ *given that action* $a$ *is taken in state* $s$*;*
- $r(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *is the reward function: the value of* $r(s, a)$ *is the expected reward received by the agent when it takes action* $a$ *in state* $s$*;*
- $\gamma \in [0, 1]$ *is the discount factor deciding how much an immediate reward is preferred to one delayed by one time step.*

The reinforcement learning tasks we consider in this thesis are *episodic*. This means that the agent's interaction with the environment is naturally divided into finite-length sequences of steps. These sequences are called *episodes*. In Chapters 5 and 6, where reinforcement learning is used to learn search policies, an episode corresponds to a single run of a planner on a single problem, which terminates when the problem is solved or when the search runs out of time. The objective of a reinforcement learning agent is to maximize the expected return $G$, defined as a discounted sum of rewards observed in an episode:

$$G = \sum_{t=1}^{T} \gamma^t r_t \tag{2.1}$$

where $T$ is the length of the episode and $r_t$ is the reward observed at time step $t$.

To maximize $G$, the agent learns a policy, which is a (possibly stochastic) mapping from the environment states to the agent's actions. A deterministic policy is a function $\pi_d : \mathcal{S} \to \mathcal{A}$, mapping an environment state $s$ to an action $a$. In the stochastic setting, a policy is a function $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\mathcal{A})$ the set of probability measures on $\mathcal{A}$. The probability assigned by policy $\pi$ to action $a$ given that the agent is in state $s$ is denoted as $\pi(a|s)$.

Many reinforcement learning algorithms are based on learning a *state value function* $V(s)$ or *state-action value function* $Q(s, a)$. $V(s)$ is the estimated return received from state $s$ to the end of the episode. $Q(s, a)$ is the estimated return from state $s$, given that the action taken from that state is $a$. Values of $Q(s, a)$ implicitly define the agent's policy: the policy optimal with respect to the $Q$ function is one that in every state chooses the action for which $Q(s, a)$ is the highest.

An alternative to computing the value functions is working with policy parameters directly. This approach is naturally suited to learning stochastic policies, unlike value-based methods, which result in deterministic policies. *Policy gradient* is a reinforcement learning approach which performs direct policy search using stochastic gradient ascent. We discuss policy gradient in more detail in Section 2.3.1.

### 2.3.1 Policy Gradient

Policy gradient methods use the recorded returns to directly update a representation of the agent's policy. The approach relies on the observation that the gradient of the agent's performance with respect to the policy parameters only depends on the gradient of the policy with respect to its parameters $\theta$, the state-action value function $q(s, a)$ and the state distribution $\mu(s)$, but not on its derivative [Wil92, SB18]:

$$\nabla_\theta \mathbb{E}G \propto \sum_s \mu(s) \sum_a q(s, a) \nabla \pi(a|s) \tag{2.2}$$

Intuitively, Equation 2.2 states that, for every state-action pair $\langle s, a \rangle$, the gradient of agent's performance is affected proportionally to: (i) how often the state is encountered ($\mu(s)$), (ii) the state-action value for the pair ($q(s, a)$), and (iii) the gradient of the action's probability in the considered state ($\nabla_\theta \pi(a|s)$).

The REINFORCE algorithm [Wil92] follows the policy gradient approach by computing a sample of $\nabla \mathbb{E}G$ after every episode and using it to update the policy parameters. In the case when nonzero reward is only observed at the end of the episode and is not discounted ($\gamma = 1$), the sample of the value function is simply the observed return $G$. The states occurring in the episode serve as a sample of $\mu(s)$. For every time step $i$ of the episode the policy parameters are updated using the following expression:

$$\Delta\theta = \alpha G \frac{\nabla_\theta \pi(a_i|s_i)}{\pi(a_i|s_i)} \tag{2.3}$$

where $\alpha$ is the learning rate, $G$ is the return for the episode, $s_i$ is the state at $i$-th step of the episode and $a_i$ is the action taken from that state. Division by $\pi(a_i|s_i)$ corrects for the fact that actions for which $\pi(a|s_i)$ is higher are sampled more frequently.

To reduce the variance of the parameter update, a *baseline* function $b(s)$ of the state, which itself does not depend on the policy parameters, is often subtracted from the observed return.

$$\Delta\theta = \alpha(G - b(s))\frac{\nabla_\theta \pi(a_i|s_i)}{\pi(a_i|s_i)} \tag{2.4}$$

A natural candidate for the baseline function $b(s)$ is the current estimates of the state value function. An example of such estimate is the average of all returns $G$ for episodes passing through state $s$. This approach is an instance of an *actor-critic* method, as it jointly learns the policy $\pi$ (the *actor*) and an estimate of the state value function $b$ (the *critic*).

In Chapter 5, a policy gradient approach is used to learn stochastic search policies, tasked with choosing search routines depending on the state of the search.

## 2.4 Evolution Strategies

*Evolution strategies* (ES) are black-box stochastic optimization methods built around the idea of maintaining a population of solutions and iteratively updating it based on relative performance of the solutions in the current population. Examples of evolution strategies include the *cross-entropy method* (CEM) [Rub99], *covariance matrix adaptation* (CMA) [HO01] and *Natural Evolution Strategies* (NES) [WSG+14]. In recent years, evolution strategies have been successfully applied to a number of policy search problems, including many (deep) reinforcement learning problems such as, for example, the Atari games [MRG03, SHCS17, CLH18, CMS+18]. Policy search with evolution strategies has a number of advantages compared to gradient-based methods. In particular, it is able to work with non-differentiable policy parametrizations, handle arbitrarily delayed rewards, and does not require learning of a value function. In Chapter 6, CEM is used to learn search policies in a mixed discrete-continuous action space generated by a parametrized search routine template.

### 2.4.1 The Cross-Entropy Method

The *cross-entropy* method (CEM) [Rub99, dBKMR05] is a member of the ES family of optimization algorithms. The approach maintains a parametrized distribution over the possible

solutions. At every iteration, the distribution is updated to maximize the likelihood of the highest-scoring solutions.

Formally, given a possibly stochastic function $s(\mathbf{x}), \mathbf{x} \in \mathcal{X}$, whose value we want to maximize, the cross-entropy method introduces an auxilliary distribution $f_{\mathbf{v}}(\mathbf{x})$ over $\mathcal{X}$. The distribution is itself parametrized by vector $\mathbf{v}$. At every iteration $t$, $n$ samples $\mathbf{x}_t^{(1)}, \mathbf{x}_t^{(2)} \dots \mathbf{x}_t^{(n)}$ are drawn from the current distribution and evaluated. To select $m$ best solutions ($m < n$), a threshold $\gamma_t$ is introduced, equal to the $\frac{m}{n} \cdot 100$-th percentile of the candidate scores $s(\mathbf{x}_t^{(1)}), s(\mathbf{x}_t^{(2)}) \dots s(\mathbf{x}_t^{(n)})$. The distribution is then updated to maximize the likelihood of observing the $m$ best samples by solving:

$$\tilde{\mathbf{v}}_t = \underset{\mathbf{v} \in \mathcal{V}}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} \mathbf{I}_{s(\mathbf{x}_t^{(i)}) > \gamma_t} \log f_{\mathbf{v}}(\mathbf{x}_t^{(i)}) \tag{2.5}$$

where $\mathbf{I}_{s(\mathbf{x}_t^{(i)}) > \gamma_t}$ is the indicator variable of the event $s(\mathbf{x}_t^{(i)}) > \gamma_t$, i.e., it takes the value of 1 if $s(\mathbf{x}_t^{(i)}) > \gamma_t$ and the value of 0 otherwise.

To prevent premature convergence to a local extremum, the changes of the distribution can also be smoothed by computing a weighted sum of the previous parameters and the solution of Equation 2.5 [dBKMR05]:

$$\mathbf{v_{t+1}} = \alpha \tilde{\mathbf{v}}_\mathbf{t} + (1 - \alpha)\mathbf{v_t} \tag{2.6}$$

where $\alpha$ is the smoothing factor.

In the particular case when $f_{\mathbf{v}}(\mathbf{x})$ is a multivariate Gaussian distribution, with mean and covariance given by $\mathbf{v}$, solving the Equation 2.5 amounts to setting $\tilde{\mathbf{v}}_t$ to the mean and covariance of the $m$ samples above the performance threshold $\gamma_t$.

The result of the optimization process after a number of iterations $T$ is the mean of the final distribution $f_{\mathbf{v_{T+1}}}$.

# Chapter 3

# Related Work

In this chapter, we present the landscape of scientific work related to the contributions of this thesis. We first examine the existing learning approaches to classical planning. We then step outside planning and discuss examples of machine learning applications in other combinatorial search problems, such as SAT-solving and automated theorem proving. We also provide an overview of work combining multiple search techniques, which is related to our work on learning the search policies. Finally, we discuss approaches in which elements of planning are used within reinforcement learning algorithms.

## 3.1 Learning in Classical Planning

Learning and planning have been combined in many ways and with many objectives. The foundational classical planning paper [FN71] was soon followed by a learning extension [FHN72], which learned macro operators, i.e., compositions of multiple operators, which can then be considered atomically, as additional operators. This allowed planners to automatically create, and switch between, different levels of abstraction. Since then the problem of learning macro operators has been considered by numerous works [DS77, Kor85, BEMS05, CS07]. The PbP planner [GSV09] is a portfolio planner which automatically constructs sets of macro operators for each of its component planners.

Learning has also been used to induce domain-specific rules to guide the planner's search [LZ98, YFG08, dlRCB08, dlRCFB11]. The rules control the search by suggesting the next operator or goal proposition to be considered by the planner. For example, a learned rule might state the preference for loading the package to the truck if they are both at the same location, which is not the package's destination. Subsequent work has also explored weighted combinations of multiple search control rules, where both the rules and their weights are learned [XFY10]. In an extreme case, the control rules can be used without search, as purely reactive policies indicating the next action to be taken (for example [Kha99, MG04, TTTX18]). However, in general, in the classical planning setting, the reactive policies have struggled to match the performance of search-based planners [YFG08]. Learning reactive policies is also close in spirit to the objective of generalized planning [CAJ19], which aims at computing plans which solve multiple problems representing the same domain. Learning search control rules is related to the task of learning search policies, which we introduce in Chapters 5 and 6. The key difference is that the control rules are learned at the level of specific planning operators, while our search policies are learned in the space of possible search algorithms.

A natural way of assisting planning with learned knowledge is to learn a heuristic function to guide best-first search of the planner [YFG06, YFG08, VBA13, GKLP16]. Such approaches typically base their feature representation on the computation of existing, hard-coded heuristic functions. When all the learning features are values of existing heuristics, the learning amounts to finding a weighted sum of multiple functions, which will most accurately predict the actual cost of reaching the goal from the evaluated state [VBA13]. Computing multiple heuristic functions, however, significantly increases the time required to evaluate a state. On the other hand, when learning is based on features extracted from the computation of a single heuristic, learning can be viewed as an attempt to improve on the original heuristic by using additional information extracted from its approximate plan [YFG06, YFG08, GKLP16]. In other related works, learning has been used to learn domain-specific ranking functions for node selection in beam search [XFY07, XFY09]. To the best of our knowledge, no work has attempted learning heuristic functions in a domain-independent setting before the developments presented in this thesis. Only very recently, a new approach was proposed, considering both domain-dependent

and domain-independent settings when learning from a hypergraphs representing a delete-relaxation of the problem. The learning mechanism is based on graph neural networks [STT20].

In addition to improving planner's performance, learning has also been used to refine incomplete planning models or even to acquire them from scratch, using either symbolic inputs (for example [YWJ07, ACO19]) or image representation of recorded traces [AF18, Asa19]. A detailed survey of learning approaches in classical planning has been conducted by Jimenez et al. [JDFB12].

## 3.2 Learning in Search

Machine learning has also been used to support search in tasks other than classical planning. Tasks such as theorem proving or deciding the satisiability of a given formula are closely related to planning. The common challenge is the search for sequence of steps that will lead to a state satisfying a certain property. One prominent example is learning to select the next formula transformation in a *Satisfiability Modulo Theory* (SMT) solver [BBV18]. Other work considered training neural networks directly to predict the satisfiability of a boolean formula [SLB$^+$19]. Learning has also been used to indicate the next variable to be assigned in a *Quantified Boolean Formula* (QBF) [LRS18] or to guide the search in automated theorem proving [KUMO18].

Machine learning has also been applied to a number of combinatorial optimization problems, such as the *Travelling Salesman Problem* (TSP) [VFJ15] or *Mixed Integer Linear Programming* (MILP) [BLBMT18]. For a survey of machine learning approaches to combinatorial optimization see [BLP18].

## 3.3 Planning with Multiple Techniques

By combining multiple search approaches, our work on learning the search policies is also related to existing literature on planning with multiple algorithms. Such approaches typically perform a number of independent search runs, each using a different algorithm. This idea underlies the

original Fast Forward planner [HN01], which first attempts to solve the problem using the very greedy but incomplete approach of *enforced hill-climbing*. If this fails, the planner switches to complete best-first search. The LAMA planner [RW10] aims to make the best use of available time limit by performing multiple runs of weighted A* with decreasing weights. The idea is to quickly find a potentially costly solution using greedy search and then try to improve on it by restarting the search with a more conservative approach.

The idea of performing a number of independent search runs is taken even further in *portfolio* planners (for example [HRK11, CDF16, SKS+19]), which divide the available time among a number different, potentially unrelated, algorithms. When many planners are involved in a portfolio, the question of choosing among them becomes a challenging problem itself. *Fast Downward Stone Soup* uses a simple hill-climbing approach to portfolio configuration: given a set of training instances a portfolio is constructed iteratively by considering possible additions and evaluating the resulting set of planners. The IBaCoP (*Instance-based Configured Portfolio*) [CDF16] takes a more advanced approach based on learning of predictive models, which are used to reconfigure the portfolio for each encountered problem. The learned models predict whether a given problem will be solved by a given planner and estimate the planner's running time. This information is used to select a subset of the available planners and to assign them individual time limits. A similar approach is used by *Delfi* [SKS+19], which uses a performance predictor trained on an image representation of the planning problem structure to select one of the component planners.

The multi-algorithm approaches discussed so far rely on division of the available time among multiple independent planners. While such approaches have demonstrated very strong performance in planning competitions, their performance is necessarily bounded by the performance of their components. To advance the state of the art, new search approaches and heuristics are necessary. The exisiting work on combining multiple search techniques within a single search run has been limited. Xie et al. [XMH14] have augmented best-first search with an exploratory technique, such as local search and random walks. Their search algorithm defaults to greedy best-first search and switches to the exploration when the search fails to progress for a given number of expansions (the progress is measured by changes in the lowest known heuristic value).

A similar approach is taken by Lipovetzky and Geffner [LG17], who use *width-based* search as the exploratory technique (other contributions of that paper include using the novelty measure, along with the heuristic, to order the open list, and a planner performing two independent runs, with the first one aggressively pruning states of low novelty). The algorithms using a single exploration technique are naturally limited to just two search approaches and a manually specified criterion for switching between them. The work presented in Chapters 6 and 5 adopts a different approach, in which many search techniques are available to the planner and the problem of choosing between them, given the state of the search, is handled using a trainable search policy.

## 3.4 Model-Based Reinforcement Learning

So far, the work on planning and learning discussed in this chapter focused on applications of machine learning techniques to planning problems. On the other hand, elements of planning have been a part of many reinforcement learning approaches. Adding some form of model-based planning to a reinforcement learning system can improve its sample efficiency, by allowing learning from 'imagined' traces in addition to the observed ones. Furthermore, a learned model can be used to perform a look ahead, helping to inform the choice of action by considering future states and rewards. One of the first examples of model-based reinforcement learning is the *Dyna* architecture [Sut90, SSGB08], which uses the agent's interaction with the environment to learn both the policy and a model of the environment. The model predictions are used to further improve the policy, without additional interactions with the environment. More recently, model-based approaches have been integrated with deep reinforcement learning (for example [OGL+15, SvHH+17, PLV+17, GWA+18]). Monte-Carlo tree search using a model encoding the rules of the game has been at the core of AlphaGo [SHM+16] and AlphaZero [SSS+17] systems, which achieved superhuman performance in the games of go, chess and shogi. In recent work, similar performance was achieved using a learned model instead of a manually specified one [SAH+19]. Tree-search using a model optimized directly for prediction of rewards and value functions allowed the approach to achieve state-of-the-art results on the Atari suite, in which

model-based approaches previously struggled to match the performance of the best model-free learners.

## 3.5 Summary

In this chapter, we presented an overview of existing literature at the intersection of planning and learning, with a particular focus on learning as a mean of improving the performance of classical planners. We also discussed existing approaches combining multiple search techniques in a single planner, which are related to our work presented in Chapters 5 and 6. In the remainder of this thesis, we will revisit the approaches most relevant to our work, and discuss them in more detail. For a systematic survey of learning approaches in planning we refer the reader to [JDFB12].

# Chapter 4

# Learning Heuristic Functions

The heuristic functions used in classical planning are typically hand-crafted algorithms. Existing approaches to learning heuristic functions are limited to domain-dependent settings: the heuristics are learned for a particular planning domain [YFG08, GKLP16]. In this chapter, we consider the problem of learning domain-independent heuristics. We aim to investigate whether useful heuristics can be learned without tying them to a particular domain and thus losing the generality of the planner. In our approach, the heuristics can be trained on data representing multiple domains and then deployed on any domain, including unseen ones.

The learning setting takes the supervised learning approach. A number of training problems are solved using existing planners. Each of the environment states lying along the computed plans becomes a training example. The machine learning model trained on this data is then used to provide the heuristic evaluation of unseen states, encountered while solving new planning problems.

In line with existing work on learning domain-dependent heuristics, we use state features extracted from computation of existing heuristic functions. Our experimental evaluation shows that in many cases the heuristics learned in a domain-independent setting offer a performance improvement over the original heuristic. Some of the ideas presented in this chapter, as well as the results of preliminary experiments are the subject of our paper presented at the *Workshop on Architectures for Generality and Autonomy* at IJCAI 2017 [GARB17].

The remainder of the chapter is structured as follows. In Section 4.1, we provide the motivation for our domain-independent approach. Section 4.2 discusses the related work on learning heuristic functions. In Section 4.3, we introduce a domain-independent representation of the environment states. Section 4.4 details the learning, while Section 4.5 describes our training data. We empirically evaluate the performance of the learned heuristics in Section 4.6 and summarize our findings in Section 4.7.

## 4.1 Motivation

The motivation for learning *domain-independent* heuristics is twofold. First, one of the key features of classical planning is the ability of the planner to handle problems representing multiple domains, including ones not seen previously. This is what motivates the use of domain-independent languages, formalisms and search algorithms in planning. Naturally, the research in classical planning relies on the assumption that techniques which work well on a number of known domains are also likely to be useful in other domains, even if they are not yet known. The planners' performance on unseen domains is tested, among others, by the *International Planning Competition*, which adds new domains in every edition. By learning a *domain-independent* heuristic we preserve the domain-independence of the planner. Rather than using learning to acquire domain-specific knowledge, we explore the possibility of using it to advance domain-independent planning.

Our interest in learning domain-independent heuristics is also inspired by the study of real-world systems in which a planner is used as a low-level problem solving mechanism [BSM+16, BMPR17]. Such systems are typically dynamic and open-ended. As a result, the domains faced by the planners are subject to change, for example when a new mode of transport or payment method is added to the system. Planning in such a setting requires domain independence, including the ability to handle unseen domains. On the other hand, dynamically created domains resulting from addition or removal of certain components are likely to exhibit similar properties, which makes learning from the planner's experience a promising approach.

## 4.2 Related Work

Learning heuristics is a natural way to learn from planners' experience: a learned heuristic function can be easily combined with a standard search algorithm, such as best-first search. However, the existing work on learning heuristics has been restricted to domain-dependent settings, in which the learned heuristic function is specific to one planning domain [YFG06, YFG08, PZ08, AZH11, VBA13, GKLP16]. The work by Yoon et al. [YFG06, YFG08] uses features extracted from the computation of a relaxed plan heuristic to learn a linear model. A similar approach is taken by Garret at al. [GKLP16], who use approximate plan features to train a Ranking SVM model. The ranking approach is motivated by the fact the heuristic value is only used to discriminate between states encountered while solving a given problem. This objective is naturally encoded in a ranking model. The training examples become pairs of states representing the same state space. This is in contrast with a regression for the distance to the goal, in which case states from multiple plans are treated jointly. Empirical results confirm that the ranking approach compares favourably to a regression. The domain-dependent *single* and *pairwise* action features [GKLP16], which we discuss further in Section 4.3, are the basis of many of the domain-independent features introduced in this chapter.

## 4.3 Feature Representation

Providing a feature representation is a key challenge in learning a heuristic function. Since the work presented in this chapter is focused on learning domain-independent heuristics, the feature representation cannot depend on particular planning domain, for example by referring to particular object or action types. In its purest form, the learning process should use simple features, directly extracted from the description of the planning problem and the current environment state. However, learning from such simple features has proven very hard. In most cases, the heuristics trained on top of such features perform much worse than state-of-the-art domain-independent planning heuristics [GARB17]. A common approach in recent works on learning domain-dependent heuristics [YFG08, GKLP16] is to build a feature representation

based on the approximate plans constructed by one of the existing domain-independent heuristics. It can be argued that deriving the feature representation from the computation of an existing heuristic effectively changes the learning task to learning a *correction* of the original heuristic. In this work, we follow a similar approach, adapted to a domain-independent setting. The meaning of our features is therefore never tied to a particular planning domain.

In line with previous work on learning domain-dependent heuristics [GKLP16], we consider features extracted from the computation of *Fast Forward* (FF) and *context-enhanced additive* (CEA) heuristics. The CEA heuristic [HG08] is based on ideas similar to the *additive* heuristic [BLG97, BG01], which uses a relaxed version of the problem to construct relaxed plans for each of the goal propositions, and then uses a sum of the plans' costs as the heuristic value. In a nutshell, CEA differs from plain additive heuristic by retaining more information about the effects of the original operators. Rather than ignoring them as a part of the relaxation process, it uses some of them to create *contexts*, in which subseqent propositions are evaluated. The common property of FF and CEA is the construction of an approximate (partially-ordered) plan, whose cost is used as the heuristic value for the evaluated state. Extracting features form the plan allows for retaining more information than just the plan's cost. Concretely, we work with three groups of features. The first is extracted directly from the plan. The two other groups are domain-independent *single* and *pairwise action* features, inspired respectively by the domain-dependent *single* and *pairwise action* features used in existing work on learning domain-dependent heuristics [GKLP16].

**Approximate plan features** The features extracted directly from the approximate plan include the value of the heuristic itself (the cost of the approximate plan) as well as additional information related to the length of the plan and the amount of relaxation used in the process of constructing it. This group of features includes:

- The value of the original heuristic (FF or CEA).
- The total number of operators used in the approximate plan.

- The total number of *ignored* effects of operators in the approximate plan: under the MPT formalism, the approximate plans are computed using *unary* operators. The unary operators are derived from the original ones by keeping only one of their effects. The number of *ignored* effects is computed as $n_{ign} = \sum_{o \in P}(|\operatorname{eff}(o)| - 1)$, where $\operatorname{eff}(o)$ is the set of effects of the original operator for a unary operator $o$ included in the approximate plan $P$.

- The average number of *ignored* effects in the approximate plan (ratio of the two numbers above).

- The number of layers in the approximate plan's graph.

**Single action features** The *single action features* considered by the previous work [GKLP16] are the numbers of occurrences of each of the operator schemata in the approximate plan. This information is inherently domain-dependent, as the operator schemata (as well as their number) vary between domains. In this chapter, we consider a set of domain-independent single action features, which we define as quantiles (specifically, the three quartiles) of the domain-dependent *single action* features distribution. The first step for obtaining domain-independent features is to compute the domain-dependent ones, i.e., the numbers of occurrences of each operator schema. The three domain-independent features represent the quartiles of this set of domain-dependent features, i.e., numbers $q_1, q_2, q_3$, such that a quarter, half, and three quarters, respectively, of the action schemata occurred no more times than $q_1$, $q_2$ and $q_3$.

**Pairwise action features** The *pairwise action features* from the previous work encode relations between the operators included in the approximate plan. Specifically, they are Boolean matrices of size $n_{os} \times n_{os}$, where $n_{os}$ is the number of operator schemata in the domain. The element at the intersection of row and column representing two schemata is *true* if, in the approximate plan, an instance of one schema precedes an instance of the other schema. Otherwise, the element is *false*. Similarly, the second matrix encodes situations when an instance of one schema is used in the plan to satisfy a precondition of an instance of the other schema.

In our domain-independent approach, we first construct the domain-dependent features and then build two domain-independent features on top of them. Specifically, we first construct matrix $F^{pair}_{n_{os} \times n_{os}}$ which is an elementwise conjunction of the two matrices above, i.e., an element is 1 if one of the schemata precedes the other *and* its effect is the other schema's precondition. Otherwise, the element is 0. Intuitively, a value of 1 indicates that an instance of one schema likely depends on an instance of the other to achieve one of its preconditions. Our domain-independent features are the number of *true* elements in $F^{pair}$ and the number of symmetric dependencies, i.e., cases when $F^{pair}_{ij} \wedge F^{pair}_{ji}$ holds.

In addition to the features extracted from the approximate plans, our feature set includes the number of goal propositions not satisfied in the evaluated state (i.e., the value of the *goal count* heuristic).

## 4.4 Learning Algorithm

In line with recent work on learning domain-dependent heuristics [GKLP16], we adopt a Ranking SVM [Joa02] as the learning mechanism. Given a number of training plans ($n$), the learning examples are all the pairs of states occurring in the same plan. We denote the feature representation of a state $s^i_j$ occurring at the $j$-th position in the $i$-th training plan as $\phi(s^i_j)$. The SVM optimization problem takes the following form:

$$\min_{w} \quad ||w||^2 + C \sum_{i=1}^{n} \sum_{j=1}^{m_i} \sum_{k=j+1}^{m_i} \xi_{ijk}$$

$$\text{subject to:} \quad \phi(s^i_j)^\top w \geq \phi(s^i_k)^\top w + 1 - \xi_{ijk} \quad \forall i; \quad \forall j,k : j < k$$

$$\xi_{ijk} \geq 0 \quad \forall i,j,k$$

where $w$ is the vector of trainable weights, $C$ is the regularization parameter, $n$ is the number of training plans, $m_i$ is the number of states along the $i$-th training plan and $\xi_{ijk}$ is the slack variable for $j$-th and $k$-th states of plan $i$. The leftmost sum is over the training problems, the latter two iterate through all pairs of states, just like the standard RSVM formulation in Chapter

2. Throughout the rest of the chapter, we use the default value of the regularization parameter $C = 1$. Preliminary experiments with different values of $C$ have shown no improvement over this configuration.

## 4.5 Training Data

In this section, we describe the process of collecting the data necessary for learning the heuristic functions. As hinted before, we only use optimal plans in training. This ensures that there is no noise in the training data, i.e., all states along the plan trajectories are ordered by their factual distance to the goal. Naturally, this puts a practical constraint on the maximum size of the planning problems used for training: finding provably optimal solutions in reasonable time is only feasible for relatively small planning problems. We made the decision to restrict the training setting to optimal plans after preliminary experiments with data that also included suboptimal solutions to larger problems, which turned out not to improve the performance of the learned functions.

For the experimental evaluation presented in this chapter, we use the four IPC domains of *Transport*, *Parking*, *Elevators* and *No-mystery*. The same set of four domains has also been used by previous work on learning domain-dependent heuristics [GKLP16]. Compared to the set of domains we use in later chapters for evaluating learned search policies, we exclude the domain of *Floortile*. The main reason for this is that, with our approach learning from optimal solutions, gathering enough data points would not be feasible (the *Floortile* problems small enough to be solved optimally are too similar to one another to gather a number of unique samples comparable with what is feasible for the other domains).

The training problems were generated using the problem generators provided by the organizers of the learning track of IPC 2014[1]. Below, we list the values of the parameters passed to the generators. Overall, the data set consists of solutions to randomly generated problems representing the following problem distributions:

[1] `http://www.cs.colostate.edu/~ipc2014/`

- three distributions of *Transport* problems generated using parameters corresponding to Problems 2, 12 and 22 of the demonstration problem set of IPC 2014: the maps contained between 4 and 10 locations, each connected to two other locations, 2 trucks and 4 packages to deliver;
- four distributions of *Parking* problems: 4 curbs with 6 cars, 5 curbs with 7 cars, 5 curbs with 8 cars and 6 curbs with 8 cars;
- *Elevators* problems with 12 floors, 3 passengers, area factor of 6, 1 fast elevator and two slow ones, all of capacity 4;
- *No-mystery* problems with 6 locations, 6 packages and a quality constraint of 1.1;

For each of the domains, we aimed for a dataset containing about 20000 examples. This amounted to using solutions of 1000 *Transport*, 1803 *Parking*, 1002 *Elevators* and 1005 *No-mystery* problems.

## 4.6 Evaluation

We evaluate the learned heuristic functions in two settings. The first one contains relatively small instances of all the four domains. In the case of *Transport*, we use the demonstration set of 30 problems published by the organisers of the IPC 2014 learning track. To match the *Transport* setting, for the remaining domains, we use randomly generated sets of 30 problems of increasing size, roughly matching the difficulty of the *Transport* set, in the sense that a similar proportion of the problems can be solved by the baseline planner using GBFS with either FF or CEA heuristic, with a time limit of 1 minute. The second setting uses larger problems, sourced from the satisficing track of IPC 2011 and a time limit of 3 minutes (this matches the setting we use for evaluating the search policies learned in Chapters 5 and 6).

To evaluate the learned domain-independent heuristics, we perform cross-validation across the four domains. For each of the test sets, a heuristic function is learned on the three remaining

| | Elevators | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|
| FF | 2.46 | 8.48 | 18.71 | 8.48 | 38.13 |
| Learned (FF) | 14.2 | 12.73 | 21.29 | 8.46 | 56.68 |
| Learned (FF, DS) | 3.14 | 11.73 | 19.61 | 11.48 | 45.96 |
| Learned (FF, DD) | 2.7 | 12.81 | 21.39 | 12.62 | 49.52 |
| FF (unary costs) | 26.49 | 8.48 | 18.71 | 13.91 | 67.59 |

Table 4.1: The IPC scores obtained by GBFS with different FF-derived heuristics on small problems (time limit of 1 minute).

domains, excluding data from the domain in question. For comparison, we also evaluate the heuristic functions learned in two domain specific settings. The first one uses the same domain-independent features described above. However, the training involves only data representing the target domain. For example, the heuristic for the *Elevators* domain is trained using *Elevators* data, as opposed to combined data representing the other three domains. In Tables 4.1 – 4.4, this setting is denoted as *DS* for *Domain (data) Set*. The second reference setting is a completely domain-dependent approach that uses target domain training data, as well as our implementation of the domain-dependent features of Garrett et al. [GKLP16] instead of our domain-independent features. This setting is denoted as *DD* for *Domain-Dependent*.

The IPC scores obtained on the small test problems by the heuristics learned with FF features are reported in Table 4.1. The learned domain-independent correction of FF achieved higher score than the base heuristic in three out of the four domains and practically the same score in *Transport*. The comparison between the domain-independent and analogous domain-dependent setting suggests that learning from data representing other domains rather than from the target one does not necessarily hurt the planner performance (domain-independent is better in *Elevators*, worse in *Transport*, the is similar in *No-mystery* and *Parking*). An important caveat is that we evaluate the domain-dependent learner in a setting which was primarily designed for learning domain-independent functions. This influenced not only the feature representation, but also other design choices, for example the decision to learn from optimal solutions to small problems, rather than suboptimal plans for larger instances. With small problems inherently exhibiting limited variability, mixing data from multiple domains may have a regularizing influence on the learner, thus putting the domain-dependent variant at a disadvantage.

We initially only compared the learned heuristics to the original heuristic. As evidenced by the upper section of Table 4.1, the comparison looked favourably for the learned domain-independent functions, which generally outperformed the original heuristic. However, a closer examination of the learned heuristics revealed that in many cases their behaviour amounted to discarding the operator costs (the dominating feature was the number of operators in the approximate plan). After making this observation, we decided to add a comparison against the original heuristics computed in unary-cost setting (i.e. with the cost of all operators set to 1).

It turned out that search guided by the unary-cost heuristic outperformed our learned functions in two out of the four domains. In particular, in the *Elevators* it was able to solve 28 out of 30 problems (as opposed to 19 problems solved by the learned heuristic and just 3 solved by the original heuristic). We hypothesize that the reason behind the poor performance of the cost-aware heuristics in this domain is the presence of zero cost operators, representing passengers boarding and leaving an elevator. Such operators, although they are necessary to solve the problem, and often present in the approximate plans, do not increase the value of the original heuristic, which consequently fails to distinguish between many of the environment states. Consider for example, a situation where a number of passengers need to board an elevator on a given floor. The state before the first passenger enters the elevator has the same heuristic value as the one after all passengers have boarded, even if the latter one is significantly closer to the goal. By using features other than the value of the original heuristic, our learned correction was able to recover some of the distance between the original and the unary-cost heuristic.

Importantly, the learned heuristic correction achieved scores higher than both the original and the unary-cost heuristic in *No-mystery* and *Parking* domains, which do not assign operator costs. In this domains, the base and unary-cost versions of the heuristic coincide, resulting in identical scores. This shows that the learner can improve also on the unary-cost version of the heuristic. One possible direction of future work is to only use the unary-cost version of the heuristic to compute the state features, even in the domains where operators have costs, like *Elevators* and *No-mystery*.

In Table 4.2, we report the IPC scores obtained on the IPC 2011 problems by the functions

| | Elevators | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|
| FF | 0 | 8.85 | 8.65 | 0 | 17.5 |
| Learned (FF) | 5.12 | 4.94 | 0.76 | 0 | 10.82 |
| Learned (FF, DS) | 0 | 4.94 | 0.79 | 0 | 5.73 |
| Learned (FF, DD) | 0 | 6.94 | 9.63 | 0 | 16.57 |
| FF (unary costs) | 17 | 8.85 | 8.65 | 0 | 34.5 |

Table 4.2: The IPC scores obtained by GBFS with different FF-derived heuristics on IPC-2011 problems (time limit of 3 minutes).

| | Elevators | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|
| CEA | 12.11 | 4.97 | 19.53 | 17.9 | 54.51 |
| Learned (CEA) | 12.72 | 5.36 | 13.31 | 14.53 | 45.92 |
| Learned (CEA, DS) | 15.72 | 5.15 | 15.25 | 15.66 | 51.78 |
| Learned (CEA, DD) | 14.94 | 5.15 | 11.39 | 17.54 | 49.02 |
| CEA (unary costs) | 26.97 | 4.97 | 19.53 | 21.19 | 72.66 |

Table 4.3: The IPC scores obtained by GBFS with different CEA-derived heuristics on small problems (time limit of 1 minute).

learned using FF features. These problems are even further in size from the optimally solvable training problems than the first evaluation set. As a consequence, the relative performance of the learned functions is worse and in most cases fails to match the original heuristic. The exception is the *Elevators* domain, where they are still able to recover some of the distance between the base and unary version of the original heuristic. These results indicate that there is a limit to how far outside the training distribution the learned functions can generalize. This suggests that for larger problems training using suboptimal solutions to larger problems may indeed be necessary.

We originally developed our learning approach using features extracted from the FF heuristic. However, since the features are extracted from the approximate plans, the approach can also be used with any other heuristic constructing a plan approximation in the process of evaluating a state [GKLP16]. Table 4.3 presents the IPC scores obtained by heuristics learned using features extracted from the computation of the *Context-Enhanced Additive* heuristic (CEA) [HG08].

The learned heuristic functions achieve scores higher than the original heuristic in *Elevators* and *No-mystery* domains and lower in *Parking* and *Transport* domains. As was the case with FF, the unary-cost version of CEA generally outperforms other heuristics. Among the two domains without operator costs, the heuristic learned using CEA features outperforms the unary-cost

| | Elevators | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|
| CEA | 1.37 | 5.77 | 3.69 | 5.48 | 16.31 |
| Learned (CEA) | 2.22 | 5.78 | 0 | 3.86 | 11.86 |
| Learned (CEA, DS) | 1.72 | 5.78 | 1.41 | 4.28 | 13.19 |
| Learned (CEA, DD) | 1.34 | 5.78 | 0 | 3.91 | 11.03 |
| CEA (unary costs) | 7.05 | 5.77 | 3.69 | 5.94 | 22.45 |

Table 4.4: The IPC scores obtained by GBFS with different CEA-derived heuristics on IPC-2011 problems (time limit of 3 minutes).

CEA in *No-mystery*, but falls behind in *Parking*.

The scores obtained by CEA-based heuristics on IPC 2011 problems are reported in Table 4.4. Consistently with the FF case, the relative performance of the learned heuristics seems to degrade on larger problems, which are more distant from the training instances. The learned functions lose to both base and unary-cost CEA, with the exception of *No-mystery*, where their scores are minimally higher.

## 4.7 Summary

In this chapter, we have adapted the idea of learning (corrections of) heuristic functions to a domain-independent setting. We have introduced a domain-independent representation of environment states, based on the computation of approximate-plan heuristics and domain-dependent features introduced in previous works. In the experimental evaluation, we have shown that in some cases the learned functions outperform the original heuristics on domains not seen in the training process. The comparison against two domain-dependent approaches suggests that learning from domains other than the target one is possible and often leads to similar performance of the learned heuristic.

On the other hand, the behaviour of the learned heuristics in domains with operator costs falls behind the unary-cost version of the base heuristic, only recovering a part of the difference between the base and unary-cost versions of the original heuristic. The fact that the learned heuristic functions often slot between the original heuristic and its unary cost version suggests that the behaviour of the learned functions remains close to the original. Furthermore, the

performance of the learned heuristics degrades significantly when testing on larger problems, like, for example, full-scale IPC problems.

# Chapter 5

# Learning Search Policies with a Finite Action Space

In this chapter, we introduce the task of learning search policies. The policies are responsible for choosing the most appropriate way of proceeding with the search given its current state. We first construct a search algorithm capable of alternating between a number of search routines while solving a planning problem. We then define the task of learning to choose a routine given a performance objective, a particular distribution of planning problems and the current state of the search. The proposed approach is based on reinforcement learning with a discrete action space comprising alternative search routines. At every step, a routine is chosen depending on the state of the search. The choices are rewarded using the planner's overall performance with respect to the selected performance objective. Training on a given distribution of planning problem enables the discovery of distribution-specific search policies that maximize the planner's performance with respect to the given objective.

The approach presented in this chapter is focused on learning *stochastic* policies. With our high-level representation of the search state, detailed in Section 5.4, the planner typically takes a number of actions from any given state. In this setting, stochasticity of the search policy allows it to explore combinations of multiple routines created by interleaving between them.

The ideas and results presented in this chapter have been published in a paper presented at

the *International Conference on Automated Planning and Scheduling* (ICAPS) 2019 [GAR19]. The chapter is structured as follows. Section 5.1 relates our approach to existing work in the field of classical planning. Our search algorithm alternating between multiple search routines is introduced in Section 5.2. The general learning framework is detailed in Section 5.3. Section 5.4 presents a possible instantiation of the framework, which is then empirically evaluated in Section 5.5.

## 5.1 Related Work

While learning search policies is itself a novel problem, our approach is related to some of the existing work combining multiple search techniques in a single search run. A prominent example of this is the work by Xie et al. [XMH14], where GBFS is supported with auxiliary exploration mechanism whenever the search fails to make progress for a certain number of node expansions. To determine progress, the algorithm keeps track of the lowest heuristic value encountered since the search began. If the value has not changed for a given number of expansions, the algorithm switches to the exploration mechanism. Two versions of the algorithm are considered: one exploring using random walks and one with local search. A similar idea underlies BFS with Width-Based Exploration [LG17], where the exploration mechanism is width-based search [LG12].

Unlike our work, these approaches rely on manually specifying the primary (GBFS) and auxiliary search technique (random walks, local search or width-based search), as well as the criterion for switching between the techniques. In our work, multiple search techniques are always available to the planner, without manually distinguishing between the primary and auxiliary ones. The task of choosing among them, given the state of the search, is subject to a search policy trained using a reinforcement learning algorithm.

## 5.2 Planning with Alternating Search Routines

As a first step towards a planner that learns search policies, we introduce a planner capable of alternating between multiple search routines. The general idea behind planning with alternating search routines (ASR) is to encapsulate the operations determining the nature of the search in a set of alternative routines. The routines are designed so that each call of a routine applies an atomic change to the state of the search, e.g., a single node expansion. When the learning approach is introduced in Section 5.3, the routines will correspond to actions in the reinforcement learning framework.

Algorithm 2 outlines a planning approach that alternates between various search routines in the process of solving a planning problem. In addition to the initial state $s_0$, goal $g$ and set of operators $\mathcal{O}$, the function takes as an argument a set of routines $A$. The algorithm chooses one of the routines (line 4) and applies it to the current search frontier (*open_list*) for a number of steps (lines 5-10), after which another routine can be chosen. If application of the routine results in finding a plan or exhausting the search space, the result is returned from the planner (lines 7-9).

An example of a search routine intended for use with Algorithm 2 is shown in Algorithm 3. The arguments of the function are the current search frontier (*open_list*) and the set of planning operators $\mathcal{O}$. The routine removes a single search node from the open list (line 5) and expands it using all applicable operators (lines 6-14). The successor states are added to the open list (line 12). If the node expansion is completed the routine returns the *in-progress* token. Otherwise, it returns *failure* if the open list turned out empty (line 3), or the plan if a goal state has been found (line 10). The routine can represent breadth-first, depth-first, greedy or $\epsilon$-greedy search depending on the implementation of the *next* and *insert* functions. It is also possible to come up with other search routines, not necessarily fitting the template of Algorithm 3. For example, a search routine could follow the search node expansion with a random walk starting at the expanded state.

**Algorithm 2** Planning with alternating search routines

```
1: function PlanASR(s_0, g, O, A)
2:     open_list ← [s_0]
3:     while true do
4:         routine ← choose(A)
5:         while some condition do
6:             result ← routine(open_list, O)        ▷ open_list is processed with routine
7:             if result ≠ in-progress then
8:                 return result                     ▷ The result is a plan or failure
9:             end if
10:        end while
11:    end while
12: end function
```

**Algorithm 3** A generic search routine for ASR planning

```
1: function ASRroutine(open_list, O)
2:     if open_list is empty then
3:         return failure
4:     else
5:         s_c ← next(open_list)
6:         for all o ∈ O applicable in s_c do
7:             s ← o(s_c)
8:             if s ⊨ g then
9:                 plan ← extract-ancestors(s)
10:                return plan
11:            else
12:                insert(open_list, s)
13:            end if
14:        end for
15:        return in-progress
16:    end if
17: end function
```

## 5.3 The Learning Framework

Having introduced the concept of a planner alternating between multiple search routines, we can proceed to consider the notion of *search policies* that define a probability distribution over the available routines, given the current state of the search. In this section, we assume the existence of a set of possible states of the search $S$ given by possible assignments over a set of variables $\mathcal{V}_S$, and a set of search routines $A$. Sets $S$ and $A$ form the basis of a reinforcement learning task as defined in Chapter 2. A possible instantiation of $S$ and $A$ is presented in Section 5.4.

Reinforcement learning policies are typically modelled using a trainable set of parameters. As noted before, in this chapter we are primarily interested in learning stochastic policies. A common way to parametrize a policy mapping each state $s \in S$ to a probability distribution over actions in $A$ is to associate each state-action pair $\langle s_i, a_j \rangle$ with a weight $\theta_{i,j}$. The probability of taking action $a_i$ (i.e., choosing routine $a_i$) in state $s_j$ is then given by softmax over the weights associated with that state:

$$\pi(a_j|s_i) = \frac{e^{\theta_{i,j}}}{\sum_j e^{\theta_{i,j}}} \tag{5.1}$$

Algorithm 4 adapts Algorithm 2 to the learning setting. The changes include:

- an additional argument $\theta$ – the parameters of the policy, i.e., weights $\theta_{i,j}$ for all state-action pairs $\langle s_i, a_j \rangle$,
- computing features representing the planner's state (line 4),
- choosing the next routine according to policy $\theta$ and the state features (line 5).

The search policies are learned using a policy gradient approach [Wil92] presented in Section 2.3.1. The learning process is outlined by Algorithm 5. The main loop of the process (lines 3-10) consists of sampling a problem from the set of training problems (line 4), running the current planner $N$ times (lines 5-7), and updating the policy parameters according to observed performance of the planner (lines 8 and 9).

**Algorithm 4** Planning with a learned search strategy

**function** PLANASR($s_0, g, \mathcal{O}, A, t_r, \theta$)
  *queue* $\leftarrow [s_0]$
  **while** true **do**
    compute planner's state features $F$ ▷ e.g. $F = \langle far, early \rangle$
    *routine* $\leftarrow$ choose($A, F, \theta$) ▷ select a routine from $A$, given $F$ and $\theta$
    $t_{start} \leftarrow$ now() ▷ current time
    **while** now() $- t_{start} < t_r$ **do** ▷ keep applying the routine for time $t_r$
      *result* $\leftarrow$ *routine*(*queue*, $\mathcal{O}$)
      **if** *result* $\neq$ in-progress **then**
        return *result* ▷ terminate with a plan or with a failure
      **end if**
    **end while**
  **end while**
**end function**

**Algorithm 5** Learning search strategies

**function** LEARNSTRATEGY($P, A, S, N, t$)
  $\theta \leftarrow$ zeros($|A| \times |S|$) ▷ initialize all $|A| \times |S|$ weights to 0
  **while** elapsed time $< t$ **do**
    $p \leftarrow$ sample($P$) ▷ randomly choose one of the problems
    **for** $i = 1$ to $N$ **do**
      $G_i =$ score_planner($\theta, p$) ▷ run the planner on $p$, return the reward
    **end for**
    update $V(s)$ for each $s \in S$, by averaging $G$ for all past runs visiting $s$
    update $\theta$ according to Equation 5.2
  **end while**
  return $\theta$
**end function**

The parameter update follows the REINFORCE algorithm [Wil92]. The equation for updating the weights $\theta$ after running the planner $N$ times on problem $p$ takes form:

$$\Delta\theta = \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{L_i} (G_i - V(s^{i,j})) \frac{\nabla_\theta \pi(a^{i,j}|s^{i,j})}{\pi(a^{i,j}|s^{i,j})} \tag{5.2}$$

where $\alpha$ is the learning rate, $L_i$ is the length of episode $i$, $G_i$ is the score obtained by the planner for run $i$, $s^{i,j}$ is the planner's state at time step $j$ of run $i$ and $a^{i,j}$ is the action taken from that state.

In our specific setting, in which the total number of state-action pairs is relatively small (the instantiation of our framework introduced in the next section results in 20 state-action pairs), it is computationally convenient to reorder the internal sum in Equation 5.2 by state-action pairs (rather than time steps).

Introducing the term:

$$\zeta(i, s, a) = (G_i - V(s)) \frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} \tag{5.3}$$

we can rewrite Equation 5.2, reordering the internal sum by state-action pairs::

$$\Delta\theta = \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{L_i} (G_i - V(s^{i,j})) \frac{\nabla_\theta \pi(a^{i,j}|s^{i,j})}{\pi(a^{i,j}|s^{i,j})} = \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{L_i} \zeta(i, s^{i,j}, a^{i,j}) =$$

$$\alpha \frac{1}{N} \sum_{i=1}^{N} \left( \zeta(i, s^{i,1}, a^{i,1}) + \zeta(i, s^{i,2}, a^{i,2}) + \ldots + \zeta(i, s^{i,L_i}, a^{i,L_i}) \right) =$$

$$\alpha \frac{1}{N} \sum_{i=1}^{N} \left( \underbrace{\zeta(i, s_1, a_1) + \ldots + \zeta(i, s_1, a_1)}_{\eta_i(s_1,a_1) \text{ times}} + \zeta(i, s_1, a_2) + \ldots + \underbrace{\zeta(i, s_n, a_m) + \ldots + \zeta(i, s_n, a_m)}_{\eta_i(s_n,a_m) \text{ times}} \right) =$$

$$\alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{s,a} \eta_i(s, a) \zeta(i, s, a) =$$

$$\alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{s,a} \eta_i(s, a) (G_i - V(s)) \frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} \tag{5.4}$$

where $s_1, s_2 \ldots s_n$ are the environment states in set $S$; $a_1, a_2 \ldots a_m$ are the actions in set $A$; and $\eta_i(s, a)$ is the number of times action $a$ was applied from state $s$ in run $i$. The final form of the

parameter update equation:

$$\Delta\theta = \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{s,a} \eta_i(s,a)(G_i - V(s)) \frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} \tag{5.5}$$

is more convenient because each value $\zeta(i,s,a)$ only needs to be computed once and the repeated summation of equal terms $\eta_i(s,a)$ is replaced by multiplication.

In the particular case of a softmax policy with the parameter vector $\theta$ containing one weight for each state-action pair $(s_i, a_j)$, differentiation of the policy $\pi$ with respect to the parameters $\theta$ leads to the following expression for $\nabla_\theta \pi(a|s)$:

$$\frac{\partial \pi(a|s)}{\partial \theta_{i,j}} = \begin{cases} 0, & \text{if } s \neq s_i \\ 1 - \pi(a|s), & \text{if } s = s_i \text{ and } a = a_j \\ -\pi(a_j|s), & \text{if } s = s_i \text{ and } a \neq a_j \end{cases} \tag{5.6}$$

## 5.4 Framework Instantiation

To instantiate the framework described in Section 5.3, we need to specify the set of search routines $A$, the set of planner's states $S$ and the reward function. In this section, we present possible design choices for each of the components. An empirical evaluation of the resulting system is presented in Section 5.5.

### 5.4.1 Representation of the Planner's State

The state set $S$ can be specified by introducing a set of finite-domain variables $\mathcal{V}_S$ that describe the state of the planner. The set $S$ is then the set of possible assignments over the variables of $\mathcal{V}_S$. Here, we consider $\mathcal{V}_S = \{d, t\}$, a set of two high-level binary features of the planner's state.

Variable $d$ is a binary heuristic estimate of the distance to the goal. Formally:

$$d = \begin{cases} 0 & \text{if } h_{\text{best}} < h_0/2 \\ 1 & \text{otherwise} \end{cases}$$

where $h_{\text{best}}$ is the lowest value of the heuristic function encountered in the search so far and $h_0$ is the value of the heuristic for the initial state. The value of $d$ enables assessing the relative progress towards the goal made by the search so far. Naturally, the quality of the estimate depends on the quality of the heuristic.

Variable $t$ is a binary encoding of how much time the planner has left to solve the problem:

$$t = \begin{cases} 0 & \text{if } t_{\text{elapsed}} < t_{\text{max}}/2 \\ 1 & \text{otherwise} \end{cases}$$

where $t_{\text{elapsed}}$ is the time elapsed from the beginning of the search and $t_{max}$ is the total time allocated for the search.

Intuitively, these two variables capture essential information about the state of the search. For example, if $d = 0$ and $t = 0$, the situation is relatively promising, since substantial progress towards the goal has been made early. If, on the contrary, $d = 1$ and $t = 1$, the search frontier is still far from the goal, while time is running out. This may suggest that a change in the search approach is necessary, for example by increasing the amount of exploratory behaviour, even if it can negatively affect the quality of the plan.

In the remainder of this chapter, we refer to the states of the search described by $d$ and $t$ as *near* ($d = 0$) or *far* ($d = 1$) and *early* ($t = 0$) or *late* ($t = 1$). Using a state representation comprising two binary variables allows us to use a simple tabular variant of policy gradient, explicitly storing one weight parameter for each state-action pair.

### 5.4.2 Search Routines

The action set $A$ includes the search routines available to the learning planner. Below, we describe the routines included in our instantiation of the framework. The routines include plain and randomized best-first search, random walks, local search and depth-first search. This is a varied set of planning techniques which can fit in the template of Algorithm 4.

**Greedy Best-First Search** The first routine is plain greedy best-first search (GBFS) which always expands a node with the lowest value of the heuristic function. A single step of this routine extracts a single node of the open list, generates all of its successor nodes by applying each of the applicable operators, and adds the successor nodes to the open list.

**$\epsilon$-Greedy Best-First Search** Randomization of the order of node expansion is one of the common modifications of greedy search [IK11, VSSX14, XMHI14, AF17]. A simple but effective technique is $\epsilon$-greedy search, first applied in the context of classical planning setting by Valenzano et al. [VSSX14]. It differs from plain GBFS by selecting a random node from the open list with probability of $\epsilon$ and a node with the lowest value of the heuristic function with probability of $1 - \epsilon$. If a random node is to be selected, the probability of selection is uniform across all the nodes in the open list. Throughout this chapter a fixed value of $\epsilon = 0.2$ is used.

**Greedy Best-First Search with Random Walks** This method is inspired by Xie et al. [XMH14]. A single step of the routine removes one node and expands it in a way analogous to plain GBFS. Performing a full expansion of a node removed from the open list preserves the completeness of a planner using this routine. If no decrease in the lowest known heuristic value has been observed for the last $s$ expansions, the expansion is followed by a single random walk of length up to $l$, starting at the recently expanded node. The walk stops immediately if a state with new lowest heuristic value is encountered. All the nodes encountered during the random walks are added to the open list. In this chapter we use the parameters of $s = 5$ and $l = 20$.

**Local Search** When local search routine is selected, a single node with lowest value of the heuristic function is extracted from the open list and used to start a new (local) open list. The local open list persists between the steps of local search, each of which consists of extracting a single node, expanding it and adding its successors to the open list. When the action time limit $t_r$ is reached, the local queue is merged into the global one by inserting all the nodes from the local list into the global one. This preserves the completeness of the planner. If the local list is emptied before time limit $t_r$ and the global list still contains unexpanded nodes, another node is moved from the global list to the local one. This effectively starts new local search from the recently moved node.

**Heuristic-Guided Depth-First Search** Similar to local search, depth-first search (DFS) is performed using a local data structure. When the routine is selected, a single node with the lowest value of the heuristic function is removed from the open list and placed on a stack. At every step, a single node is popped from the stack and expanded. The descendants are placed on the stack in the order of decreasing heuristic value, so that the next node expanded is the most promising descendant of the current node. All the descendant nodes are also added to the global open list. If the stack is emptied before time limit $t_r$, another node from the open list is moved to the stack and the search continues.

### 5.4.3 Reward Computation

The trade-off between the cost of the plan and the time required to find a plan (or the chance to find a plan within a given time limit) is a central problem in satisficing classical planning. One of the key advantages of learning search policies is the possibility of using the learning process to directly optimize for any performance objective that can be estimated during training. This includes performance measures commonly used in the planning community, such as problem coverage or the IPC score, but may also be extended to other ways of assessing planner performance, including application specific ones. For example, training of a real-world application of the system could involve feedback based on end-user rating of the plans.

In this chapter, we consider rewards based on three different performance objectives. In addition to the traditional IPC score, they include the squared IPC score, which increases emphasis on plan quality, and the time score, which is only concerned with how fast a plan is found.

**IPC score** The IPC score, first used in IPC 2008[1], is a common way of measuring the performance of a satisficing planner. Given that a problem has been solved, higher scores are awarded for plans of lower cost. For every solved problem, the planner receives between 0 and 1 point:

$$z = \frac{c_{min}}{c}$$

where $c$ is the cost of the plan returned by the planner and $c_{min}$ is the lowest known cost of a plan solving the problem. Typically, the value of $c_{min}$ is designated by taking the minimum over all the evaluated planners.

To provide a reward for training our system, we modify to the traditional IPC score in two ways. First, we exclude the cost returned by the trained planner from the minimum operation, allowing for a situation, where $c < c_{min}$ and $z > 1$. This way, we can differentiate between situations when the returned plan is as good as the best known so far ($z = 1$) and cases when it is strictly better ($z > 1$). In cases when the trained planner solves a problem for which no solution was known, and so $c_{min}$ is not defined, the reward is set to a fixed value of 2. Intuitively, the value of 2 models the advantage the trained planner can get by solving a problem not solved by its competitors: it is 1 point more relative to the situation when the trained planner and a competitor solve the problem with the same plan cost, in which case the reward is 1. Formally,

$$r_{ipc} = \begin{cases} \frac{c_{min}}{c_L} & \text{if both } c_{min} \text{ and } c_L \text{ are defined} \\ 0 & \text{if only } c_{min} \text{ is defined} \\ 2 & \text{if only } c_L \text{ is defined} \end{cases}$$

where $c_L$ is the cost of the plan computed by the trained strategy.

[1] `http://icaps-conference.org/ipc2008/deterministic/`

This modified version of IPC score is only used as a reward in the training phase. Naturally, when the trained planner is evaluated, it is treated on par with all the reference planners and the cost of its plan is included in the minimum operator, effectively capping the maximum score for a single problem at 1.

**Squared IPC score** The second performance measure we consider is the IPC score raised to the power of two. The motivation behind it is to increase the emphasis on minimizing the cost of the plan. For example, a planner returning a plan of cost two times higher than the best known plan would only receive $(\frac{1}{2})^2 = \frac{1}{4}$ points. During training, we apply here the same modifications as for the plain IPC score. We additionally cap the reward at 2, to prevent excessive premium for performance strictly better than that of reference planners (similar to the plain IPC score, the reward for solving a problem not solved by any of the reference planners is 2). Formally, the score is defined in terms of $r_{ipc}$ as follows:

$$r_{ipc2} = \min(r_{ipc}^2, 2)$$

**Time score** The last performance objective we consider is based solely on the time required to find a plan and disregards the cost of the returned plan. We define it as follows:

$$r_{time} = \begin{cases} \frac{T-t}{T} & \text{if solved} \\ 0 & \text{otherwise} \end{cases}$$

where $T$ is the total time limit under which the planner is run and $t$ is the actual time after which the solution was found. The maximum score a planner can receive is 1, which is awarded for solving the problem immediately. The score decreases proportionally to the time used by the planner, down to 0 for the case when all available time is used. Failing to solve the problem also results in a score of 0. The rationale behind this score is to incentivize the planner to a plan as quickly as possible, no matter its quality. In disregarding the plan cost, our time score is related to the *coverage* score, which is 1 if the problem is solved in time and 0 otherwise.

Unlike coverage, the time score assigns higher reward for problems solved sooner. As a result, it retains more information than coverage and better expresses the goal of finding a plan as quickly as possible.

## 5.5 Evaluation

In this section, we present detailed results of experimental evaluation of the learning learning approach introduced in this chapter.

### 5.5.1 Experimental Setup

The learner was evaluated using five IPC domains: *Transport*, *Parking*, *Elevators*, *No-mystery* and *Floortile*. These are all the domains of the learning track of IPC 2014 [VCG+15], with the exception of the *Spanner* domain, which was designed specifically not to work well with delete-relaxation heuristics, such as FF, which we use throughout our experiments. This is also a superset of the domains used in Chapter 4. The time allocated for solving a problem was 5 seconds. The motivation behind the time limit is twofold. First, the training process requires multiple runs of the planner (each of the runs corresponds to a single learning episode). Given fixed computational resources, operating on a smaller time scale allows for performing more episodes. Working with substantially larger time limits would also be challenging because of the credit assignment problem, which arises because the planner's performance can only be assessed after its run. Since the reward is only assigned at the end of an episode, longer episodes would further increase the average number of time steps between an action and the corresponding reward.

As in Chapter 4, for each of the five domains, we used the problem generator provided by the organizers of the Learning Track at IPC 2014[2]. The parameters passed to the generators (such as the number of trucks and packages for *Transport*) were chosen so as to match the 5-second

[2] `http://www.cs.colostate.edu/~ipc2014/`

| IPC reward | | | | | |
|---|---|---|---|---|---|
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | **0.98** | 0.01 | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| near late | 0.01 | $\sim 0$ | **0.97** | 0.02 | $\sim 0$ |
| far early | $\sim 0$ | $\sim 0$ | $\sim 0$ | **1.00** | $\sim 0$ |
| far late | 0.01 | 0.01 | **0.94** | 0.01 | 0.02 |
| IPC^2 reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.01 | **0.98** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| near late | 0.14 | **0.22** | **0.60** | 0.04 | $\sim 0$ |
| far early | **0.96** | 0.02 | 0.01 | $\sim 0$ | $\sim 0$ |
| far late | 0.02 | 0.02 | **0.91** | 0.05 | 0.01 |
| Time reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.01 | 0.01 | **0.95** | 0.01 | 0.01 |
| near late | 0.04 | 0.03 | 0.05 | 0.09 | **0.78** |
| far early | 0.01 | $\sim 0$ | **0.97** | 0.01 | 0.01 |
| far late | 0.19 | 0.17 | **0.23** | 0.19 | **0.21** |

Table 5.1: The policies learned for the *Transport* problems.

time limit. Specifically, we searched for parameter combinations such that around half of the generated problems could be solved within the limit by a baseline planner using GBFS guided by the FF heuristic [HN01] and unary operator costs. The same heuristic function was used with the learner and other baseline planners considered in this chapter.

We trained a separate model for every pair of a domain and a performance objective. The training was conducted using a single CPU (Intel Core i7-4790) for a period of 48 hours. The problems were sampled from the training set with uniform probability. Every sampled problem was attempted $N = 5$ times to compute the policy update averaged over N runs, according to equation 5.2. The learning rate was $\alpha = 0.02$. We remark that parallelization of the training process is possible as multiple runs of the planner can be performed in parallel. This remains a direction of future work.

### 5.5.2 Learned Policies

The policies learned for each of the domains and performance objectives are presented in Tables 5.1 - 5.5. Each of the tables reports the policies for one of the domains and all three objectives.

| IPC reward | | | | | |
|---|---|---|---|---|---|
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | $\sim 0$ | $\sim 0$ | $\sim 0$ | **0.99** |
| near late | 0.01 | 0.01 | 0.01 | 0.02 | **0.94** |
| far early | $\sim 0$ | $\sim 0$ | $\sim 0$ | $\sim 0$ | **0.99** |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| IPC^2 reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.2 | 0.02 | 0.01 | 0.15 | **0.80** |
| near late | 0.01 | 0.01 | 0.01 | 0.02 | **0.96** |
| far early | $\sim 0$ | $\sim 0$ | 0.01 | $\sim 0$ | **0.98** |
| far late | 0.19 | 0.19 | 0.19 | 0.19 | **0.25** |
| Time reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | $\sim 0$ | $\sim 0$ | 0.01 | **0.98** |
| near late | 0.19 | 0.19 | 0.18 | 0.19 | **0.25** |
| far early | 0.01 | 0.01 | 0.01 | 0.01 | **0.97** |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |

Table 5.2: The policies learned for the *Parking* problems.

The rows represent possible states of the search. Each of the entries denotes the probability of selecting the routine given by the column in the state given by the row. Probabilities higher than 0.2 (assigned by the uniform policy) are highlighted in bold.

In most states, the learned policy approaches a deterministic one. The probability of choosing the most dominant routine is typically over 0.9. For some cases, however, the policy remains nondeterministic. An expected reason for this is when two routines are clearly preferred over the rest (e.g., the *near late* state in *No-mystery* domain under the time reward, see Figure 5.6). If the combination of the policies is not significantly better than either of them on its own, the policy may also oscillate between them (e.g., the *near early* state in *Elevators* domain under the IPC reward, see Figure 5.5). Another reason for nondeterminism is that the state occurs rarely in training. In the extreme case of the *far late* state in *Parking*, the state does not occur at all when training with the IPC and time rewards, and the policies for the state remain uniform. To shed some light on the process of learning the policies, we plot how the action probabilities change over the course of training for several representative cases (Figures 5.1-5.6).

In the logistics-type domains of *Tranpsort* (Table 5.1) and *Elevators* (Table 5.3), the preferred routine changes with the state of the search. For example, the *Transport* policy trained with

| IPC reward | | | | | |
|---|---|---|---|---|---|
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.05 | **0.58** | 0.03 | **0.32** | 0.02 |
| near late | 0.01 | $\sim 0$ | 0.01 | **0.98** | $\sim 0$ |
| far early | 0.02 | 0.01 | **0.92** | 0.02 | 0.04 |
| far late | 0.11 | 0.09 | 0.11 | 0.16 | **0.53** |
| IPC^2 reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.01 | $\sim 0$ | **0.98** | $\sim 0$ | $\sim 0$ |
| near late | 0.01 | 0.01 | **0.97** | 0.01 | $\sim 0$ |
| far early | 0.01 | 0.03 | **0.45** | **0.51** | $\sim 0$ |
| far late | 0.14 | 0.14 | 0.19 | **0.28** | **0.25** |
| Time reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.02 | 0.01 | 0.09 | 0.08 | **0.79** |
| near late | 0.07 | 0.04 | **0.30** | **0.38** | **0.21** |
| far early | 0.03 | 0.03 | 0.04 | 0.04 | **0.86** |
| far late | 0.19 | 0.19 | **0.21** | 0.20 | **0.21** |

Table 5.3: The policies learned for the *Elevators* problems.

the IPC reward starts with local search (the *far early* state) and then switches to plain GBFS if fast progress has been made (the *near early* state) or random walks otherwise (the *late* states). The influence of various performance objectives can also be observed. For instance, the policies trained with the time reward tend to make heavier use of DFS, which is especially visible in the *Elevators* domain. Logistics-type domains are characterized by existence of multiple paths to the goal and lack of dead ends, so indeed it is often possible to quickly find low-quality solutions with a depth-biased approach.

The policies learned for the remaining three domains are less varied. In *Floortile* (Table 5.5) and *No-mystery*, (Table 5.4) the learner prefers relatively conservative routines of greedy and $\epsilon$-greedy BFS, irrespective of the chosen performance objective. This can be attributed to the presence of dead ends in these two domains, which makes aggressive exploration potentially harmful. The only exception is the *far late* state, which typically does not occur during training on these domains.

The opposite happens in the *Parking* domain (Table 5.2). Here, it is possible to search successfully even with very strong depth bias. It turns out that applying heuristic guided DFS allows for solving most of the problems with only very slight increase in the plan costs. The

| IPC reward | | | | | |
|---|---|---|---|---|---|
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | **0.93** | $\sim 0$ | 0.06 | 0.01 |
| near late | **0.94** | 0.05 | 0.01 | $\sim 0$ | $\sim 0$ |
| far early | 0.01 | **0.86** | 0.01 | 0.01 | 0.10 |
| far late | 0.18 | 0.19 | **0.24** | 0.20 | 0.19 |
| IPC^2 reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.06 | **0.87** | 0.01 | 0.02 | 0.04 |
| near late | 0.01 | **0.99** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| far early | 0.01 | **0.92** | 0.02 | 0.03 | 0.02 |
| far late | 0.19 | **0.22** | 0.20 | 0.20 | 0.20 |
| Time reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | 0.01 | **0.97** | 0.01 | $\sim 0$ | 0.01 |
| near late | **0.42** | **0.38** | 0.09 | 0.05 | 0.06 |
| far early | 0.01 | **0.96** | 0.01 | 0.01 | 0.01 |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |

Table 5.4: The policies learned for the *No-mystery* problems.

| IPC reward | | | | | |
|---|---|---|---|---|---|
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | **1.00** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| near late | **0.42** | **0.55** | $\sim 0$ | 0.01 | $\sim 0$ |
| far early | 0.01 | **0.98** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| IPC^2 reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | **1.0** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| near late | 0.02 | **0.97** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| far early | 0.01 | **0.97** | 0.01 | 0.01 | 0.01 |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| Time reward | | | | | |
| | GBFS | $\epsilon$-greedy | RW | Local | DFS |
| near early | $\sim 0$ | **1.00** | $\sim 0$ | $\sim 0$ | $\sim 0$ |
| near late | **0.79** | 0.19 | 0.01 | 0.01 | 0.01 |
| far early | 0.01 | **0.97** | 0.01 | 0.01 | 0.01 |
| far late | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |

Table 5.5: The policies learned for the *Floortile* problems.

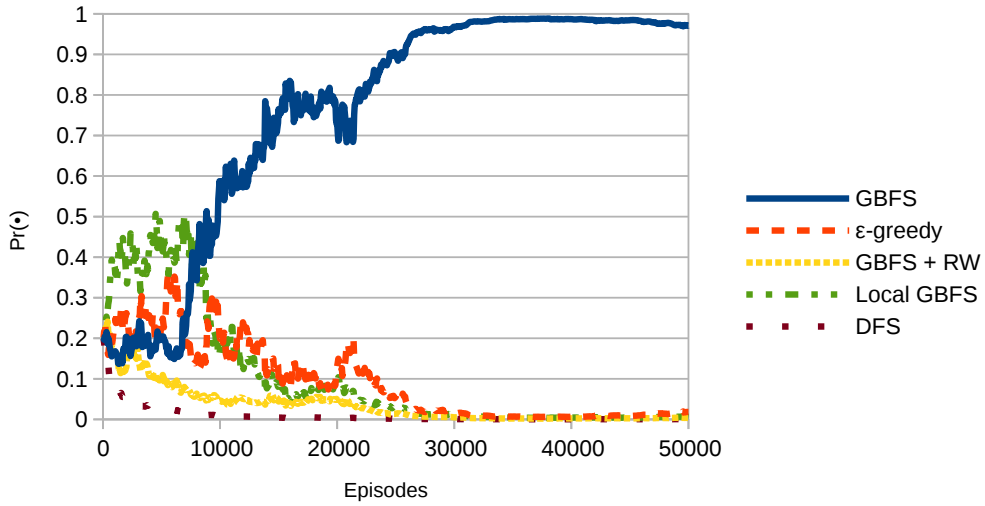



Figure 5.1: Action probabilities as a function of the number of episodes for *near early* state in *Transport* domain under the IPC reward. The learned policy chooses GBFS nearly deterministically. DFS is discarded particularly early because in *Transport* is results in plans of very high cost.

learner takes this opportunity and chooses DFS nearly deterministically, irrespective of the search state and performance objective. The only exceptions are in the *far late* state, where the policy remains close to uniform. This state occurs infrequently during training, because depth-focused policies tend to make quick progress towards the goal which results in transition to the *near* states. The same effect can be observed for the near late state under the time reward (see also Figure 5.4).

### 5.5.3 Results

#### Problems from the training distributions

We first assess the learned policies on problems following the same distributions as the training ones. For every domain, we generated 60 held-out test problems. We compare the policies against baselines corresponding to each of the search routines applied on their own. We also report the performance of a uniform policy, randomly choosing between the routines at every step. The results with respect to all three performance objectives are reported in Table 5.6. The learned policies (last three rows in each section of the table) consistently outperform most

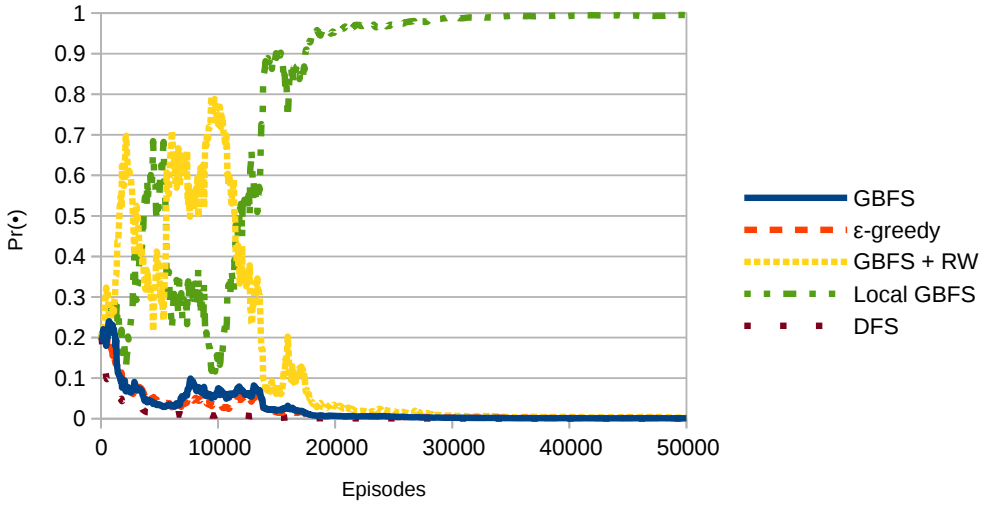



Figure 5.2: Action probabilities as a function of the number of episodes for *far early* state in *Transport* domain under the IPC reward. The policy oscillated between random walks and local search, before converging to the latter.

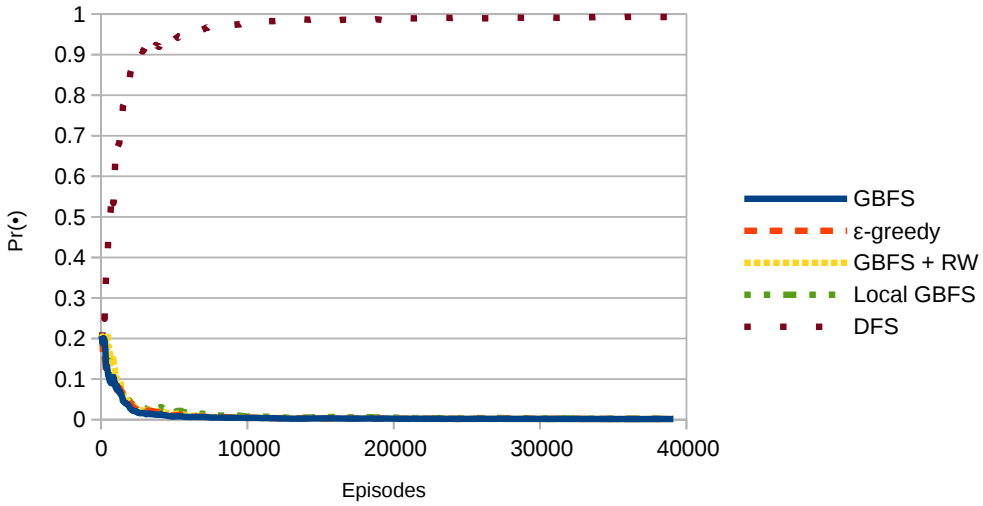



Figure 5.3: Action probabilities as a function of the number of episodes for *near early* state in *Parking* domain under the IPC reward. The policy quickly converges to DFS. In the considered distribution of *Parking* problems, DFS allows for quickly finding solutions without sacrificing the quality of the plans.

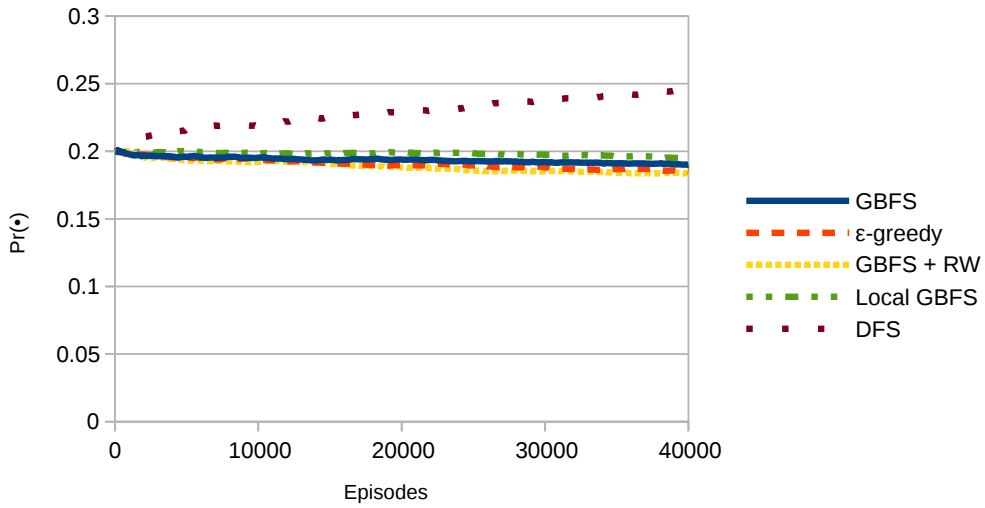



Figure 5.4: Action probabilities as a function of the number of episodes for *near late* state in *Parking* domain under the time reward. The *late* states occur infrequently during training because DFS-heavy policies make rapid progress towards the goal. This is why the policy in this state hardly diverges from uniform one.

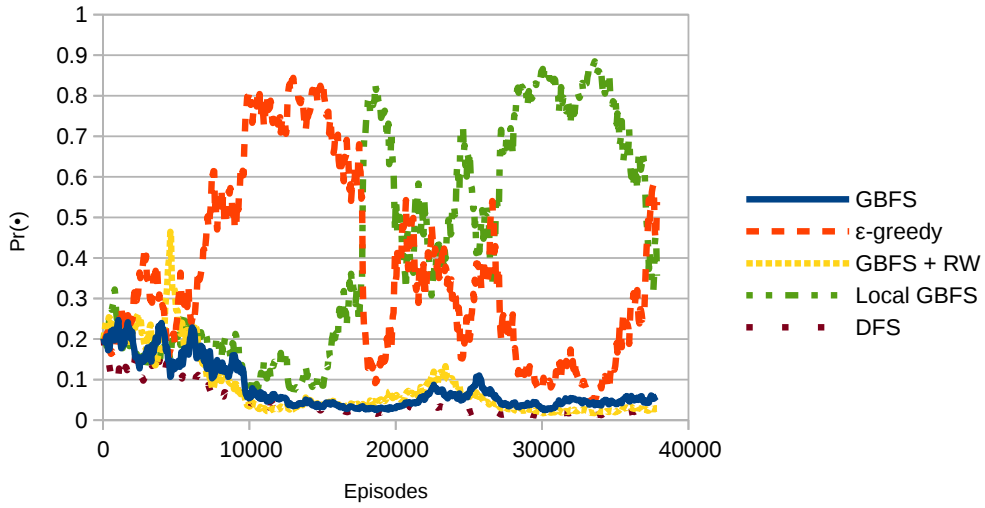



Figure 5.5: Action probabilities as a function of the number of episodes for *near early* state in *Elevators* domain under the IPC reward. The policy does not converge and oscillates between local and $\epsilon$-greedy search, both of which seem reasonable choices for the domain.

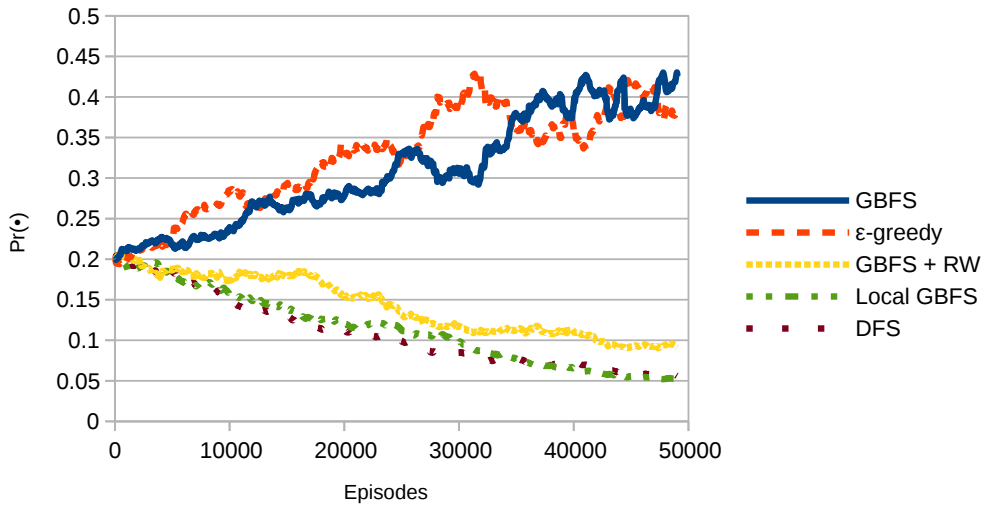



Figure 5.6: Action probabilities as a function of the number of episodes for *near late* state in *No-mystery* domain under the time reward. The policy mixes greedy and $\epsilon$-greedy search and prefers them over more aggressive routines, which in this domain are likely to lead to dead ends.

of the baselines, reaching the highest or nearly-highest scores. In the *Transport* domain, the scores obtained when learning with the corresponding reward are higher than all the baselines. In *Elevators*, the highest-performing routines (local search for IPC, GBFS for IPC^2 and DFS for time score) score slightly higher than the learner, which still outperforms most of the other baselines. In the remaining domains of *Parking*, *No-mystery* and *Floortile*, the learned policies perform very similarly to best single routines. This is in accordance with the form of the learned policies. The *Parking* policy almost always chooses DFS (Table 5.2), while the *No-mystery* and *Floortile* policies mostly use greedy and $\epsilon$-greedy search (Tables 5.4) and 5.5), which are closely related and perform very similarly in these two domains.

### IPC problems

Additionally, we evaluate the policies on larger problems sourced from IPC. We use the problems from the satisficing track of IPC 2011, which is the last one where the five domains occurred together. Each of the domains was represented by 20 problems of varying size. We initially set the timeout to the standard IPC value of 30 minutes. The results obtained by the learned policies and relevant baselines are reported in Table 5.7.

| IPC score | | | | | | |
|---|---|---|---|---|---|---|
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 29.46 | 15.57 | 28.73 | 24 | 28.27 | 126.03 |
| $\epsilon$-GBFS | 27.23 | 10.88 | 21.87 | **30.64** | **50.33** | 140.95 |
| Random Walks | 30.92 | 9.79 | 31.82 | 19.48 | 4.91 | 96.92 |
| Local Search | 37.18 | 21.89 | **32.87** | 20 | 23.9 | 135.84 |
| DFS | 12.95 | **37.45** | 16.92 | 7.7 | 0.13 | 75.15 |
| Uniform | 24.07 | 25.71 | 26.79 | 20.75 | 32.44 | 129.76 |
| Learned (IPC) | **38.85** | 37.27 | 32.67 | 26.62 | 49.83 | **185.24** |
| Learned (IPC^2) | 36.79 | 37.38 | 32.13 | 27.2 | 49 | 182.5 |
| Learned (Time) | 30.58 | 36.56 | 19.54 | 26.37 | 46.84 | 159.89 |
| IPC^2 score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 27.49 | 13.66 | **27.6** | 24 | 24.67 | 117.42 |
| $\epsilon$-GBFS | 22.82 | 9.52 | 20.87 | **30.5** | **44.69** | 128.4 |
| Random Walks | 19.26 | 8.41 | 24.27 | 19.38 | 3.95 | 75.27 |
| Local Search | 28.58 | 16.76 | 26.44 | 20 | 20.54 | 112.32 |
| DFS | 4.07 | 25.93 | 6.42 | 7.7 | 0.06 | 44.18 |
| Uniform | 13.73 | 19.71 | 18.24 | 20.71 | 28.33 | 100.72 |
| Learned (IPC) | **30.7** | 26.18 | 25.13 | 26.55 | 44.28 | **152.84** |
| Learned (IPC^2) | 30.55 | **26.51** | 24.57 | 27.1 | 43.59 | 152.32 |
| Learned (time) | 18.75 | 25.5 | 8.77 | 26.26 | 41.36 | 120.64 |
| Time score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 23.66 | 14.27 | 14.83 | 20.58 | 17.13 | 90.47 |
| $\epsilon$-GBFS | 21.48 | 10.15 | 10.44 | **21.58** | **38.47** | 102.12 |
| Random Walks | 39.93 | 9.35 | 21.28 | 16.77 | 3.96 | 91.29 |
| Local Search | 33.87 | 22.45 | 19.22 | 15.98 | 14.02 | 105.54 |
| DFS | 36.88 | **48.6** | **25.41** | 5.44 | 0.21 | 116.54 |
| Uniform | 34.55 | 27.75 | 20.25 | 17.18 | 19.71 | 119.44 |
| Learned (IPC) | 33.97 | 47.69 | 20.64 | 21.12 | 38.39 | 161.81 |
| Learned (IPC^2) | 29.27 | 46.98 | 21.11 | 21.56 | 37.9 | 156.82 |
| Learned (time) | **41.32** | 47.13 | 24.8 | 20.83 | 37.02 | **171.1** |

Table 5.6: IPC, IPC^2 and time score of the learned planning strategies and relevant baselines on 60 test problems randomly generated for each of the five domains. Average of 10 test runs. Highest values highlighted in **bold**.

The relative performance of the learned policies depends on the selected metric. They remain competitive as far the planning time is concerned, but loose ground when the cost of the plans is taken into account. The search algorithms attaining the highest IPC scores are local search, greedy and $\epsilon$-greedy best-first search. The policy trained with the IPC reward scores 2 points less than local and $\epsilon$-greedy search. This effect can be attributed to the *Parking* and *Elevators* problems being relatively easy for all of the involved planners under the original 30-minute time limit. In fact, all of the problems from these two domains were solved by all of the planners, with the exception of plain GBFS, which only failed one problem of each two domains. In this situation the overall score was decided mostly by the quality of the plan, generally putting the more conservative approaches like greedy and $\epsilon$-greedy search at an advantage: given sufficient time, global greedy search is likely to find solutions of lower cost than local search or random walks.

For this reason, we decided to perform another round of evaluation, using the same IPC problem sets but with a reduced timeout of 60 seconds. With less time, the problems become more challenging and better reflect the training arrangement, under which only half of the training problems could be solved by GBFS within the per-problem training time limit of 5 seconds. Indeed, it turns out that in 60 seconds GBFS is able to solve 6 out of 20 IPC *Parking* problems and 12 out of 20 *Elevators* problems. The results for this setting are reported in Table 5.8. The policies trained with IPC and IPC^2 rewards again obtained the highest scores, although their relative advantage over the baselines was smaller than on the training distributions. The policy minimizing solution time generalizes worse, but its time score is still above the baselines.

## 5.6 Summary

In this chapter, we introduced the first approach to learning search policies alternating between a number of search routines while solving a planning problem. We instantiated the approach using a simple two-dimensional representation of the planner's state and a set of five different search routines. Experimental evaluation shows that the system is capable of learning effective

| IPC score | | | | | | |
|---|---|---|---|---|---|---|
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | **17.8** | **19.48** | **9.94** | 6.48 | 53.7 |
| $\epsilon$-GBFS | 0.89 | 17.38 | 19.41 | 8.69 | **8.73** | 55.1 |
| Random Walks | 5.32 | 14.34 | 14.2 | 7.73 | 2.6 | 44.19 |
| Local Search | **8.29** | 15.24 | 16.96 | 8.92 | 6.03 | **55.44** |
| DFS | 6.7 | 16.47 | 10.94 | 6.89 | 0 | 41 |
| Uniform | 5.67 | 15.31 | 14.93 | 7.88 | 6.32 | 50.11 |
| Learned (IPC) | 6.37 | 15.61 | 15.82 | 7.69 | 7.54 | 53.03 |
| Learned (IPC^2) | 2.38 | 16.45 | 15.03 | 8.73 | 7.49 | 50.08 |
| Learned (time) | 5.39 | 16.65 | 11.93 | 8.66 | 7.45 | 50.08 |
| IPC^2 score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | **16.77** | **19.02** | **9.89** | 6.01 | 51.69 |
| $\epsilon$-GBFS | 0.79 | 15.39 | 18.9 | 8.43 | **8.48** | **51.99** |
| Random Walks | 3.78 | 11.35 | 10.55 | 7.53 | 2.29 | 35.5 |
| Local Search | **7.71** | 12.03 | 14.53 | 8.85 | 5.24 | 48.36 |
| DFS | 5.26 | 13.95 | 7.24 | 6.78 | 0 | 33.23 |
| Uniform | 3.78 | 12 | 11.7 | 7.77 | 5.74 | 40.99 |
| Learned (IPC) | 5.33 | 12.51 | 12.84 | 7.43 | 7.16 | 45.27 |
| Learned (IPC^2) | 1.48 | 13.85 | 11.62 | 8.49 | 7.04 | 42.48 |
| Learned (time) | 3.66 | 14.2 | 8.08 | 8.36 | 6.95 | 41.25 |
| Time score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | 15.66 | 18.64 | **9.28** | 6.66 | 50.24 |
| $\epsilon$-GBFS | 0.73 | 15.52 | 19.22 | 8.96 | 8.6 | 53.03 |
| Random Walks | 4.78 | 13.62 | 19.52 | 7.71 | 2.97 | 48.6 |
| Local Search | **7.2** | 18.65 | 19.51 | 7.93 | 6.58 | **59.87** |
| DFS | 5.51 | 18.79 | 19.41 | 6.88 | 0 | 50.59 |
| Uniform | 3.88 | 18.52 | **19.63** | 7 | 6.83 | 55.86 |
| Learned (IPC) | 2.3 | 19.01 | 17.45 | 7.99 | 7.77 | 54.52 |
| Learned (IPC^2) | 1.3 | **19.05** | 19.49 | 8.8 | **7.84** | 56.48 |
| Learned (time) | 4.39 | 19.04 | 19.54 | 8.76 | 7.82 | 59.55 |

Table 5.7: IPC, IPC^2 and time score of the learned planning strategies and relevant baselines on IPC-11 problems under 30 minute time limit. Highest values highlighted in **bold**.

| IPC score | | | | | | |
|---|---|---|---|---|---|---|
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | 5.89 | 11.56 | **8** | 3.62 | 29.07 |
| $\epsilon$-GBFS | 0 | 3.84 | 11.68 | 7.72 | 4.87 | 28.11 |
| Random Walks | 0.92 | 4.55 | 9.88 | 6.91 | 2.6 | 24.86 |
| Local Search | **2** | 9.98 | 11.63 | 6.98 | 3.64 | 34.23 |
| DFS | 0 | 8.88 | 7.47 | 7.77 | 0 | 24.12 |
| Uniform | 0 | 6.21 | 10.81 | 7 | 3.61 | 27.63 |
| Learned (IPC) | 0 | **12.27** | **11.78** | 7.72 | 5.7 | 37.47 |
| Learned (IPC^2) | 0 | 12.09 | 11.15 | 7.72 | **6.72** | **37.68** |
| Learned (time) | 0.48 | 7.59 | 7.67 | 7.72 | 4.76 | 28.22 |
| IPC^2 score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | 5.79 | 11.15 | **8** | 3.29 | 28.23 |
| $\epsilon$-GBFS | 0 | 3.7 | **11.39** | 7.48 | 4.75 | 27.32 |
| Random Walks | 0.85 | 4.15 | 8.05 | 6.83 | 2.29 | 22.17 |
| Local Search | **2** | 8.42 | 9.78 | 6.96 | 3.32 | 30.48 |
| DFS | 0 | 7.38 | 4.63 | 7.55 | 0 | 19.56 |
| Uniform | 0 | 4.9 | 8.8 | 7 | 3.29 | 23.99 |
| Learned (IPC) | 0 | 10.28 | 9.6 | 7.48 | 5.48 | 32.84 |
| Learned (IPC^2) | 0 | **10.69** | 9.32 | 7.48 | **6.48** | **33.97** |
| Learned (time) | 0.23 | 6.5 | 4.82 | 7.48 | 4.55 | 23.58 |
| Time score | | | | | | |
| | Transport | Parking | Elevators | No-mystery | Floortile | Sum |
| GBFS | 0 | 4.03 | 10.35 | 7.77 | 3.99 | 26.14 |
| $\epsilon$-GBFS | 0 | 2.95 | 10.07 | 7.86 | 4.96 | 25.84 |
| Random Walks | **0.72** | 3.05 | 10.3 | 6.3 | 2.97 | 23.34 |
| Local Search | 0.56 | 8.49 | 10.81 | 6.92 | 3.97 | 30.75 |
| DFS | 0 | 8.47 | **11.67** | 7.32 | 0 | 27.46 |
| Uniform | 0 | 5.51 | 10.76 | 6.46 | 3.98 | 26.71 |
| Learned (IPC) | 0 | **10.85** | 10.65 | 7.86 | 5.93 | 35.29 |
| Learned (IPC^2) | 0 | 10.33 | 11.02 | 7.73 | **6.9** | **35.98** |
| Learned (time) | 0.13 | 7.28 | 11.64 | **7.89** | 4.95 | 31.89 |

Table 5.8: IPC, IPC^2 and time score of the learned planning strategies and relevant baselines on IPC-11 problems under 60 second time limit. Highest values highlighted in **bold**.

search policies tailored to a specific problem distribution and performance objective. The policies can also be useful for problems outside the training distribution, provided that the interplay between the problem size and the time limit roughly matches the training setting. The approach is extensible: potential directions for future work include a richer state representation and adding further search routines, such as novelty-based search [LG12] and local search using a different heuristic function.

The limitations of this approach include operating with a number of fixed parameters, which imposes significant constraints on the policy space. Some of the parameters are properties of individual subroutines (for example, the number and length of random walks), while others relate to how the routines are integrated. The most prominent one is $t_r$, the time period over which the selected routine is applied. Another issue is the sensitivity of the learning process to the length of training traces, effectively enforcing training on relatively small problems. In Chapter 6, we address all of the above limitations by introducing a different search parametrization and a training scheme based on evolution strategies.

# Chapter 6

# Learning Search Policies with an Infinite Action Space

In this chapter, we consider learning search policies in an infinite action space, generated by a number of discrete and continuous parameters of a paramterized search routine template. This is in contrast with the work presented in Chapter 5, where the policies were tasked with choosing from a fixed set of distinct search routines. In reinforcement learning terms, the approach introduced in Chapter 5 dealt with a discrete action space, while the one presented here is closer in spirit to learning in a continuous action space.[1]

We start by introducing the notion of a *parametrized search routine template*. The template combines a number of search techniques in a flexible manner: depending on the values of the parameters, the routine can perform best-first search, random search, iterated local search or various combinations of the above. We then proceed to introduce the task of learning policies which set the values of the parameters given the current state of the search. As in Chapter 5, the objective is to optimize the planner's performance for a given distribution of planning problems. We model the policy using a feed-forward neural network and train it using a stochastic optimisation technique known as the *cross entropy method* (CEM).

[1]The action space is not strictly continuous as some of the search parameters are integers. However, the presence of real-valued search parameters makes the space infinite, hence the title of the chapter.

We also consider a simpler setting, in which we directly apply CEM to find the values of the search parameters. In this approach, there is no state-dependent search policy and the values of the search parameters remain fixed throughout the process of solving the planning problem.

## 6.1 Relation to Finite Action Spaces

The approach to learning search policies introduced in Chapter 5 relies on a discrete and finite action space. The learning planner is equipped with a number of fixed search routines, each constituting one action of the learner. This is a simple approach for combining a number of search mechanisms. However, it also has a number of limitations.

To begin with, using a fixed set of search routines requires a number of design choices, pertaining to each of the routines and the way of combining them. Many of the routines come with their own parameters, such as the value of $\epsilon$ for $\epsilon$-greedy search and the number and length of random walks. The choice of action duration time $t_r$ also has important implications on the ability of the algorithm to interleave between different techniques and on the effective span of the local search. The necessity to set the values of all these parameters seems at odds with the idea of learning the search policies. Intuitively, parameters like these should be trainable themselves, allowing the learning process to find the values that best suit a given problem distribution. In a more general solution, the values of the parameters could also be adjusted during search, for example to increase the amount of exploratory behaviour in the face of a plateau of the heuristic function.

Furthermore, the separation of the search techniques into a set of fixed routines imposes significant restrictions on the resulting search algorithms. For example, the learned policies have to choose between $\epsilon$-greedy and plain greedy ($\epsilon = 0$) search, possibly interleaving between the two, but they do not have the option of selecting an intermediate value of $\epsilon$. Similarly, the policies can choose between greedy search with or without random walks, but do not have the option of using an intermediate number or length of the walks. As another example, local and $\epsilon$-greedy search are made mutually exclusive, i.e. there is no option of randomized local search.

The approach presented in this chapter aims to address these limitations by combining various search techniques in a single parametrized search routine template. Using different values of the template's parameters, the search policies can navigate an infinite space of search algorithms, including, but not limited to best-first search, random search and iterated local search.

## 6.2 Parameterized Search Routine

The core idea behind the approach presented in this chapter is that techniques such as randomized order of node expansion, local search and random walks are not mutually exclusive and can be merged in a single parametrized search routine. When the values of the parameters are set accordingly, the parametrized routine can assume any of the techniques on their own. For intermediary values of the parameters, the routine includes elements of multiple techniques. For example, global and local search can be combined by interleaving between the two. A number of atomic steps of global search (for example, node expansions) can be followed by a period of local search. It is also possible to randomize the order of node expansion in both global and local search and additionally follow some of the expansions with a number of random walks.

One way of combining the search mechanisms is introduced in Algorithm 6. The algorithm is a general planner. Its arguments include the initial state $s_0$, the goal $g$ and a set of operators $\mathcal{O}$. The algorithm interleaves between global and local search. Nodes are expanded in a randomized ($\epsilon$-greedy) order. The expansions are optionally followed by a number of random walks from the expanded node.

The algorithm starts by initializing the global open list with the initial state (line 2). The main loop of the algorithm (lines 3-13) interleaves between global search (lines 5-7) and local search (lines 9-11). The global search is performed using the global open list. The period of global search is followed by the creation of a new local list, initialized using the best node from the global list (line 8). The local list is used to perform a number of local node expansions (lines 9-11). After that, all the nodes form the local queue are merged into the global one (line 12).

The number of steps of local and global search are determined using two parameters: $C$ which

is the total number of node expansions in a single iteration of the main loop, and $c$ which is the proportion of local expansions in $C$. It follows that a single iteration of the main loop contains $(1 - c) \cdot C$ global and $c \cdot C$ local expansions.

The global and local open lists are both processed using the STEP function (lines 16-42). If the open list given by the argument is not empty, the function extracts a single node from the open list (line 28) and expands it (line 33). All the successor states are added to the open list (line 34). Each expansion can optionally be followed by a number of random walks starting from the expanded node. The algorithm keeps track of the number of node expansion performed since the last drop of the lowest known value of the heuristic ($expansions_without_progress$). If the number exceeds the value of parameter $S$ (line 35), $N$ random walks of length $L$ are performed from the expanded node (line 37). All the states encountered during the random walk are added to the open list (line 38). If the open list given as argument is empty, the algorithm checks whether it is the global list (line 18). If so, the search fails. Otherwise, it attempts to continue by reseeding the local list with another state removed from the global one (lines 24).

Overall, the algorithm has the following parameters:

- $\epsilon$ – the probability of selecting a random node from the open list;
- $S$ – the threshold number of expansions without progress, above which random walks follow every expansion;
- $R$ – the number of random walks launched from a single node;
- $L$ – the length of a random walk;
- $C$ – the number of node expansions in one iteration of the main loop;
- $c$ – the proportion of local search in every iteration of the main loop.

The character of the search varies depending on the values of the parameters. For example, with $c = 0$, the algorithm results in a purely global search (no local expansions). If additionally $R = 0$ and $\epsilon = 0$, the algorithm degrades to plain GBFS. Given $c \neq 0$, the value of $C$ determines

**Algorithm 6** Parametrized planner

```
1: function PLANNER(s_0, g, O)
2:     global_open ← [s_0]
3:     while true do
4:         ε, S, R, L, C, c ← set_search_parameters()
5:         for i = 1 ... (1 − c) · C do
6:             STEP(global_open, O, g)
7:         end for
8:         local_open ← [pop(global_open)]
9:         for i = 1 ... c · C do
10:            STEP(local_open, O, g)
11:        end for
12:        merge local_open into global_open
13:    end while
14: end function
15:
16: function STEP(open, O, g)
17:    if open is empty then
18:        if open is global_open then
19:            return failure                          ▷ return from PLANNER
20:        else
21:            if global_open is empty then
22:                return failure                      ▷ return from PLANNER
23:            else
24:                local_open ← [pop(global_open)]
25:            end if
26:        end if
27:    end if
28:    s ← pop(open, ε)
29:    if s ∈ g then
30:        plan ← extract_plan(s)
31:        return plan                                 ▷ return from PLANNER
32:    end if
33:    successor_states ← expand(s, O)
34:    add(open, successor_states)
35:    if expansions_without_progress > S then
36:        for i = 1 ... R do
37:            walk_states ← random_walk(s, L)
38:            add(open, walk_states)
39:        end for
40:    end if
41:    return in_progress                              ▷ return to the PLANNER loop
42: end function
```

the frequency of interleaving between global and local search as well as the span of local search, i.e., the number of expansions performed using the local list before it is merged into the global one. The number and length of random walks, as well as the condition for triggering them are directly defined by $R$, $L$ and $S$ parameters.

## 6.3 Neural Search Policies

The parametrized search routine template introduced in Section 6.2 allows for creating various search routines by assigning specific values to the template's parameters. In this section, we discuss the problem of setting the search parameters given the current state of the search. In particular, we consider search policies in the form of a multi-valued function, mapping the state of the search to the values of the search routine's parameters.

We consider a neural model of the mapping, consisting of a feed-forward neural network. The network's inputs are the features of the planner's state. Its outputs are the values of the search routine's parameters $\langle \epsilon, S, R, L, C, c \rangle$.

The remainder of this section describes a concrete representation of the search state and a search policy model based on the feed-forward neural network.

### 6.3.1 State Representation

The representation of the planner's state considered in this chapter includes seven high-level features capturing the relevant information about the current problem, the progress of the search and the remaining time. In particular, we consider the following state features:

- the value of the heuristic for the initial state $h(s_0)$;
- the lowest value of the heuristic encountered within the search so far $h_{min}$;
- the time since the search started;

- the number of node expansions performed since $h_{min}$ last changed;
- the number of states the search has generated;
- the number of unique states the search has generated;
- the number of nodes the search has expanded.

The initial and minimal heuristic values allow for estimation of the progress towards the goal in the time elapsed. Note that in our training setting, detailed in Section 6.5, the time allocated for the planner is always the same, so the elapsed time feature implicitly encodes how much time there is left. These three features provide the policy with the same information as the two state features considered in Chapter 5, but retain more information by avoiding discretization. Furthermore, including the heuristic estimate for the initial state allows for differentiating between problems representing the same domain but varying in size. Another feature is the number of node expansions since the last change of lowest known heuristic value. This is a quantity which is also computed for the purpose of controlling random walks in the parametrized search routine. The number is a good indication of whether the search is facing a local minimum or a plateau of the heuristic. Additionally, the feature representation includes the standard search statistics computed by Fast Downward, such as the numbers of generated states and expanded search nodes.

### 6.3.2 Model Architecture

We model the state-dependent search policies using a feed-forward neural network with a single hidden layer. The inputs of the network are the state features listed in Section 6.3.1. The outputs are the search parameters $\langle \epsilon, S, R, L, C, c \rangle$. The number of hidden units is equal to the number of inputs. The hidden layer uses the sigmoid activation function $f(z) = \frac{1}{1-e^{-z}}$. The network outputs are treated according to the range of the search parameters they represent. For the real-valued search parameters of $\epsilon$ and $c$, which require values in $[0, 1]$ interval, the network outputs are passed through the sigmoid function. For the integer search parameters of

$S$, $R$, $L$ and $C$, the outputs are restricted to nonnegative values $y_r = \max(y, 0)$ and truncated to the integer part.

Since the planner's state features differ in the order of magnitude, we apply scaling to the inputs of the network to prevent the differences from affecting the learned policy. Without scaling, the policies would be biased to rely on features of larger magnitude. We determine the scaling factors for each problem distribution based on the maximum feature values observed by a reference planner on an additional batch of problems representing the distribution. The reference planner is simply our parametrized planner in a single hand-crafted configuration which makes moderate use of all the available search techniques by setting $\epsilon = 0.5$, $S = 10$, $R = 5$, $L = 10$, $C = 200$ and $c = 0.5$. Every feature $\phi$ is then linearly scaled using the maximum observed values, so that a scaled value of 1 corresponds to the recorded maximum:

$$\phi_{\text{scaled}} = \frac{\phi}{\phi_{\max}}$$

For example, if the initial heuristic value $h_0$ for a given problem is 100 and the highest initial heuristic value recorded in the normalization batch $h_{0,\max}$ is 400, the scaled value of the feature for the given problem is $\frac{h_0}{h_{0,\max}} = 0.25$. The feature scales are computed once for each of the problem distributions and remain fixed throughout training and evaluation of the policy. Note that the scaled value of a feature can be larger than 1 when the observed value of the feature is higher than the maximum over the scaling batch.

To ensure that the policies can manipulate the search parameters in adequate range, we also scale the outputs of the network. We choose the scaling factors so that a value of 1 of the original output corresponds to moderate use of the corresponding search technique. Specifically, we multiply the output defining the number of random walks R by 5, the length of the walks $L$ by 10, the stall condition $S$ by 10 and the length of the global-local cycle $C$ by 100. For example, if the output defining the number of random walks has the value of 0.5, the resulting number of random walks is $0.5 \cdot 10 = 5$. The values of $\epsilon$ and $c$ do not require scaling as the sigmoid function puts them in the desired range.

The following section introduces an approach for learning search policies tailored to given distributions of planning problems.

## 6.4 Learning Search Policies

In this section, we present an algorithm for training the neural search policies using the cross-entropy method, described in general terms in Section 2.4.1. We introduce a multivariate Gaussian distribution over the parameters $\theta$ of the policies. The distribution is itself parametrized by its mean vector $\mu$ and covariance matrix $\Sigma$:

$$\theta \sim \mathcal{N}(\mu, \Sigma)$$

The distribution is initialized to an isotropic Gaussian by setting $\mu$ to a zero vector and $\Sigma$ to an identity matrix. Then, at every iteration $t$, a number of policies ($n$) are sampled from the current distribution. The policies are evaluated on a number of problems sampled from the problem distribution $\mathcal{P}$. The mean of the $m$ best performing policies ($\nu_t$ in Equations 6.1-6.3) and their covariance are used to perform a smoothed update of the mean (Equation 6.2) and covariance (Equation 6.3) of parameter distribution for the next iteration.

$$\nu_t = \frac{1}{m} \sum_{i=1}^{m} \theta_i^t \tag{6.1}$$

$$\mu_{t+1} = (1 - \alpha)\mu_t + \alpha\nu_t \tag{6.2}$$

$$\Sigma_{t+1} = (1 - \alpha)\Sigma_t + \alpha \frac{1}{m-1} \sum_{i=1}^{m} (\theta_i^t - \nu_t)(\theta_i^t - \nu_t)^\top \tag{6.3}$$

Pseudocode of the approach is presented in Algorithm 7. The inputs of the algorithm include a distribution of planning problems $\mathcal{P}$, the number of iterations $u$, the number of problems sampled at each iteration $r$, the number of policies sampled at each iteration $n$ and the number of best-performing policies used to update the distribution $m$. The algorithm starts by initializing the distribution to an isotropic Gaussian (line 2). The main loop of the algorithm samples $r$

problems from the distribution $\mathcal{P}$ (line 4). Note that in general the problems do not have to be independently and identically distributed. In fact, in our evaluation (Section 6.5), we use problem batches approximating the IPC problem sets, which contain problems of increasing size. Further, the algorithm samples $n$ policies from the current distribution (line 5). All of the $n$ policies are used to tackle all of the $r$ problems (lines 6-10). The information about the success or failure of the planner and the plan cost are recorded for the purpose of computing the policies' IPC scores (line 11). The scores are used to sort the policies and designate the $m$ highest performing ones. The statistics of the $m$ policies are then used to update the distribution for the next iteration, according to Equations 6.2 and 6.3 (line 13). The output of the algorithm is the mean of the final policy distribution (line 15).

**Algorithm 7** Learning Search Policies

```
 1: function TRAIN(P, u, r, n, m)
 2:     initialize μ and Σ
 3:     for i = 1...u do
 4:         p_1...p_r ← P                                  ▷ sample r problems
 5:         θ_1...θ_n ← N(μ, Σ)                            ▷ sample n parameter vectors
 6:         for j = 1...n do
 7:             for k = 1...r do
 8:                 run planner parametrized by θ_j on p_k, record plan cost c_{j,k}
 9:             end for
10:         end for
11:         G_1...G_n ← compute IPC score for θ_1...θ_n
12:         sort θ_1...θ_n by scores G_1...G_n (highest first)
13:         update μ and Σ according to Equations 6.2 and 6.3
14:     end for
15:     return μ
16: end function
```

The learning algorithm presented in this section is evaluated empirically in Section 6.5.

### 6.4.1 Learning the Search Parameters Directly

The same approach can also be used to learn the search parameters directly. In this simpler setting, the neural model does not appear and the vector $\theta$ simply stores the values of the search parameters $\theta = \langle \epsilon, S, R, L, C, c \rangle$. The values of the parameters remain fixed throughout the run of the planner, without regard to the current state of the search.

The training scheme follows Algorithm 7. At each iteration, $n$ vectors $\langle \epsilon, S, R, L, C, c \rangle$ are sampled from the current distribution. The resulting search configurations are evaluated on a number of training problems in order to move the distribution towards the $m$ highest-scoring configurations.

In Section 6.5 we evaluate this simplified approach along with the state-dependent neural policies.

## 6.5 Evaluation

In this section, we evaluate both variants of our approach: (a) the neural search policies depending on the state of the search, and (b) the simpler *stateless* approach, in which the search parameters are learned directly and remain fixed throughout the search. As in previous chapters, we evaluate our approach using five IPC domains of *Elevators*, *Floortile*, *No-mystery*, *Parking* and *Transport*. We train the search policies on problem distributions following the IPC problem sets and compare the performance of the learned policies against a number of relevant baseline planners. As in previous chapters, the heuristic used throughout the experiments is the *Fast Forward* (FF) heuristic [HN01].

We use the problem generators published by the organizers of the learning track of IPC 2014 (the same as in Chapter 5). The parameters passed to the problem generators are chosen to match the problems of the satisficing track of IPC 2011, which is the last edition in which the domains occurred together in the satisficing track. The parameters determine the size of the problems, by specifying quantities such as the numbers of locations, packages and trucks in the *Transport* domain, the number of cars and curbs in the *Parking* domain and so on. Note that the values of the parameters used to generate the IPC problems do not necessarily give a full account of their expected difficulty, because the selection of competition problems may have involved additional steps, such as manually rejecting problems which appear too easy or too hard for some set of reference planners. Throughout training and evaluation, we use a time limit of 3 minutes per problem. We remark that training with a higher 30-minute limit

traditionally used in IPC is possible, but much more expensive, requiring up to 10 times more computation time per policy and problem.

### 6.5.1 Training

We trained the policies using Algorithm 7 with a population size $n = 50$ and an elite size $m = 10$. The population size was primarily motivated by the amount of computational resources required for training (note that every evaluation of a sampled policy on a batch of 20 problems amounts to 20 runs of the planner). In a limited set of experiments with a population ten times larger ($n = 500$, $m = 100$), we did not observe a significant improvement in the performance of the learned policies. Given the relatively small size of the population, we set the smoothing factor $\alpha$ to 0.7, the largest value in the range suggested by Mannor et al. [MRG03], who first applied the cross-entropy method to policy search. At every iteration, $r = 20$ problems were generated, using generator parameters matching the ones used for the IPC-2011 problem sets. For example, since the IPC set for *Parking* contains two problems with 22 cars, three problems with 24 cars and so on, the same holds for every batch of problems generated during training for this domain.

Training was performed using a total of 32 CPU cores. For the state-dependent neural search policies, the training time was 48 hours. Depending on the domain, this setup allowed between 41 (*Floortile*) and 91 (*Elevators*) iterations. However, most of the policy improvement appeared to occur in the first several iterations of the process. This is evidenced by the plots in Figure 6.1, which show the average IPC scores obtained by the policies sampled at each iteration. Note that the plots are not typical training curves. First, the policy distribution is not the only source of variability in the plots because every iteration uses a newly generated batch of problems. As mentioned before, the empirical difficulty of a problem can vary even for problems generated with the same generator parameters. Second, the IPC is a relative measure of planners' performance. In a hypothetical situation, where all members of the population improved the cost of their plans at iteration by the same proportion, their scores would remain the same. We only decided to include the plots after realising that the primary way in which

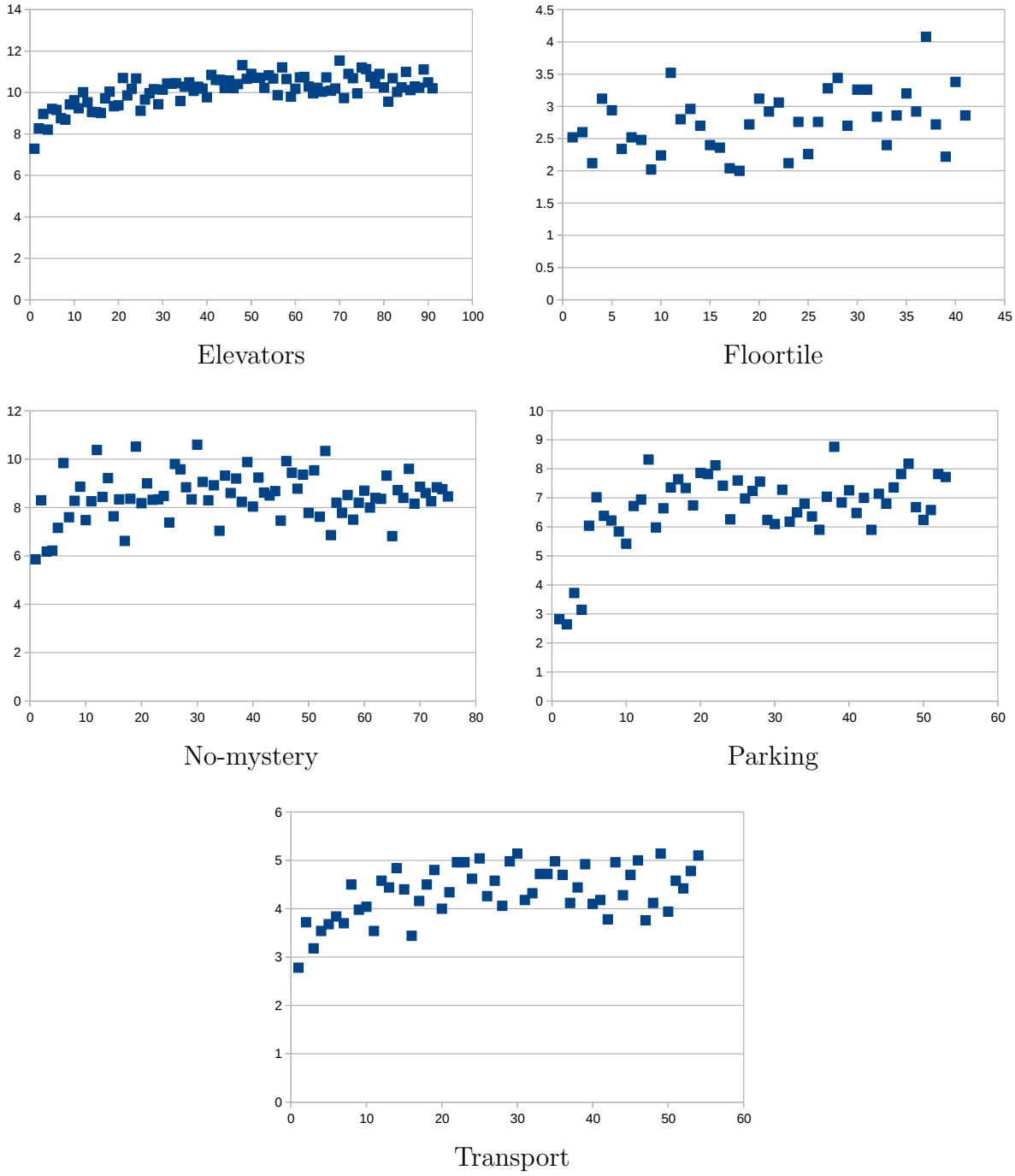



Figure 6.1: The average IPC score ($y$-axis) obtained on the training batch by the policies sampled at each iteration of the learning process ($x$-axis). The variance in score is largely a result of random generation of the training batch (every iteration has a newly generated set of problems). The dynamic of the scores still suggests that most of the policy improvement happens in the first handful of iterations.

the scores improved between iterations was by solving more problems. In fact, the plots for *coverage* look very similar. Between the two measures (IPC score and coverage), we decided to plot the IPC score as this is our performance objective.

The plots suggest that the policies did not improve significantly after the 20th iteration. In the extreme case of the *Floortile* domain, it is not clear from the plot whether the learner made any progress at all. However, the results discussed in the next section show that a reasonably effective policy has still been learned. When training the search parameters directly, the distribution over the parameters turned out to converge much faster and so we stopped training after the first 24 hours.

### 6.5.2 Quantitative Evaluation

We compare the learned search policies against a set of baseline planners, including planners using each of the involved search techniques on its own, as well as a hand-crafted combination of all of the techniques. We also compare against the search policies learned in Chapter 5, using a discrete action space. For plain GBFS, we use the Fast Downward implementation. For other baselines, we use our parametrized search routine, with the values of the parameters set accordingly. Overall, the set of baseline planners includes:

- plain greedy BFS [GBFS];
- $\epsilon$-greedy BFS with $\epsilon = 0.5$ ($R = 0$, $c = 0$) [$\epsilon$-GBFS];
- greedy BFS following a node expansion with $R = 5$ random walks of length $L = 10$ if the lowest known heuristic value has not changed for the last $S = 10$ expansions ($c = 0$) [Random Walks];
- local BFS with a span of $C = 200$ expansions ($R = 0$, $c = 1$) [Local Search];
- a combination of all the techniques, using search parameter values $\epsilon = 0.5$, $S = 10$, $R = 5$ and $L = 10$ (the same as the values of corresponding hyperparameters of individual

subroutines in Chapter 5), as well as $C = 200$ and $c = 0.5$ (i.e. 100 global expansions followed by 100 local expansions) [Mixed].

We evaluate the learned policies in two different settings. In the first one, we use the IPC-2011 problem sets. We perform 10 test runs of the learned policy and each of the baselines. In the second setting, we use a separate randomly generated batch of test problems for each of the 10 test runs. Evaluation on randomly generated problems allows us to assess the performance of the planners over a total of 200 problems per domain, rather than just 20. Furthermore, it shows the planner's performance on problems following the same distribution as the training ones, removing the effect of any additional problem selection which may have occurred in the construction of the IPC sets.

The IPC scores for all evaluated planners are reported in Table 6.1 (the IPC problem sets) and Table 6.2 (the randomly generated test sets). The learning approaches achieve the highest scores in all cases except the *No-mystery* IPC set, where they narrowly trail $\epsilon$-greedy search. In general, in the *No-mystery* domain, the performance of learners is very similar to greedy and $\epsilon$-greedy search. Remarkably, the *Mixed* configuration achieved the lowest overall scores. This shows that a naive combination of all involved techniques does not result in high performance and allows us to attribute the strength of the distribution-specific policies to the learner. The policies using our parametrized search routine generally perform better than the one relying on a discrete action space. An important exception is the *Parking* domain. Recall from Chapter 5 that the discrete *Parking* policy relies heavily on DFS, which turns out to quickly solve many problems without sacrificing the plan cost. DFS cannot be represented by our parametrized search routine in its current form. The closest available technique in terms of depth-bias is short-span local search. But in this case the depth-bias cannot be increased to the point of constructing DFS-like linear paths: in the extreme case of the span equal to 1, the search will behave like plain BFS. This is because every period of local search is started using a node removed from the *global* open list. As shown in the next section, short-span local search is indeed the approach chosen by the learners based on the parametrized search routine, especially in the stateless version.

| | Elevators | Floortile | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|---|
| GBFS | 16.09 | 4.13 | **8.84** | 8.04 | 0 | 37.1 |
| $\epsilon$-GBFS | 13.61 | 5.87 | 8.43 | 5.88 | 0 | 33.79 |
| Random Walks | 14.68 | 2.34 | 6.58 | 6.51 | 0.45 | 30.55 |
| Local Search | 15.29 | 4.37 | 5.79 | 10.52 | 0.73 | 36.7 |
| Mixed | 10.92 | 2.81 | 7.49 | 5.34 | 0.43 | 26.98 |
| Learned (discrete) | 14.03 | 6.16 | 7.98 | **12.82** | 1.27 | 42.25 |
| Learned (stateless) | 15.6 | 6.06 | 8.78 | 12.3 | **2.72** | **45.46** |
| Learned (NSP) | **16.62** | **6.28** | 8.56 | 11.09 | 1.01 | 43.56 |

Table 6.1: IPC scores for IPC problem sets (average of 10 runs).

Learning the search parameters directly, without regard to the state of the search turns out surprisingly effective compared to the more complex approach using state-dependent search policies. One possible explanation is that, while the neural search policies are strictly more expressive than the stateless model, the optimization process was able to better explore the smaller parameter space of the stateless case. When testing on unseen problems following the same distribution as the training ones, the state-dependent policy is only a little better than the stateless one. When the policies are tested outside the training distribution, the stateless version scores higher than the state-dependent one. With the caveat of the IPC problem sets containing a total of 100 problems (rather than the 1000 in our test sets), we can conjecture that the simpler model generalizes better to problems from outside the training distribution. It is important to note that in the case of the *Elevators* domain the state-dependent learner achieves significantly higher performance then the stateless one, in both problem sets, thanks to applying different approaches in different problems and search states. We elaborate on the details of the learned policies in the following section, where we discuss the parameters learned in the stateless case and qualitatively analyze the behavior of the state-dependent policies on selected validation problems.

### 6.5.3 Qualitative Evaluation

In this section, we investigate the search policies learned for each of the domains. For the case of learning the search parameters directly, we simply report their values and comment on the resulting behaviour on the search. For the state-dependent neural search policies, we report our

| | Elevators | Floortile | No-mystery | Parking | Transport | Sum |
|---|---|---|---|---|---|---|
| GBFS | 14.62 | 2.21 | 8.16 | 9.1 | 2.58 | 36.68 |
| $\epsilon$-GBFS | 13.05 | 2.6 | 8.91 | 7.32 | 2.66 | 34.53 |
| Random Walks | 14.56 | 0.47 | 6.76 | 7.81 | 3.47 | 33.07 |
| Local Search | 15.93 | 1.89 | 7.13 | 11.58 | 4.37 | 40.9 |
| Mixed | 11.55 | 1.23 | 6.67 | 5.95 | 2.79 | 28.21 |
| Learned (discrete) | 13.86 | 3.24 | 8.77 | **14.86** | 4.41 | 45.13 |
| Learned (stateless) | 14.59 | **3.46** | 8.83 | 13.4 | **5.24** | 45.52 |
| Learned (NSP) | **16.33** | 3.24 | **9.01** | 12.57 | 4.98 | **46.12** |

Table 6.2: IPC scores for randomly generated test problems (average over 10 sets).

observations from stepping through a number of validation problems and recording the search parameters output by the policies.

### The Stateless Case

Table 6.3 lists the values of the search parameters learned directly in the stateless setting, as well as the parameter values used for the baseline planners. The effective behaviour of the search ranges from pure global search for the *Elevators* domain to pure local search in the case of *Transport*. Interestingly, random walks are not used in any domain.

In *Elevators*, this behaviour is achieved by setting the cycle length $C$ to a very low value of 3. In the current implementation of the parametrized search routine, the local expansions proportion $c = 0.123$ still results in a cycle of 2 global and 1 local expansions. However, a single local expansion is qualitatively no different than a global expansion since the children of the locally expanded node are moved to the global list immediately after the first and only local step. As a result, the search behaves like global BFS, with a relatively small value of $\epsilon = 0.05$.

The *Floortile* parameters amount to interleaving between 161 global expansions and 17 local expansions, selecting a random node with a probability of $\epsilon \sim 0.09$. Note that the stall size $S = 1$ and random walk length $L = 10$ have no effect, since the number of random walks $R$ is 0. Another way of effectively turning off random walks is used for *No-mystery*, where the length of the walks is set to 1. Walks of length 1, starting from the expanded node can only reach the node's direct descendants, which have been placed in the open list already. Performing

| | $\epsilon$ | S | R | L | C | c | Remarks |
|---|---|---|---|---|---|---|---|
| Learned (stateless) | | | | | | | |
| Elevators | 0.050 | 0 | 0 | 0 | 3 | 0.123 | $\epsilon$-GBFS with $\epsilon \sim 0.05$ |
| Floortile | 0.169 | 1 | 0 | 10 | 178 | 0.090 | 161 global and 17 local expansions, $\epsilon \sim 0.17$ |
| No-mystery | 0.349 | 1 | 2 | 1 | 24 | 0.067 | 22 global and 2 local expansions, $\epsilon \sim 0.35$ |
| Parking | 0.030 | 6 | 1 | 0 | 6 | 0.654 | 2 global and 4 local expansions, $\epsilon \sim 0.03$ |
| Transport | 0.086 | 5 | 2 | 1 | 33 | 0.965 | $\sim$ local search with a span of 32 expansions |
| Baselines | | | | | | | |
| $\epsilon$-GBFS | 0.5 | 0 | 0 | 0 | 200 | 0.0 | $\epsilon$-GBFS; C has no effect when c = 0.0 |
| Random Walks | 0.0 | 10 | 5 | 10 | 200 | 0.0 | 5 walks of length 10 if stalled for 10 steps |
| Local Search | 0.0 | 0 | 0 | 0 | 200 | 1.0 | local search with a span of 200 expansions |
| Mixed | 0.5 | 10 | 5 | 10 | 200 | 0.5 | a combination of the above |

Table 6.3: The search parameters optimized for each of the settings and the baseline configurations.

random walks of length 1 is wasteful, but not too expensive as duplicate nodes are efficiently recognised and rejected by the open list. Otherwise, the search is mostly $\epsilon$-GBFS with two local expansions in a cycle of 24 and $\epsilon = 0.35$.

The search parameters for *Parking* result in a significant depth-bias achieved by performing short-span local search (a cycle of 2 global and 4 local expansions). This behavior is relatively close to DFS, both qualitatively and in terms of overall performance as evidenced by comparison to the discrete action space learner (recall from Chapter 5 that its search policy is based on DFS). Local search is also chosen for the *Transport* domain, in this case with a span of 32 to expansions. The cycle also includes a single global expansion, which probably does not have a significant effect on the search performance (as noted before, a single global expansion in a cycle is no different from a single local expansion).

### Neural Search Policies

To shed some light on the effective behaviour of the search driven by the learned neural search policies, we tested the policies on a number of validation problems, recording the parameter values output by the policies at every step (i.e., every iteration of the main loop of the parametrized planner). Our observations for each of the domains are reported below.

**Elevators** The learned policy solves the smallest problems with pure GBFS ($c = 0$, $\epsilon$ on the order of 0.01). When the search fails to progress for around 30 expansions, a large number ($> 15$) of short (3 steps) random walks follows every expansion. For larger instances, the walks are longer (9 steps) and there are more of them ($> 30$). In the largest problems, a small number of local expansions (2 in a global-local cycle of 50 expansions) is added towards the end of the search.

**Floortile** The policy learned for this domain mostly relies on $\epsilon$-GBFS, with relatively large values of $\epsilon$. The search typically starts with $\epsilon \approx 0.8$, which means that the value of the heuristic function is ignored most of the time. For the largest problems solved within the time limit, the value of $\epsilon$ decreases towards the end of the search.

**No-mystery** In this domain, the values of $\epsilon$ range are even higher than in *Floortile* and range between 0.75 and 0.98, increasing as $h_{min}$ decreases. The search is purely global and no random walks are performed. The very high values of $\epsilon$ suggest that the FF heuristic is not very helpful in this domain. This shows one of the limitations of the parametrized search routine in its current form. While the heuristic value is almost never used, it still needs to be computed for the purpose of ordering the open list. This requires a considerable amount of time. We aim to address this limitation in the future work. One possible solution would be to allow the parametrized approach to work with multiple open lists [RH10], without the requirement of evaluating all involved heuristics for all the states encountered in the search.

**Parking** In small *Parking* problems the search policy is close to BFS. This behaviour is obtained by setting the cycle length $C$ to a very small value. Note that a global-local cycle with 0 or 1 expansions is equivalent to BFS. For larger instances the search starts to interleave between global and local search in a balanced manner ($c \approx 0.5$), with a varying length of the cycle $C$.

**Transport** For the small *Transport* problems the learned policy remains close to $\epsilon$-GBFS, with the value of $\epsilon$ varying between 0.08 and 0.2. For larger instances, the policy behaves like iterated local search, with a span in the range between 10 and 150. Occasionally, the length of the cycle decreases to very low values, making the search equivalent to global GBFS. The value of $\epsilon$ tends to increase as the search progresses, reaching values of around 0.25.

## 6.6 Summary

In this chapter, we introduced a parametrized search algorithm, combining a number of search techniques in a single routine. Depending on the values of its parameters, the algorithm can assume one of the techniques on its own, or various combinations of multiple techniques. The parametrized algorithm forms the basis of the task of learning search policies, mapping the current state of the search to an assignment over the algorithm's parameters. In many ways this approach generalizes the work introduced in Chapter 5. First, it allows the learner to explore the space between the fixed search routines. Secondly, it reduces the number of intricate hyperparameters by handing over the control over critical quantities, such as the length of random walks and the span of local search to the learner itself. In addition to the search policies, we consider a simpler scenario in which the values of the search parameters are optimized directly and remain fixed throughout the search.

In both settings, the learned planners outperform the relevant baselines. With the current form of the parametrized search routine, the simpler stateless setup achieves performance remarkably close to the state-dependent policies. However, the search policies still have an advantage in varied problem distributions, where approaching different problems with different strategies is beneficial. This effect can be expected to be even more pronounced when training jointly on more varied sets of problems or with more complex variants of the parametrized search algorithm, for example including search techniques which can increase the chance of finding a plan at the expense of its cost, such as DFS in logistics-type domains.

# Chapter 7

# Conclusion

This thesis has introduced a number of novel ways of learning from the planner's experience to improve its future performance. In this chapter, we summarize the key contributions of the thesis and discuss possible directions of future research.

## 7.1 Summary of Thesis Achievements

In the first part of the thesis, we investigated learning domain-independent heuristic functions. We have introduced a domain-independent representation of the environment state, inspired by existing work on learning heuristics in a domain-dependent setting. We have demonstrated the potential of the approach to improve on the performance of the original heuristic. On the other hand, our experiments have shown that the behaviour of the search guided by the learned heuristic typically remains close to the original heuristic, and in particular its unary-cost version.

In the second part of the thesis, we introduced two approaches to learning search policies for classical planning. In the first one, the planner is equipped with a number of different search routines. Choosing between them is the task of a learnable stochastic policy, which can be trained to maximize selected performance objective for a given distribution of planning

problems. We have shown effectiveness of the approach, but also indicated its limitations, including action space limited to a discrete set of fixed search routines and a number of intricate hyperparameters.

We then continued the idea of learning search policies in a parametrized action setting akin to reinforcement learning in continuous action spaces. For the purpose of this work, we have introduced a novel parametrized search routine, combining elements of multiple search techniques. This setting allows the learning process to optimize quantities such as, for example, the span of the local search and the length of the random walks. We have presented two ways of using the parametrized search routine. In the more general one, we use a learnable policy to map the state of the search to the values of the routine's parameters. In a simpler setting, the parameters are optimized directly for a specific problem distribution. We have demonstrated the capacity of both approaches to find search policies outperforming relevant baselines on given problem distributions.

## 7.2 Future Work

Our work on learning heuristic functions has shown that with the current approach the behaviour of the search guided by the learned correction typically remains close to search with the original heuristic. An interesting challenge would be to learn heuristic functions without using features derived from the computation of an existing heuristic. We conjecture that to compete with existing heuristics which construct approximate plans, the learning architecture will need to incorporate some sort of iterative computation, present for example in graph neural networks [SGT+09, BHB+18]. At the moment of submission of this thesis, the first approaches in this spirit are starting to appear [STT20].

Regarding learning of search policies, a promising direction of future work is allowing the parametrized search routine to work with multiple open lists. This would open the way to extending the approach with some state-of-the-art search mechanisms such as *preferred operators* and alternating between multiple heuristic functions [RH10]. Furthermore, it would allow

leaving the choice of currently followed heuristic to the learned policy. Other possible additions to the parametrized search routine include novelty-based search [LG17] and beam search with learnable beam width.

Another interesting research direction is learning the representation of the planner's state. Capturing the relevant information in a domain-independent manner is highly challenging and remains the key to improving the expressiveness of the learned policies. One possible approach would be to use graph networks on graphs capturing relevant information about the problem at hand and the state of the search, including relaxed plans and dependencies between the operators.

In future work it may also be worth considering setting in which the heuristic function and the search policy are trained jointly, for example by interleaving between policy optimization and supervised learning of heuristics using the plans computed in the process of policy evaluation.

# Bibliography


[ACO19] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia. Learning action models with minimal observability. *Artificial Intelligence*, 275:104–137, 2019.

[AF17] Masataro Asai and Alex Fukunaga. Exploration Among and Within Plateaus in Greedy Best-First Search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017*, pages 11–19, 2017.

[AF18] Masataro Asai and Alex Fukunaga. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pages 6094–6101, 2018.

[Asa19] Masataro Asai. Unsupervised Grounding of Plannable First-Order Logic Representation from Images. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019*, pages 583–591, 2019.

[AZH11] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[BBV18] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, pages 10338–10349, 2018.

[BEMS05] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[BG01] Blai Bonet and Hctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5 – 33, 2001.

[BGV92] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992*, pages 144–152, 1992.

[BHB+18] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[BLBMT18] Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks. Technical report, Imperial College London, 2018.

[BLG97] Blai Bonet, Gábor Loerincs, and Hector Geffner. A Robust and Fast Action Selection Mechanism for Planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 714–719, 1997.

[BLP18] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *CoRR*, abs/1811.06128, 2018.

[BMPR17] Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, and Heorhi Raik. A context-aware framework for dynamic composition of process fragments in the internet of services. *J. Internet Services and Applications*, 8(1):6:1–6:23, 2017.

[BN95] Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–656, 1995.

[BSM+16] Antonio Bucchiarone, Martina De Sanctis, Annapaola Marconi, Marco Pistore, and Paolo Traverso. Incremental Composition for Adaptive By-Design Service Based Systems. In *IEEE International Conference on Web Services, ICWS 2016*, pages 236–243, 2016.

[Byl94] Tom Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[CAJ19] Sergio Jiménez Celorrio, Javier Segovia Aguas, and Anders Jonsson. A review of generalized planning. *Knowledge Eng. Review*, 34:e5, 2019.

[CDF16] Isabel Cenamor, Tomás De La Rosa, and Fernando Fernández. The IBaCoP planning system: Instance-based configured portfolios. *Journal of Artificial Intelligence Research*, 56:657–691, 2016.

[CLH18] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*, pages 1419–1426, 2018.

[CMS+18] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, pages 5032–5043, 2018.

[CS07] Andrew Coles and Amanda Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[dBKMR05] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Manor, and Reuven Y. Rubinstein. A Tutorial on the Cross-Entropy Method. *Annals OR*, 134(1):19–67, 2005.

[DHK15] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, 221:73–114, 2015.

[dlRCB08] Tomás de la Rosa, Sergio Jiménez Celorrio, and Daniel Borrajo. Learning Relational Decision Trees for Guiding Heuristic Planning. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008*, pages 60–67, 2008.

[dlRCFB11] Tomás de la Rosa, Sergio Jiménez Celorrio, Raquel Fuentetaja, and Daniel Borrajo. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research*, 40:767–813, 2011.

[DS77] Clive Dawson and Laurent Siklóssy. The Role of Preprocessing in Problem Solving Systems. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 465–471, 1977.

[FHN72] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(1972):251–288, 1972.

[FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189 – 208, 1971.

[GAR19] Pawel Gomoluch, Dalal Alrajeh, and Alessandra Russo. Learning Classical Planning Strategies with Policy Gradient. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019*, pages 637–645, 2019.

[GARB17] Pawel Gomoluch, Dalal Alrajeh, Alessandra Russo, and Antonio Bucchiarone. Towards learning domain-independent planning heuristics. In *Workshop on Architectures for Generality and Autonomy at the International Joint Conference on Artificial Intelligence, IJCAI 2017*, 2017.

[GARB20] Pawel Gomoluch, Dalal Alrajeh, Alessandra Russo, and Antonio Bucchiarone. Learning Neural Search Policies for Classical Planning. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, ICAPS 2020 (forthcoming)*, 2020.

[GKLP16] Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Learning to Rank for Synthesizing Planning Heuristics. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 3089–3095, 2016.

[GSV09] Alfonso Gerevini, Alessandro Saetti, and Mauro Vallati. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009*, pages 350–353, 2009.

[GWA+18] Arthur Guez, Theophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to Search with MCTSnets. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, pages 1817–1826, 2018.

[Hel06] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[HG08] Malte Helmert and Hector Geffner. Unifying the Causal Graph and Additive Heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008*, pages 140–147, 2008.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:263–312, 2001.

[HO01] Nikolaus Hansen and Andreas Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[HRK11] Malte Helmert, Gabriele Röger, and Erez Karpas. Fast downward stone soup: A baseline for building planner portfolios. *ICAPS 2011 Workshop on Planning and Learning*, pages 28–35, 2011.

[IK11] Tatsuya Imai and Akihiro Kishimoto. A Novel Technique for Avoiding Plateaus of Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011*, 2011.

[JDFB12] Sergio Jimenez, Tomas De La Rosa, Fernando Fernandez, and Daniel Borrajo. A Review of Machine Learning for Automated Planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012.

[Joa02] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 133–142, 2002.

[Kha99] Roni Khardon. Learning Action Strategies for Planning Domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[Kor85] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement Learning of Theorem Proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, pages 8836–8847, 2018.

[LCO15] Carlos Linares López, Sergio Jiménez Celorrio, and Angel García Olaya. The deterministic part of the seventh international planning competition. *Artificial Intelligence*, 223:82–119, 2015.

[LG12] Nir Lipovetzky and Hector Geffner. Width and Serialization of Classical Planning Problems. In *ECAI 2012 - 20th European Conference on Artificial Intelligence*, pages 540–545, 2012.

[LG17] Nir Lipovetzky and Hector Geffner. Best-First Width Search: Exploration and Exploitation in Classical Planning. *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 3590–3596, 2017.

[LRS18] Gil Lederman, Markus N. Rabe, and Sanjit A. Seshia. Learning Heuristics for Automated Reasoning through Deep Reinforcement Learning. *CoRR*, abs/1807.08058, 2018.

[LZ98] Christopher Leckie and Ingrid Zukerman. Inductive learning of search control rules for planning. *Artificial Intelligence*, 101:63–98, 1998.

[MG04] Mario Martín and Hector Geffner. Learning Generalized Policies from Planning Examples Using Concept Languages. *Applied Intelligence*, 20(1):9–19, 2004.

[MGH+98] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[MRG03] Shie Mannor, Reuven Rubinstein, and Yohai Gat. The Cross Entropy Method for Fast Policy Search. In *ICML'03 Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, pages 512–519, 2003.

[OGL+15] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L. Lewis, and Satinder P. Singh. Action-Conditional Video Prediction using Deep Networks in Atari Games.

In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*, pages 2863–2871, 2015.

[PLV+17] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sébastien Racanière, David P. Reichert, Theophane Weber, Daan Wierstra, and Peter W. Battaglia. Learning model-based planning from scratch. *CoRR*, abs/1707.06170, 2017.

[PZ08] Marek Petrik and Shlomo Zilberstein. Learning Heuristic Functions through Approximate Linear Programming. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008*, pages 248–255, 2008.

[RH10] Gabriele Röger and Malte Helmert. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, pages 246–249, 2010.

[Rub99] Reuven Rubinstein. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, 1:127–190, 1999.

[RW10] Silvia Richter and Matthias Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[SAH+19] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *CoRR*, abs/1911.08265, 2019.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[SGT+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[SHCS17] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR*, abs/1703.03864, 2017.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SKS+19] Silvan Sievers, Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Patrick Ferber. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 7715–7723, 2019.

[SLB+19] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[SSGB08] Richard S. Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael H. Bowling. Dyna-Style Planning with Linear Function Approximation and Prioritized Sweeping. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 528–536, 2008.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[STT20] William Shen, Felipe W. Trevizan, and Sylvie Thiébaux. Learning Planning Heuristics with Hypergraph Networks. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, ICAPS 2020 (forthcoming)*, 2020.

[Sut90] Richard S. Sutton. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[SvHH+17] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P. Reichert, Neil C. Rabinowitz, André Barreto, and Thomas Degris. The Predictron: End-To-End Learning and Planning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pages 3191–3199, 2017.

[TTTX18] Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action Schema Networks: Generalised Policies With Deep Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pages 6294–6301, 2018.

[VBA13] Jesus Virseda, Daniel Borrajo, and Vidal Alcazar. Learning heuristic functions for cost-based planning. *Preprints of the ICAPS'13 PAL Workshop on Planning and Learning*, pages 6–13, 2013.

[VCG+15] Mauro Vallati, Lukás Chrpa, Marek Grzes, Thomas Leo McCluskey, Mark Roberts, and Scott Sanner. The 2014 International Planning Competition: Progress and Trends. *AI Magazine*, 36(3):90–98, 2015.

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*, pages 2692–2700, 2015.

[VSSX14] Richard Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie. A Comparison of Knowledge-Based GBFS Enhancements and Knowledge-Free Exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014*, 2014.

[Wil92] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, 1992.

[WSG+14] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural Evolution Strategies. *Journal of Machine Learning Research*, 15(1):949–980, 2014.

[XFY07] Yuehua Xu, Alan Fern, and Sung Wook Yoon. Discriminative Learning of Beam-Search Heuristics for Planning. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2041–2046, 2007.

[XFY09] Yuehua Xu, Alan Fern, and Sung Wook Yoon. Learning Linear Ranking Functions for Beam Search with Application to Planning. *Journal of Machine Learning Research*, 10:1571–1610, 2009.

[XFY10] Yuehua Xu, Alan Fern, and Sung Wook Yoon. Iterative Learning of Weighted Rule Sets for Greedy Search. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, pages 201–208, 2010.

[XMH14] Fan Xie, Martin Müller, and Robert Holte. Adding Local Exploration to Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2388–2394, 2014.

[XMHI14] Fan Xie, Martin Müller, Robert Holte, and Tatsuya Imai. Type-based exploration with multiple search queues for satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2395–2402, 2014.

[YFG06] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning Heuristic Functions from Relaxed Plans. In *Proceedings of the Sixteenth International Conference*

*on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 162–171, 2006.

[YFG08] Sungwook Yoon, Alan Fern, and Robert Givan. Learning Control Knowledge for Forward Search Planning. *The Journal of Machine Learning Research*, 9:683–718, 2008.

[YWJ07] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3):107–143, 2007.