Machine Learning for Classical Planning: Neural Network Heuristics, Online Portfolios, and State Space Topologies

Inauguraldissertation

zur
Erlangung der Würde eines
Doktors der Philosophie/Doktors der Ingenieurwissenschaften
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel
&
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von

PATRICK CHRISTOPH FERBER

Basel/Saarbrücken, 2022

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät der Universität Basel auf Antrag von

Prof. Dr. Malte Helmert,
Universität Basel, Schweiz, Erstbetreuer

Prof. Dr. Jörg Hoffmann,
Universität des Saarlandes, Deutschland, zusätzlicher Erstbetreuer

Prof. Dr. Heiko Schuldt,
Universität Basel, Schweiz, Zweitbetreuer

Prof. Sylvie Thiébaux, PhD,
The Australian National University, Australien, externe Expertin

Prof. Scott Sanner, PhD,
University of Toronto, Kanada, externer Experte

Basel, den 15.11.2022

Prof. Dr. Marcel Mayor,
Universität Basel, Dekan

Tag des Kolloquiums	17.11.2022
Ort des Kolloquiums	Basel, Schweiz
Dekane der Fakultäten	
Universität Basel	Prof. Dr. Marcel Mayor
Universität des Saarlandes	Univ.-Prof. Dr. Jürgen Steimle
Prüfungsvorsitzender	Prof. Dr. Ivan Dokmanić
Gutachter	Prof. Dr. Malte Helmert
	Prof. Dr. Jörg Hoffmann
	Prof. Sylvie Thiébaux, PhD
	Prof. Scott Sanner, PhD
Beisitz	Prof. Dr. Heiko Schuldt

To my wife

Abstract

State space search solves navigation tasks and many other real world problems. Heuristic search, especially greedy best-first search, is one of the most successful algorithms for state space search. We improve the state of the art in heuristic search in three directions.

In Part I, we present methods to train neural networks as powerful heuristics for a given state space. We present a universal approach to generate training data using random walks from a (partial) state. We demonstrate that our heuristics trained for a specific task are often better than heuristics trained for a whole domain. We show that the performance of all trained heuristics is highly complementary. There is no clear pattern, which trained heuristic to prefer for a specific task. In general, model-based planners still outperform planners with trained heuristics. But our approaches exceed the model-based algorithms in the Storage domain. To our knowledge, only once before in the Spanner domain, a learning-based planner exceeded the state-of-the-art model-based planners.

A priori, it is unknown whether a heuristic, or in the more general case a planner, performs well on a task. Hence, we trained online portfolios to select the best planner for a task. Today, all online portfolios are based on handcrafted features. In Part II, we present new online portfolios based on neural networks, which receive the complete task as input, and not just a few handcrafted features. Additionally, our portfolios can reconsider their choices. Both extensions greatly improve the state-of-the-art of online portfolios. Finally, we show that explainable machine learning techniques, as the alternative to neural networks, are also good online portfolios. Additionally, we present methods to improve our trust in their predictions.

Even if we select the best search algorithm, we cannot solve some tasks in reasonable time. We can speed up the search if we know how it behaves in the future. In Part III, we inspect the behavior of greedy best-first search with a fixed heuristic on simple tasks of a domain to learn its behavior for any task of the same domain. Once greedy best-first search expanded a progress state, it expands only states with lower heuristic values. We learn to identify progress states and present two methods to exploit this knowledge. Building upon this, we extract the bench transition system of a task and generalize it in such a way that we can apply it to any task of the same domain. We can use this generalized bench transition system to split a task into a sequence of simpler searches.

In all three research directions, we contribute new approaches and insights to the state of the art, and we indicate interesting topics for future work.

Zusammenfassung

Viele Alltagsprobleme können mit Hilfe der Zustandsraumsuche gelöst werden. Heuristische Suche, insbesondere die gierige Bestensuche, ist einer der erfolgreichsten Algorithmen für die Zustandsraumsuche. Wir verbessern den aktuellen Stand der Wissenschaft bezüglich heuristischer Suche auf drei Arten.

Eine der wichtigsten Komponenten der heuristischen Suche ist die Heuristik. Mit einer guten Heuristik findet die Suche schnell eine Lösung. Eine gute Heuristik für ein Problem zu modellieren ist mühsam. In Teil I präsentieren wir Methoden, um automatisiert gute Heuristiken für ein Problem zu lernen. Hierfür generieren wird die Trainingsdaten mittels Zufallsbewegungen ausgehend von (Teil-) Zuständen des Problems. Wir zeigen, dass die Heuristiken, die wir für einen einzigen Zustandsraum trainieren, oft besser sind als Heuristiken, die für eine Problemklasse trainiert wurden. Weiterhin zeigen wir, dass die Qualität aller trainierten Heuristiken je nach Problemklasse stark variiert, keine Heuristik eine andere dominiert, und es nicht vorher erkennbar ist, ob eine trainierte Heuristik gut funktioniert. Wir stellen fest, dass in fast allen getesteten Problemklassen die modellbasierte Suchalgorithmen den trainierten Heuristiken überlegen sind. Lediglich in der Storage Problemklasse sind unsere Heuristiken überlegen.

Oft ist es unklar, welche Heuristik oder Suchalgorithmus man für ein Problem nutzen sollte. Daher trainieren wir online Portfolios, die für ein gegebenes Problem den besten Algorithmus vorherzusagen. Die Eingabe für das online Portfolio sind bisher immer von Menschen ausgewählte Eigenschaften des Problems. In Teil II präsentieren wir neue online Portfolios, die das gesamte Problem als Eingabe bekommen. Darüber hinaus können unsere online Portfolios ihre Entscheidung einmal korrigieren. Beide Änderungen verbessern die Qualität von online Portfolios erheblich. Weiterhin zeigen wir, dass wir auch gute online Portfolios mit erklärbaren Techniken des maschinellen Lernens trainieren können.

Selbst wenn wir den besten Algorithmus für ein Problem auswählen, kann es sein, dass das Problem zu schwierig ist, um in akzeptabler Zeit gelöst zu werden. In Teil III zeigen wir, wie wir von dem Verhalten einer gierigen Bestensuche auf einfachen Problemen ihr Verhalten auf schwierigeren Problemen der gleichen Problemklasse vorhersagen können. Dieses Wissen nutzen wir, um die Suche zu verbessern. Zuerst zeigen wir, wie man Fortschrittszustände erkennt. Immer wenn gierige Bestensuche einen Fortschrittszustand expandiert, wissen wir, dass es nie wieder einen Zustand mit gleichem oder höheren heuristischen Wert expandieren wird. Wir präsentieren zwei Methoden, die dieses Wissen verwenden. Aufbauend auf dieser Arbeit lernen wir von einem Problem, wie man jegliches Problem der gleichen Problemklasse in eine Reihe von einfacheren Suchen aufteilen kann.

Acknowledgements

I thank all the wonderful people who accompanied me on my long journey. Without your support, this thesis would not be finished.

First and foremost, I thank Jörg Hoffmann and Malte Helmert. I am deeply grateful that Jörg introduced me to planning, gave me the opportunity to write my Master's thesis, and recommended me when I moved southwards. Both of you went the additional administrative mile to supervise me together. Thank you very much for fitting me so often in your tight schedules, for introducing me to members in our community, and for bearing with me. I also thank Scott Sanner and Sylvie Thiébaux for agreeing to be my external reviewers. I still remember a discussion I had with Scott at my second ICAPS in Berkley. I thank Michael Katz. You provided me the opportunity to intern with IBM, learn how research can look like in a company, and caused me to divert from my perceived clear line of work into new topics.

I also want to thank all my colleagues in Basel and Saarbrücken Álvaro Torralba, Augusto B. Corrêa, Cedric Geissmann, Clemens Büchner, Daniel Fišer, Daniel Gnad, Daniel Höller, Florian Pommerening, Gabriele Röger, Guillem Francès, Jendrik Seipp, Julia Wichlacz, Liat Cohen, Manuel Heusner, Marcel Steinmetz, Marcel Vinzent, Maximilian Fickert, Patrick Speicher, Rebecca Eifler, Remo Christen, Salomé Eriksson, Silvan Sievers, Simon Dold, Thomas Keller, and Thorsten Klößner. You made the last five years a wonderful time. A special thanks goes to Augusto B. Corrêa, Clemens Büchner, Julian Bitterwolf, Simon Dold, and Thomas Keller for proofreading parts of this thesis. Furthermore, I especially thank Paulina Staus for proofreading the whole thesis on an extremely tight schedule.

I also thank all my friends from Trier (and around) Christian Stahl, Jan Bittner, Johannes Aubart, Johannes Krupp, Lisa Marquenie, Lukas Faber, Moritz Andres, Philipp Schon, and Yannic Klink. Time moved one, but we are still meeting and forgetting our age. Furthermore, I thank all my friends in Freiburg, especially, Anna Hudek, Dorothee Derwort, Isis Wolf, Jana Neuber, Luca Rühl-Muth, and Markus Gern. You make Freiburg a special place.

I would like to express my gratitude to my family. To my parents, Alice and Michael, who always supported me and were interested in my research. Now you can finally read up a selection of things I worked on. To my sister with whom I share so many memories and who tried to make difficult times lighter. Last but not least, to my wife Paulina. Without you, I would not have finished this thesis. Thank you for all your support and love. Thank you for being there, always.

Thank you!

Contents

1. Introduction

Planning is the art of finding sequences of actions which transform the current state of a problem into a state which satisfies some desired properties. For this purpose, planning is concerned with modelling real world problems with appropriate formalisms and finding solutions for these task. A planning task describes the possible states of the problem and the actions to move from one state to another state. We can frame many real world problems as planning tasks. Examples are satellite controls, elevator schedules, road navigation, and warehouse automation.

Over the long history of planning (McCarthy, 1958; Newell and Simon, 1963; Fikes and Nilsson, 1971) many formalisms to model problems were devised. A planning task can model deterministic and non-deterministic effects. It can model full, as well as, partial knowledge about the world. It can model single and multi-agent problems. It can model tasks with instantaneous and with durative effects, and much more. This versatility makes planning powerful, but makes it also difficult to solve arbitrary planning tasks. We restrict ourselves to *classical planning*. That means, we have full knowledge of the world, our actions are deterministic, there is a single agent executing the actions, and we have a condition, which describes the goal states. But even classical planning is EXPSPACE-complete (Erol, Nau, and Subrahmanian, 1995).

We have complete algorithms for classical planning, i.e., we have algorithms that find a solution if one exists and otherwise tell us if no solution exists. But these algorithms can take eons until they terminate. Researchers actively develop new algorithms, so called *planners*, to solve common planning tasks faster. On many tasks, today's algorithms are much faster than humans. But there are still many tasks, which they cannot solve within an acceptable time frame.

Predictive machine learning is the art of identifying patterns in data and using them to make predictions. Many machine learning techniques are trained on a set of observation - effect pairs. Afterwards, they are used to predict the effect of a new, unseen observation. Machine learning also has a long history. There exists a rich set of well established techniques to predict the future using the past, e.g., linear regression (Galton, 1886) and decision trees (Breiman et al., 1984). In recent years, neural networks made huge progress. Two decades ago, neural networks deciphered handwritten digits (LeCun et al., 1998). Today, they recognize images (Russakovsky et al., 2015), synthesize natural language (Shen et al., 2018), communicate in a dialog (Serban et al., 2016), and recently convert text to visually appealing images (Ramesh et al., 2021).

These advances kindled the idea of incorporating machine learning to improve our planning algorithms. Roberts and Howe (2009) predict for a set of planners their run-

times on a task. Then, they pick and execute the best predicted planner for that task. Many planners are guided by a heuristic function. Arfaee, Zilles, and Holte (2011) trained a simple neural network to combine multiple heuristic functions into one more powerful heuristic function. The showdown between AlphaGo (Silver et al., 2016) and the 9. Dan Go master Lee Sedol in 2016 showed that neural networks itself can even be trained as heuristic function for incredible difficult tasks using only the state as input. This sparked a new rush on using machine learning and especially deep learning in planning (Toyer et al., 2018; Agostinelli et al., 2019; Sievers et al., 2019a; Ma et al., 2020; Shen, Trevizan, and Thiébaux, 2020; Ferber, Helmert, and Hoffmann, 2020; Rivlin, Hazan, and Karpas, 2020; Yu, Kuroiwa, and Fukunaga, 2020; Karia and Srivastava, 2021; Speck et al., 2021; Nir, Shleyfman, and Karpas, 2021; Sudry and Karpas, 2022; Ferber and Seipp, 2022; Ferber et al., 2022a,b; Bhatia et al., 2022).

In this thesis, we delve into three directions to improve planning with machine learning. Many successful planners use heuristic search. It is commonly assumed that heuristic search with a better informed heuristic finds a solution faster than heuristic search with a worse informed heuristic. One key ingredient of AlphaGo (Silver et al., 2016) is a neural network which is trained to become a good heuristic for a specific task. Inspired by the successes of AlphaGo, we asked ourselves: *Can we autonomously train neural networks as powerful and competitive heuristics for arbitrary classical planning state spaces?* In Part I, we train feed-forward networks as competitive heuristics. The networks receive as input just a description of the current state. A key difference to Silver et al. (2016) is that we do not have a large team and incredible computational resources to train a heuristic for a single task. Instead, our framework runs almost autonomously while using a reasonable amount of computational power and still trains good heuristics. We evaluate parameter choices, identify difficulties, and present solutions. Furthermore, we compare our frameworks against another recent learning-based heuristic and against state-of-the-art model-based planners on tasks of interesting difficulties. We observe that learning-based planners are highly complementary and that the model-based planners still outperform *all* learning-based planners. There is a sole exception, our second framework performs best in one domain. Up to now, it was *only once* reported for *a single* domain that a learning based method exceeded model-based methods Karia and Srivastava (2021).

Not only learning-based heuristics, but also model-based planners have complementary strengths. It is not obvious, which planner to choose for a task. A planner portfolio is a schedule of planners which is executed to solve a task. In Part II, we train neural networks as online portfolios, i.e., the schedule is constructed specifically for the task to solve. Until now, all online portfolios use handcrafted input features. We present new online portfolios, which do not rely on handcrafted features. In our first step, we convert the given task to an image. A neural network identifies by itself important features. In our second step, we encode the task as graph and fed it to a graph neural network. This allows the neural network to learn arbitrary features of the task. Until today, no online portfolio considers for selection of the i-th planner in their schedule the

previous $i - 1$ unsuccessfully executed planners, although, this knowledge is essential to disregard planners which are similar to already executed planners. Finally, we show that switch to a second planner based on the knowledge that the first selected planner did not terminate successfully after half the time limit is beneficial. Each step improves the state of the art. In our last phase, we revisit *simple* machine learning methods and *simple* handcrafted features and show that they can learn online portfolios comparable to the current state of the art. We show how to inspect the explainable portfolios to increase our trust in their choices.

Even if we select the best planner for a task, the planner might be too slow to solve the tasks within a given time limit. Thus, we have to improve it. Every task induces a state space. In combination with a heuristic, this state space forms a topology (Hoffmann, 2005). If we know the topology for a heuristic during search, we can exploit it in many ways. Unfortunately, this knowledge is only available *after* the search. In Part III, we build upon the theory of Heusner (2019) and our expectation that related tasks share similarities in their topologies. We use description logic to learn formulas, which describe progress states, a feature of the topology. We show that these formulas generalize to other tasks from the same domain and how to use them during search. Furthermore, we present our current work, which learns for similar tasks how to split them in a set of subtasks. Therefore, we learn from a single simple task a graph structure which we call generalized bench transition system. We demonstrate how this could be used to solve a task by a sequence of simpler searches.

Our journey started in the middle of the hype around neural networks. We considered neural networks in each of our research directions. But, just because a technique is hyped, it does not mean that it is the right tool. In each of our research directions we use or considered using neural networks, and we evaluate how well other machine learning techniques perform. In Part III, we disregard neural networks completely and use decision trees, because neural networks could not provide the explanations we seeked.

1.1. Structure

In the first four chapters we explain the necessary background. Chapter 2 introduces our mathematical notation. Chapter 3 provides an introduction to classical planning, to the formalisms we use, and to the most important techniques we apply. Chapter 4 provides a short introduction to machine learning and presents an overview of the methods we use, especially of neural networks. Chapter 5 introduces description logic and presents a way to construct description logic interpretations for planning tasks. This chapter is mostly necessary for Part III.

Afterwards, the three main parts follow: Part I Learning Heuristics, Part II Learning Portfolios, Part III Learning State Space Topologies. Each part has the same structure. We introduce the topic and explain the necessary background for that part. Then we present our methods, followed by an experimental evaluation of our methods. We con-

clude every part with a summary of the results and describe future, as well as already started, avenues for research.

Chapter 19 concludes the thesis. Here, we provide a last high-level summary of the main contributions.

1.2. Experimental Setup

Some parameters stay almost unchanged over all experiments. We implemented all methods to sample states of a task, as well as, all search algorithms in Fast Downward (Helmert, 2006) using native C++ code. During search, the neural networks and description logic concepts and roles are evaluated in Fast Downward using the officially C/C++ libraries. Neural Fast Downward, our extension of Fast Downward, is publicly available[1]. It implements procedures to sample states or the whole states space of a planning task. It supports the machine learning frameworks Tensorflow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019). Within each part of this thesis, we use the same version Fast Downward.

We use Downward Lab (Seipp et al., 2017) to manage our experiments and to consolidate the results. Unless otherwise noted, we ran all experiments on an Intel Xeon E5-2660 CPU with 2.2GHz. We execute a single search for at most 30 minutes and with 3.8 GB of memory. As most planning algorithms use exactly one core, we use a single CPU core in all search runs. Many machine learning models, especially neural networks, profit from parallel computing power. Using two CPU cores during search, speeds up the searches with neural networks by 50%, i.e., the search expands 50% more states within the same time. Using four cores causes a further speed-up of 20% (Ferber, Helmert, and Hoffmann, 2020). A GPU could further accelerate the model evaluation.

We performed all computations on the same hardware, unless otherwise stated, but with different resource limits. We trained all neural networks with Tensorflow. We mainly use the Keras (Chollet, 2015) front-end for describing the network architectures. To train all other models, we used SKLearn (Pedregosa et al., 2011).

We mainly evaluate our approaches on the tasks from the International Planning Competitions (IPC). All IPC tasks are publicly available[2]. We indicate whenever we use additional tasks. All tasks are specified using the *Planning Domain Description Language* (McDermott et al., 1998, $PDDL$). If necessary, we compile them to a FDR (Bäckström and Nebel, 1995) representation using the translator of Fast Downward.

The code for all the experiments in this thesis, except for the unpublished ongoing work, is publicly available. We refer the interested reader to the publication list.

[1] `https://github.com/PatrickFerber/neuralfastdownward`
[2] `https://github.com/aibasel/downward-benchmarks`

1.3. Publications

The results in this thesis are based on my following publications. Publications marked by * were performed during an internship at IBM Research.

Part I: Learning Heuristics

- Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In De Giacomo, G., ed., *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, 2346–2353. IOS Press
  Code: `https://zenodo.org/record/3671553`
  Models: `https://zenodo.org/record/4000991`

- Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022b. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 583–587. AAAI Press
  Code & data: `https://zenodo.org/record/6303621`

Part II: Learning Portfolios

- *Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019a. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7715–7723. AAAI Press
  Code: `https://zenodo.org/record/6683999`

- Ferber, P.; Ma, T.; Huo, S.; Chen, J.; and Katz, M. 2019. IPC: A Benchmark Data Set for Learning with Graph-Structured Data. In *Proceedings of the ICML-2019 Workshop on Learning and Reasoning with Graph-Structured Representations*

- *Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 5077–5084. AAAI Press
  Code: `https://github.com/matenure/GNN_planner`

- Ferber, P.; and Seipp, J. 2022. Explainable Planner Selection for Classical Planning. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9741–9749. AAAI Press
  Code & data: `https://zenodo.org/record/5767692`

Part III: Learning State Space Topology

- Ferber, P.; Cohen, L.; Seipp, J.; and Keller, T. 2022a. Learning and Exploiting Progress States in Greedy Best-First Search. In De Raedt, L., ed., *Proceedings of the 31th International Joint Conference on Artificial Intelligence (IJCAI 2022)*, 4740–4746. IJCAI
  Code & data: `https://zenodo.org/record/6496716`

Results of my publications below were not included in the thesis. Workshop papers, which are superseded by a conference paper are not listed.

- Ferber, P. 2020. Simplified Planner Selection. In *ICAPS 2020 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 102–110
  Code & data: `https://zenodo.org/record/4061614`
- Büchner, C.; Ferber, P.; Seipp, J.; and Helmert, M. 2022. A Comparison of Abstraction Heuristics for Rubik's Cube. In *ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*
  Code & data: `https://zenodo.org/record/6589227`
- Steinmetz, M.; Fišer, D.; Eniser, H. F.; Ferber, P.; Gros, T.; Heim, P.; Höller, D.; Schuler, X.; Wüstholz, V.; Christakis, M.; and Hoffmann, J. 2022a. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 353–361. AAAI Press
  Code: `https://zenodo.org/record/6323289`
- Heller, D.; Ferber, P.; Bitterwolf, J.; Hein, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions: Taking Confidence into Account. In *Proceedings of the 15th Annual Symposium on Combinatorial Search (SoCS 2022)*, 223–228. AAAI Press
  Code: `https://zenodo.org/record/6553255`

1.4. Contribution to Multi-Author Papers

In our research it is common to work as a team. Below I indicate *my* contributions for the publications used for this thesis. I only add contributions without my involvement, if incorrect expectations are possible. *Not applicable (NA)* means this contribution required no work, e.g., we took it from a previous publication; *None* means I did not contribute to this part; *Minor* means I contributed minor parts, but someone else did most of the work; *Moderate* means I contributed an important portion of the work, but someone else also contributed an important portion; *Major* means I contributed most of the work. In general, all authors wrote together the paper.

Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space

Major	Training data generation
Major	Training neural network heuristic functions
Major	Evaluating neural network heuristic functions
Moderate	Writing and editing the publication

Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods.

Major	New Bootstrapping Approach
Major	Evaluating and comparing bootstrapping approach and baselines - Training data generation - Training
Moderate	Adapting supervised learning from Ferber, Helmert, and Hoffmann (2020) as baseline - Training data generation - Training
Moderate	Writing and editing the publication
Minor	Adapting STRIPS-HGN (Shen, Trevizan, and Thiébaux, 2020) as baseline - Training data generation - Training

Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection

Major	Exploring architecture and training variants and evaluating robustness
Moderate	Writing and editing the publication
Minor	Training data generation
None	Original NN architecture and training (Katz et al., 2018)

IPC: A Benchmark Data Set for Learning with Graph-Structured Data

Major	Generating Statistics
Moderate	Interpreting Statistics
Moderate	Writing and editing the publication
NA	Generating Graphs

Online Planner Selection with Graph Neural Networks and Adaptive Scheduling

Major	Adaptive scheduling approach
Moderate	Training data preparation
Moderate	Evaluating the GNNs
Moderate	Writing and editing the publication
None	Training GNNs

Explainable Planner Selection for Classical Planning

Major	Training data generation
Major	Training
Major	Evaluation
Moderate	Writing and editing the publication

Learning and Exploiting Progress States in Greedy Best-First Search

Major	Learning Formulas
Moderate	Generating Features
Moderate	Evaluating Formulas
Moderate	Writing and editing the publication
Minor	Handcrafted Formulas
None	Generating States with Labels

Unpublished Work: Learning Generalized Bench Transition Systems

Major	Constructing Generalized Bench Transition System
Major	Visualizations Generalized Bench Transition System
Moderate	Bench Walking
Moderate	Writing and editing the publication
None	Sampling Bench Transition System

2. Mathematical Notation

To pre-empt confusion, we clarify our mathematical notation and some basic concepts. This chapter is *not* an introduction to these concepts.

We represent a set as $\{x_1, x_2, \ldots, x_i\}$ and a multi-set as $\{\{x_1, x_2, \ldots, x_i\}\}$. In contrast to a set, a multi-set can contain duplicates. Sets and multi-sets have no order. We write $\langle x_1, x_2, \ldots, x_i \rangle$ to denote a tuple, an ordered collection of elements.

For scalar variables, we use lower case Latin letters, e.g., a, b. We annotate vectors with a small arrow, e.g., $\vec{a}$. For matrices, we use upper case Latin letters, e.g., A, B. To access the i-th element of a vector or tuple, we subscript its variable with i, e.g., $\vec{x}_i$. Let X be a 2-dimensional matrix. To access its i-th row, we write X_i. To access the element in the i-th row and j-th column of X, we write $X_{i,j}$. We define the content of a vector as $\vec{x} = [x_1\ x_2\ \ldots\ x_i]$. If the content of a vector follows a clear pattern, we may write $\vec{x} = [x_k]_{k=1,\ldots,i}$. We use the same abbreviations for defining tuples. By default, all our vectors are column vectors. If clear from context, we omit the transpose symbol for readability. Let $A \in \mathbb{R}^{N \times M}$ and $B \in \mathbb{R}^{N' \times M}$ be two matrices. Then $X = [A\ B]$ is a matrix with $N + N'$ rows and M columns.

The set of natural numbers ($\mathbb{N}$) excludes zero. Any set of positive numbers (e.g., $\mathbb{R}^+$) excludes zero. If we include zero, then we subscript the set with zero. We sometimes use the Boolean values *true* and *false* in numerical expressions. In these cases, we implicitly convert *true* to 1 and *false* to 0. We define $\mathbb{B}$ as the set of Boolean values.

First-Order Logic. We use formalisms based on *first-order* logic (FOL), also called *predicate logic*. The *signature* of a FOL is a tuple $\mathfrak{S} = \langle \mathfrak{V}, \mathfrak{C}, \mathfrak{F}, \mathcal{P} \rangle$ where

- $\mathfrak{V}$ is a set of variable symbols,
- $\mathfrak{C}$ is a set of constant symbols,
- $\mathfrak{F}$ is a set of function symbols, and
- $\mathcal{P}$ is a set of predicate symbols.

Every function symbol $f \in \mathfrak{F}$ has an arity $arity(f) \in \mathbb{N}_0$. Every predicate symbol $p \in \mathcal{P}$ has as well an arity $arity(p) \in \mathbb{N}_0$. A *term* is a constant symbol, a variable symbol, or $f(t_1, \ldots, t_k)$ where $f \in \mathfrak{F}$ is a function symbol with $arity(f) = k$ and all t_1 to t_k are terms. Let $p \in \mathcal{P}$ be a predicate symbol with $arity(p) = k$ and $t_1, \ldots, t_k$ be terms. Then $p(t_1, \ldots, t_k)$ is a *predicate*. A *grounded predicate* is a predicate, which

is constructed without variable symbols. We call a grounded predicate also a *fact*. The following structures are all predicates: a formula over $\mathfrak{S}$, the identity of two terms (e.g., $t_1 = t_2$), the universal or existential quantification of a variable $x \in \mathfrak{V}$ for another formula (e.g., $\forall x : \phi$), the negation of a formula, the conjunction of a formula, and the disjunction of a formula.

An *interpretation* $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ for $\mathfrak{S}$ makes a formula interpretable. The universe $\Delta^{\mathcal{I}}$ is a non-empty set. $\cdot^{\mathcal{I}}$ is a function, which assigns all constants $c \in \mathfrak{C}$ an element of the universe $c^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, defines all functions $f \in \mathfrak{F}$ with $arity(f) = k$ as $f : \Delta^{\mathcal{I}k} \to \Delta^{\mathcal{I}}$, and defines which ground predicates hold. A variable assignment is a function, which maps variables to objects in the universe. A complete variable assignment is a function $\mathbb{V} : \mathfrak{V} \to \Delta^{\mathcal{I}}$ which maps every variable to an element of the universe. We use a set notation to represent variable assignments. Let $\mathbb{V}$ be a variable assignment over the variables V. We represent $\mathbb{V}$ as a set $\{v \to \mathbb{V}(v) \mid v \in V\}$. Furthermore, we define $vars(\mathbb{V}) = \{v \mid \exists d : v \to d \in \mathbb{V}\}$. We overload this function for predicates. Let $p(t_1, \ldots, t_k)$ be a predicate. $vars(p(t_1, \ldots, t_k))$ is the set of variables occurring in the terms which define the predicate.

3. Classical Planning

Planning is about finding sequences of actions, which solve a task. Given an arbitrary planning task, a planning algorithm, so called *planner*, returns a sequence of actions, which transforms the initial state of the world as described by the tasks into a state, which satisfies a goal condition. The formalisms to define planning tasks describe a rich class of problems. Thus, we call a planner also a general purpose solver. The formalism, which describes a problem, restricts the features that can be modeled. In this thesis, we consider the feature set of *classical planning*. This means, all our planning tasks have the following properties:

- *single agent*, i.e., there is a single controller who executes a sequence of actions. This controller does not need to be a concrete entity with a location. It is forbidden to have multiple controllers, which execute independent sequences of actions.
- *no exogenous events*, i.e., nothing except for the actions applied by the single controller changes the world.
- *fully observable*, i.e., at all points, the controller knows everything about the state of the world and all existing actions.
- *deterministic*, i.e., applying the same action in the same state leads always to the same known successor state.
- *instantaneous*, i.e., executing an action applies all effects of the action immediately. For a real world problem, if suffices if the next action is applied *after* all effects of the previous action took effect.

Furthermore, all planning formalisms describe an initial state of the problem, a goal condition, which identifies states we want to reach, and a set of actions to transform states. We try to find a sequence of actions, which transforms some initial state of the world into a state which satisfies some goal condition. We name the real world challenge a *problem*. We call its representation in a planning formalism a *task*. Related work uses the word *model* as synonym for a planning task. We avoid this to prevent confusion with machine learning models. In this thesis, a model *always* describes a machine learning model, i.e., a trained or untrained concrete instantiation of a machine learning technique.

3.1. Planning Formalisms

There exist many formalisms to model classical planning tasks. We use $PDDL$ (McDermott et al., 1998) and FDR. The latter extends SAS^+ Bäckström and Nebel (1995) with support for conditional effects. The $PDDL$ formalism is based on first-order logic (see Section 2, First-Order Logic). We call formalisms based on first-order logic *lifted*.

Definition 3.1. PDDL Planning Task
A $PDDL$ task is a tuple $\Pi_{PDDL} = \langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_\text{I}, \delta \rangle$ where

- $\mathcal{O} = \{o_1, \ldots, o_n\}$ *is a finite set of objects, which represent the universe.*
- $\mathcal{P} = \{p_1, \ldots, p_n\}$ *is a finite set of* predicate symbols. *Let $p(t_1, \ldots, t_k)$ be a predicate where t_i is an object or variable. If all terms are objects ($t_1, \ldots, t_k \in \mathcal{O}$), then it is a* grounded predicate *or just fact. If at least one term is a variable, then it is a* lifted predicate. *Let $p(t_1, \ldots, t_k)$ be a lifted predicate. Let $\mathbb{V}$ be a variable assignment over the variables $vars(p(t_1, \ldots, t_k))$. We can ground $p(t_1, \ldots, t_k)$ using $\mathbb{V}$ by replacing all terms $t_i \in vars(p(t_1, \ldots, t_k))$ with $\mathbb{V}(t_i)$. F is the set of all facts. A* literal *is a fact or a negated fact. L is the set of all literals. A set $s \subseteq F$ is a state. $\mathcal{S} = 2^F$ is the set of all states. We sometimes extend s to include explicitly the remaining facts as negative literals $[\![s]\!] = s \cup \{\neg f \mid f \in F \wedge f \notin s\}$.*
- *$\mathcal{A}$ is a set of* action schemas. *An action schema $a \in \mathcal{A}$ is a tuple $\langle V_a, pre_a, \mathit{eff}_a \rangle$. V_a is a set of variables. pre_a is a set of possibly negated predicates called* precondition *of the action. eff_a is the* effect *of the action. The effect eff_a is a list of conditional effects $V_a^i, cond_a^i \triangleright \mathit{eff}_a^i$. $cond_a^i$ is a set of possibly negated predicates and called* effect condition*, eff_a^i is a single possibly negated predicated and is called* effect predicate*. V_a^i is an additional set of variables with $V_a^i \cap V_a = \emptyset$ and $V_a \cup V_a^i \supseteq vars(\mathit{eff}_a^i) \cup \bigcup_{l \in cond_a^i} vars(l)$. Given a variable assignment $\mathbb{V}$ for V_a, pre can be grounded to $pre_{a,\mathbb{V}}$. Given an additional assignment $\mathbb{V}_a^i$, $cond_a^i$ and eff_a^i can be grounded to $cond_{a,\mathbb{V},\mathbb{V}_a^i}^i$ and $\mathit{eff}_{a,\mathbb{V},\mathbb{V}_a^i}^i$.*

  A ground action or just action *$a_\mathbb{V}$ is an action schema a which is partially grounded with a variable assignment V_a over $\mathcal{O}$. The* action *$a_\mathbb{V}$, is applicable in a state $s \in \mathcal{S}$, written as $applicable(s, a_\mathbb{V})$, if $pre_{a,\mathbb{V}} \subseteq [\![s]\!]$. If $a_\mathbb{V}$ is applicable in s, then applying it, written as $s[\![a_\mathbb{V}]\!]$, leads to a new state (subscript a is omitted for readability)*

$$
\begin{aligned}
s[\![a_\mathbb{V}]\!] = \{f \mid & f \in s, \neg\exists \langle V^i : cond_\mathbb{V}^i \triangleright \mathit{eff}_\mathbb{V}^i \rangle \in \mathit{eff}^\mathbb{V}, \mathbb{V}^i : \\
& cond_{\mathbb{V},\mathbb{V}^i}^i \subseteq [\![s]\!] \wedge \neg f = \mathit{eff}_{\mathbb{V},\mathbb{V}^i}^i\} \cup \\
\{f \mid & \langle V^i : cond_\mathbb{V}^i \triangleright \mathit{eff}_\mathbb{V}^i \rangle \in \mathit{eff}_\mathbb{V}, \exists \mathbb{V}^i : \\
& cond_{\mathbb{V},\mathbb{V}^i}^i \subseteq [\![s]\!] \wedge \mathit{eff}_{\mathbb{V},\mathbb{V}^i}^i \in F\}.
\end{aligned}
$$

The first set describes all facts which were previously in the state and are not removed by the action. The second set describes all facts which are added by the action. If a ground action has an effect which removes a fact and it has another effect which adds a fact, then we follow the add-after-delete *convention, i.e., the deletion is ignored. Additionally, every ground $a_{\mathbb{V}}$ is associated with a non-negative* cost $cost(a_{\mathbb{V}}) \in \mathbb{R}_0^+$*, such that for any two variable assignments $\mathbb{V}', \mathbb{V}''$ we have $cost(a'_{\mathbb{V}}) = cost(a''_{\mathbb{V}})$.*

- $s_{\mathrm{I}} \in \mathcal{S}$ *is the* initial state.
- $\delta \subseteq L$ *is a set of literals called* goal. *Any state $[\![s]\!] \supseteq \delta$ is a* goal state. S_G *is the set of all goal states.*

If two tasks use the same predicate symbols and the same action schemas, then we say that they belong to the same *domain*.

The FDR formalisms is based on predicate logic. We call such formalisms grounded. Both formalisms are related. Sometimes, they use the same notation, but with a slightly different meaning. Where necessary, we indicate which meaning we use.

Definition 3.2. FDR Planning Task
An FDR task is a tuple $\Pi_{FDR} = \langle \mathcal{V}, A, s_{\mathrm{I}}, \delta \rangle$ where

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is a finite set of state variables. Every state variable v has a finite domain $dom(v)$. A pair $\langle v, d \rangle$ with $v \in \mathcal{V}$ and $d \in dom(v)$ is a* fact. *As before, F is the set of all facts. A* state *s is a variable assignment over $\mathcal{V}$, i.e., $s = \{v_1 \to d_1, \ldots, v_n \to d_n \mid d_i \in dom(v_i)\}$. A partial state is a variable assignment over a subset of $\mathcal{V}$. $\mathcal{S}$ is the set of all states.*
- *A is a finite set of ground actions. An action $a \in A$ is a tuple $\langle pre_a, \mathit{eff}_a \rangle$ where the* precondition *pre_a is a partial variable assignment and the* effect *$\mathit{eff}_a = [cond_a^1 \triangleright \mathit{eff}_a^1, \ldots, cond_a^n \triangleright \mathit{eff}_a^n]$ is a list of conditional effects. The* effect condition *$cond_a^i$ is a partial variable assignment, and eff_a^i is a fact.*

  An action a is applicable in a state s if $pre_a \subseteq s$. If a is applicable in s, then applying a in s, written as $s[\![a]\!]$, leads to a new state

$$\begin{aligned} s[\![a]\!] = &\{v \to d \mid v \to d \in s, \neg\exists cond_a^i \triangleright \mathit{eff}_a^i \in \mathit{eff}_a : \\ &\qquad cond_a^i \subseteq s \wedge \mathit{eff}_a^i = v \to d' \wedge d \neq d'\} \cup \\ &\{\mathit{eff}_a^i \mid cond_a^i \triangleright \mathit{eff}_a^i \in \mathit{eff}_a, cond_a^i \subseteq s\}. \end{aligned}$$

  The first set describes the facts which stay in the state, the second set describes the facts which are new. We assume conflict free actions, i.e., for every state and action it never happens that two conditional effects apply simultaneously and set

different values for the same variable. More formally, for every state $s \in \mathcal{S}$ and action $a \in A$ holds:

$$\forall cond' \triangleright v \to d', cond'' \triangleright v \to d'' \in \mathit{eff}_a : d' = d'' \vee cond' \not\subseteq s \vee cond'' \not\subseteq s$$

Every action a is associated with a non-negative cost $cost(a) \in \mathbb{R}_0^+$.

- s_{I} *is the* initial state.
- δ *is a partial state, which describes the goal. Any state $s \supseteq \delta$ is a* goal state. S_G *is the set of all goal states.*

If all actions of a task have only empty effect conditions, then we say that it has no conditional effects. We can compile a $PDDL$ tasks into an FDR task with up to an exponential increase in the size of the task representation (Helmert, 2009). In practice, many planners apply this compilation and then use the FDR task. The $PDDL$ and FDR tasks share some properties. Applying an action a to a state s as defined by $s[\![a]\!]$ is called *progression*. The state $s[\![a]\!]$ is a *successor* of s. The function $succ : \mathcal{S} \mapsto 2^{\mathcal{S}}$ maps states to their successors, i.e., $succ(s) = \{s[\![a]\!] \mid a \in A \wedge applicable(s, a)\}$. Given two states s and s', then s' is *reachable* from s if there exists a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ such that every a_i is applicable in $s[\![a_1]\!] \ldots [\![a_{i-1}]\!]$ and $s[\![a_1]\!] \ldots [\![a_n]\!] = s[\![\pi]\!] = s'$. For a state s, a sequence of actions π is an *s-plan* if the last state along π is a goal state, i.e., $s[\![\pi]\!] \in \mathrm{S}_G$. Let Π be a task with initial state s_{I}. For simplicity, we call an s_{I}-plan just a *plan*. An action sequence π has a non-negative cost $cost(\pi) = \sum_{a \in \pi} cost(a)$. An optimal plan is a plan of minimal cost. For the sake of simplicity, we ignore the notion of *derived predicates*.

3.2. Heuristics

A *heuristic* is a state-value function, i.e., a function which assigns every state to a value. In planning, heuristics commonly estimate the cost or length of a plan for a state. For this endeavor, they often solve a simplified version of the task (Korf, 1997; Edelkamp, 2001; Hoffmann and Nebel, 2001; Dräger, Finkbeiner, and Podelski, 2006; Helmert, Haslum, and Hoffmann, 2007; Katz, Hoffmann, and Domshlak, 2013; Seipp and Helmert, 2013).

Definition 3.3. Heuristic
Let Π be a planning task with a set of states $\mathcal{S}$. A heuristic for the task Π is a function $h : \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$.

The *perfect heuristic* h^* is a special heuristic. It assigns every state s the cost of its optimal plan. If no such plan exists, it assigns infinity. It is impractical to use the perfect heuristic during search, as every state estimation is as expensive as finding an optimal

plan. Another important heuristic is the perfect delete-relaxed heuristic (Hoffmann and Nebel, 2001, h^+). Let Π be a PDDL task. Let Π' be a transformation of Π where all occurrences of negated facts are removed. The h^+ value for a state s of Π is the h^* value for the same state s of Π'. Some important properties of heuristics are defined below.

Definition 3.4. Properties of Heuristics
Let Π be a planning task with states $\mathcal{S}$ and goal states S_G. A heuristic $h : \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ is

- safe *if* $h(s) = \infty \implies h^*(s) = \infty$ *for all states* $s \in \mathcal{S}$.
- goal-aware *if* $h(s) = 0$ *for all goal states* $s \in \mathrm{S}_G$.
- admissible *if* $h(s) \leq h^*(s)$ *for all states* $s \in \mathcal{S}$.
- consistent *if* $h(s) \leq cost(a) + h(s')$ *for all transitions* $s' = s[\![a]\!]$.

3.3. State Spaces

Algorithm 1 Best-First Search (BFS) without (with) Reopening. For a planning task Π, an evaluation function f, and a heuristic function h, it returns a plan or *unsolvable*. BFS is complete, if h is safe. $g(n)$ is the cost of the cheapest known path to the state of n.

```
1:  procedure BEST-FIRST SEARCH
2:      open ← Priority Queue ordered by ⟨f, h⟩
3:      if h(s_I) < ∞ then
4:          open.insert(⟨none, none, s_I⟩)
5:      closed ← {} or new HashMap<>
6:      while not open.isEmpty() do
7:          n ← open.pop_min()
8:          _, _, s ← n
9:          if s ∉ closed ∨ g(n) < closed[n[2]] then
10:             closed.insert(s) or closed[n[2]] = g(n)
11:             if is_goal(s) then
12:                 return trace_path(n)
13:             for all ⟨a, s'⟩ ∈ succ(s) do
14:                 if h(s') < ∞ then
15:                     open.insert(⟨n, a, s'⟩)
16:     return unsolvable
```

A state space is a graph structure over a set of states, which indicates an initial state and a set of goal states.

Definition 3.5. State Space
Let $\mathcal{S}$ be a set of states, $s_{\mathrm{I}} \in \mathcal{S}$ be an initial state, $\mathrm{S}_G \subseteq \mathcal{S}$ be a set of goal states, and $succ : \mathcal{S} \mapsto 2^{\mathcal{S}}$ be a function, which represents the relations between states. A state space *is a tuple $\mathcal{X} = \langle \mathcal{S}, s_{\mathrm{I}}, \mathrm{S}_G, succ \rangle$.*

Any planning task Π with states $\mathcal{S}$, initial state s_{I}, goal states S_G, and successor function $succ$ induces a state space $\mathcal{X}(\Pi) = \langle \mathcal{S}, s_{\mathrm{I}}, \mathrm{S}_G, succ \rangle$.

A common tool to solve planning tasks is forward state space search, also called forward search or progression search. One group of state space search algorithms is *best-first search*. Best-first search has a priority queue, called *open list*, of reachable states, which is initialized with the initial state s_{I}. It sorts the open list by an *evaluation function* f. Iteratively, it pops the state s with minimal f value from the open list. Best-first search without reopening expands the state s, if it was not yet expanded. Best-first search with reopening expands the state s, if it was not yet expanded *or* a new cheaper path to reach s was found. If the popped state s is a goal state, the search ends and returns the cheapest known sequence of actions π which led to the goal state s. If it is not a goal state, then the search *expands* the state s. That means it constructs all successors of the state and adds them to the open list (see Algorithm 1). We use two common instantiations of best-first search: *Greedy best-first search* (GBFS) and *A* search*.

Definition 3.6. Greedy Best-First Search (Doran and Michie, 1966)
Let h be an arbitrary heuristic function. Greedy best-first search is a best-first search without reopening with h as heuristic and *as evaluation function.*

Definition 3.7. A^* Search (Hart, Nilsson, and Raphael, 1968)
Let h be an arbitrary heuristic function and g be the function which returns for a search node n the cost of the cheapest known path to n. A^* search is a best-first search with reopening with h as heuristic and $f(n) = h(n.state) + g(n)$ as evaluation function.

Given an admissible and consistent heuristic h, A^* finds optimal plans. GBFS has no guarantee of the plan quality. In practice, it often finds plans faster. If we relate to A^* without a concrete heuristic, we assume the heuristic is admissible and consistent. We assume that all our heuristics are safe.

An alternative to forward search is backward search, also called regression search. In forward search, we start at the initial state and keep track of already reached states. We iteratively increase this set until we find a goal state. In backward search, we start at the goal condition, a partial state. We keep track of the already reached partial states. In every iteration, we take one partial state and reason from which other partial states we can reach it. We stop once we find a partial state p which includes initial state s_{I}, i.e., $p \supseteq s_{\mathrm{I}}$. The key component of this search is the regression operator, which performs the reasoning. We do not perform regression search, but we use the regression operator.

Definition 3.8. Regression Operator
Let $\Pi = \langle \mathcal{V}, A, s_{\mathrm{I}}, \delta \rangle$ be an FDR planning task without conditional effects. We simplify the effect of an action a to $\mathit{eff}'_a = \{\mathit{eff}^i_a \mid \emptyset \rhd \mathit{eff}^i_a \in \mathit{eff}_a\}$.

Let p be a partial state and let $a \in A$ be an action. p is regressable *over a if*

1. $\mathit{eff}'_a \cap p \neq \emptyset$,
2. $\neg\exists v \in \mathcal{V} : v \in \mathit{vars}(\mathit{eff}'_a) \cap \mathit{vars}(p) \wedge \mathit{eff}'_a(v) \neq p(v)$, *and*
3. $\neg\exists v \in \mathcal{V} : v \notin \mathit{vars}(\mathit{eff}'_a) \wedge v \in \mathit{vars}(\mathit{pre}_a) \cap \mathit{vars}(p) \wedge \mathit{pre}_a(v) \neq p(v)$.

If p is regressable over a, then its regression is

$$\mathit{regr}(p, a) = \{v \to p(v) \mid v \in \mathit{vars}(p), v \notin \mathit{vars}(\mathit{eff}'_a)\} \cup \mathit{pre}_a.$$

3.4. Metrics

To compare planners in our experiments, we use the *expansions*, *time*, and *coverage* metrics. Given a single planning task, the number of expansions expresses the number of states best-first search expands until it finds a goal state. Let the task have *unit-cost* actions, i.e., all actions have a cost of one. Then we assume that a better informed heuristic, causes GBFS to expand fewer states. GBFS with the *perfect heuristic*, the best informed heuristic possible, expands only the states along one optimal plan. This is also the minimum number of expansions possible with GBFS. To deduce the informedness of a heuristic from the expansions on a single task is too simplistic. The heuristic could be well-informed on most parts of the state space, but uninformed on the fraction required to solve the given task. To use the number of expansions as a proxy for the informedness, the heuristic should be evaluated on a set of diverse tasks.

Informedness is not the only important metric. Given two heuristics, the first one is better informed, and the second one is faster to evaluate. It is not obvious which heuristic we to prefer. For the less informed heuristic, the time required for additional expansions might exceed the time saved due to the faster evaluation. Thus, we also measure the *time* an algorithm takes to find a plan.

The blind heuristic predicts the same value for all states. It is the least informed, but also the fastest to evaluate. GBFS with the blind heuristic expands states quickly. For simple tasks, it quickly finds a solution. On harder tasks, it often fails, because GBFS keeps track of all reachable states and all expanded states. This quickly exceeds the available memory. In practice, we have to work with finite resources. Given a set of tasks and a resource limit, the *coverage* of a planner is the number of tasks solved within the limits.

4. Machine Learning

For humans, it is natural to observe an event, predict its effects and act on this knowledge. If we play football, we see how the ball moves, we predict where it will be in the future, and we know where to run. We can do this, because we already observed moving balls for uncountable times and learned how balls travel. But there are effects, especially in high-dimensional data, which we do not grasp that easily. Thus, we developed mathematical tools to identify patterns in data and to predict the effects of these patterns. The research area for this is *machine learning*. Most "tools" describe a model and a procedure to train the model on observations with known effects. Some models are trained almost purely on the data, others are trained additionally with a human provided bias. We use these models to predict the effects of new observations.

More formally, let O be the set of all possible observations. We define a list of input features, in short features, $\mathcal{F}$ to describe the observations. Every feature $f \in \mathcal{F}$ is a function $f : O \mapsto \mathbb{R}$ which extracts from the observation a scalar. We do *not* use the word feature to refer to intermediate outputs of a model. Instead, we call those hidden or latent representations. We do not use the term *effect*. We use the more common term *label*. $\mathcal{L}$ is the set of all possible labels. For an observation $o \in O$, we construct a feature vector $\vec{x}$ with $\vec{x}_i = \mathcal{F}_i(o)$. The label of o is a value $y \in \mathcal{L}$. A feature vector - label pair is a *sample*. The model m is a function $m : \mathbb{R}^{|\mathcal{F}|} \mapsto \mathcal{L}$ which predicts for a feature vector a label. If it predicts values from a discrete set of labels, then we call it a classification model. If it predicts continuous values, then we call it a regression model. This regression is independent of the previously defined regression operator. If it is unclear from the context, then we specify to which regression we refer.

In general, we train models on a list of samples $D = \langle\langle \vec{x}_i, y_i \rangle\rangle_{i=1..n} \approx \langle X, \vec{y} \rangle$ called *training data*. We call X the *feature matrix* with $X = [\vec{x}_i]^\top_{i=1..n}$ and $\vec{y}$ the label vector with $\vec{y} = [y_i]^\top_{i=1..m}$. To predict a label for an observation, we first construct the feature vector and then query the model.

4.1. Classical Models

Over time, many classes of models were developed. One of the earliest and simplest is *linear regression* (Galton, 1886). A linear regression model is a linear combination over the elements in the feature vector and a learned bias term. It outputs a continuous value, i.e., $\mathcal{L} = \mathbb{R}$. For each feature, it learns a weight, which represents the positive or negative impact of the feature on the prediction.

Definition 4.1. Linear Regression
Let $\mathcal{F}$ be a list of features, $\vec{x} \in \mathbb{R}^{|\mathcal{F}|}$ be the feature vector of an observation and m be a linear regression model with the weight vector $\vec{w} \in \mathbb{R}^{|\mathcal{F}|}$ and the bias $b \in \mathbb{R}$. Then the output of m is

$$m(\vec{x}) = \vec{w}^\top \vec{x} + b.$$

When training a linear regression model m, its weight vector and bias term are adapted to minimize a *loss function*. Most commonly the *mean squared error* (MSE) is minimized.

Definition 4.2. Mean Squared Error
Let $D = \langle X, \vec{y} \rangle$ be a data set with N samples. $X \in \mathbb{R}^{N \times M}$ is its feature matrix and $\vec{y} \in \mathbb{R}^N$ is its label vector. Let m be a function $m : \mathbb{R}^M \mapsto \mathbb{R}$. The mean squared error of m on D is

$$MSE(m, D) = \frac{1}{N} \sum_{i=1}^{N} (m(X_i) - \vec{y}_i)^2.$$

If a linear regression model minimizes the mean squared error, then there exists an analytical solution to find the optimal weight vector and bias term. Let $D = \langle X, \vec{y} \rangle$ be a data set with N samples. $X \in \mathbb{R}^{N \times M}$ is its feature matrix and $\vec{y} \in \mathbb{R}^N$ is its label vector. Let $c = [1]_{i=1..n}^\top$ be a vector and $X' = [X\ c]$ be the feature matrix concatenated with c. Then, we can calculate the optimal solution using

$$\vec{\beta} = (X^\top X)^{-1} X^\top \vec{y}.$$

The optimal weight vector is $\vec{w} = \vec{\beta}_{1..M}$ and the optimal bias term is $b = \vec{\beta}_{M+1}$.

It is expected that the models cannot perfectly capture the relation between observation and effect. Some reasons are that relevant features are missing, the effect is not deterministic, or the model is not able to learn the true underlying function. For example, linear regression can only predict linear combinations of the values in the feature vector. If the true function requires a quadratic transformation of a feature value, linear regression can never accurately learn this function. Nevertheless, the training process minimizes the loss as much as possible. As a consequence, the training process can use the weights of irrelevant features to minimize the loss on the training data. This is called *overfitting*. On a new data set, the overfitted weights have a detrimental effect. A technique to prevent overfitting is *regularization*. Regularization adapts the loss function to incorporate the weights and biases of a model. A common technique is $L1$ regularization, which adds the $L1$ norm of the weights to the MSE. (Tibshirani, 1996). The new loss function is

$$(MSE + L1)(\lambda, m, D) = \frac{1}{N} \sum_{i=1}^{N} (m(X_i) - \vec{y}_i)^2 + \lambda(|b| + \sum_{j=1}^{M} |w_j|).$$

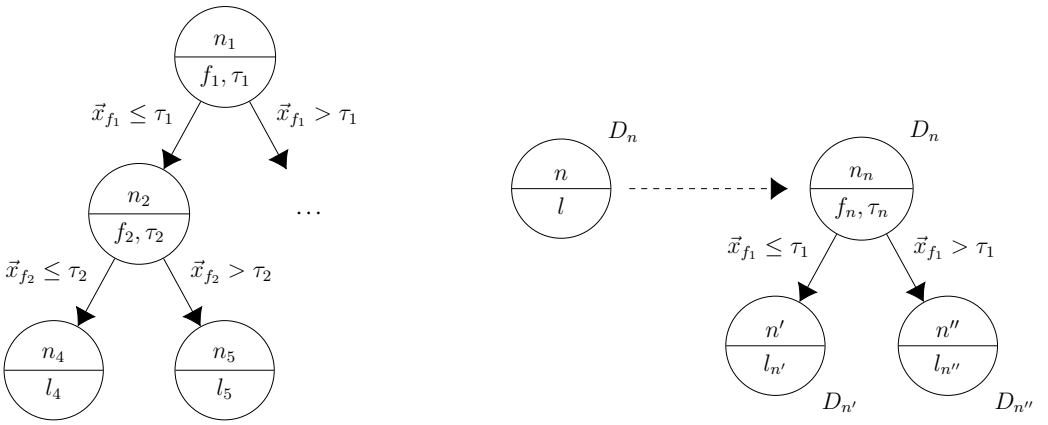


Figure 4.1: (Left) Visualization of a decision tree. (Right) Conversion from a leave node n to an internal node n_n.

Alternatively or additionally, $L2$ regularization can be used. $L2$ regularization adds the $L2$ norm of the weights to the loss function. The $L2$ norm squares the absolute value of each weight before summation. Both techniques add a trade-off to the optimization. Increasing the value of a weight parameter might reduce the MSE in the loss function, but increases the $L1$ and $L2$ penalty. $L1$ regularization removes unnecessary weights to zero and thus can be used for feature selection.

Another class of machine learning models are *decision trees* (Breiman et al., 1984). In contrast to linear regression, a decision tree does not predict a continuous value, but an element from a finite sets of labels.

Definition 4.3. Decision Tree
Let $\mathcal{F}$ be a set of features and $\mathcal{L}$ be a finite set of labels. A decision tree T is a binary tree with the internal nodes $internal(T)$, the leave nodes $leaves(T)$, $nodes(T) = internal(T) \cup leaves(T)$, and a root node $n_1 \in nodes(T)$. Every node $n \in internal(T)$ is associated with a feature $f_n \in \mathcal{F}$ and a threshold τ_n. Every node $n \in leaves(T)$ is associated with a label $l_n \in \mathcal{L}$.

To evaluate a decision tree (e.g., Figure 4.1) on a feature vector $\vec{x}$, we start at the root node n_1. If the current node n is a leave node, we predict the label l_n associated with n and terminate. If the current node n is an internal node and the value of the feature associated with the node is smaller than or equal to the threshold, i.e., $\vec{x}_{f_n} \leq \tau_n$, then we traverse the first child. Otherwise, we traverse the second child. Then, we repeat the procedure until we reach a leaf node.

We greedily construct a decision tree T for some data $D = \langle X, \vec{y} \rangle$. We associate every node $n \in nodes(T)$ with a non-exclusive subset of samples $D_n = \langle X_n, \vec{y}_n \rangle$ from D. Initially, the decision tree has only the root node n_1 which is a leaf node and is associated with the full training data D. We associate every leaf node $n \in leaves(T)$ with the most frequent label in $\vec{y}_n$. Now, we iteratively convert leaf nodes to internal

nodes (see Figure 4.1). We pick an arbitrary leaf node $n \in leaves(T)$ where $\vec{y}_n$ contains different labels. Then, we create two new nodes n' and n'' as children of n. n' and n'' are leave nodes, n is now an internal node. We associate a feature $f_n \in \mathcal{F}$ and a threshold $\tau_n \in \mathbb{R}$ with n and we construct two new data sets $D_{n'}$ and $D_{n''}$. $D_{n'}$ contains all samples from D_n with $X_{n,f_n} \leq \tau_n$. $D_{n''}$ contains all samples from D_n with $X_{n,f_n} > \tau_n$. We stop converting leaf nodes to internal nodes, once all leaves are associated with a single label.

The final question is *how to pick a feature f_n and threshold τ_n node n?* The simplest, frequently used solutions enumerate and evaluate all or a random subset of possibilities. To evaluate an $\langle f_n, \tau_n \rangle$ pair, the children resulting from this pair are evaluated using an impurity metric, and the quality of the split is a weighted average over the impurity of the children. A common impurity metric is the Gini impurity (Breiman, 1996).

Definition 4.4. Gini Impurity
Let $\vec{y} \in \mathcal{L}^N$ be a label vector with N entries. For every $l \in \mathcal{L}$, let

$$p_l = \frac{1}{N} \sum_{i=1}^{N} 1_{\vec{y}_i = l} \ .$$

The Gini impurity for $\vec{y}$ is

$$Gini(\vec{y}) = 1 - \sum_{l \in \mathcal{L}} p_l^2 .$$

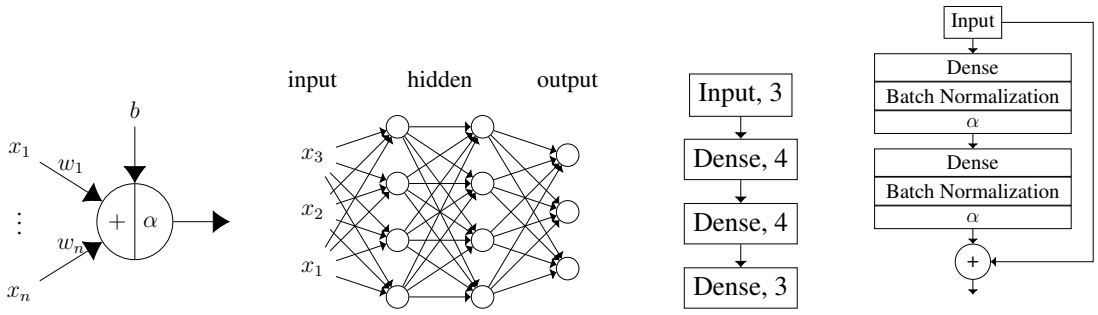
Given a label vector $\vec{y}$, Gini($\vec{y}$) denotes the probability that a sample is misclassified if the samples have the same label distribution as $\vec{y}$. Thus, lower values are better.

A straight forward extension of decision trees are *random forests* (Breiman, 2001). A random forest is an ensemble method. Ensemble methods use internally multiple models and combine their prediction into a single new prediction.

Definition 4.5. Random Forest
Let $T_1, \ldots, T_n$ be decision trees for the features $\mathcal{F}$. $\mathcal{T} = \{T_1, \ldots, T_n\}$ is a random forest. Let $\vec{x} \in \mathbb{R}^{|\mathcal{F}|}$ be a feature vector and $P = \{\{T(\vec{x}) \mid T \in \mathcal{T}\}\}$ be the multi-set containing the predictions of every tree of the random forest. $\mathcal{T}(\vec{x})$ predicts the most frequent label in P.

4.2. Neural Networks

The fundamental building block of a *neural network* (NN) is the *neuron* (see Figure 4.2a). A neuron computes a linear combination of its inputs and transforms this result with an *activation* function α.



(a) Neuron **(b)** Feed-forward network **(c)** Architecture **(d)** Residual block

Figure 4.2: Basic concepts of neural networks.

Definition 4.6. Neuron
Let $\vec{w} \in \mathbb{R}^N$ be a weight *vector, $b \in \mathbb{R}$ be a* bias *term, and $\alpha : \mathbb{R} \mapsto \mathbb{R}$ be an* activation *function. Let $\vec{x} \in \mathbb{R}^N$ be some input. A neuron n is a function $n : \mathbb{R}^N \mapsto \mathbb{R}$ with*

$$n(\vec{x}) = \alpha(\vec{x} * \vec{w} + b).$$

In general, the weights and bias of a neuron are trainable parameters. The activation function introduces non-linearity. A single neuron is the simplest neural network. A *layer* is a set of neurons with the same input dimensionality and the same activation function. The *dense* layer, also called *fully connected*, is one of the most used layer types.

Definition 4.7. Dense Layer
Let $\mathcal{N} = \{n_1, \ldots, n_N\}$ be a set of neurons with M inputs each. Let $\vec{w}_i \in \mathbb{R}^M$ and $b_i \in \mathbb{R}$ be the weight vector and bias of neuron n_i. All neurons use the same activation function α. The weight matrix of the dense layer d is $W = [\vec{w}_1 \ \ldots \ \vec{w}_N]$ and the bias is $\vec{b} = [b_1 \ \ldots \ b_N]$. A dense layer d is a function $d : \mathbb{R}^M \mapsto \mathbb{R}^N$ with

$$d(\vec{x}) = \alpha(W\vec{x} + \vec{b}).$$

A common and simple neural network is a *feed-forward network* (FFN, see Figure 4.2b, see Goodfellow, Bengio, and Courville, 2016). An FFN consists of a sequence of layers. The last layer is called *output layer*. All others are called *hidden layers*. Proposition 4.1 shows the importance of the activation function for FFNs.

Proposition 4.1.
Let $m : \mathbb{R}^M \mapsto \mathbb{R}^N$ be an FFN where all layers use the identity function $f(x) = x$ as activation. Then, there exists an FFN m' consisting of a single dense layer with N neurons such that for all possible input vectors $\vec{x} \in \mathbb{R}^M$ it holds $m(\vec{x}) = m'(\vec{x})$.

Proof:
The function represented by the FFN m with l layers and $\alpha(x) = x$ is

$$\begin{aligned} m(\vec{x}) &= \alpha(\alpha(\dots\alpha(\alpha(\vec{x}W_1 + \vec{b}_1)W_2 + \vec{b}_2)\dots)W_l + \vec{b}_l) \\ &= (((\vec{x}W_1 + \vec{b}_1)W_2 + \vec{b}_2)\dots)W_l + \vec{b}_l \\ &= \vec{x}W_1 \dots W_l + \vec{b}_1 W_2 \dots W_l + \vec{b}_2 W_3 \dots W_l + \dots \\ &= \vec{x}\prod_{i=1}^{l} W_l + \sum_{i=1}^{l} \vec{b}_i \prod_{j=i+1}^{l} W_j \\ &= \vec{x}W' + \vec{b}' \end{aligned}$$

with

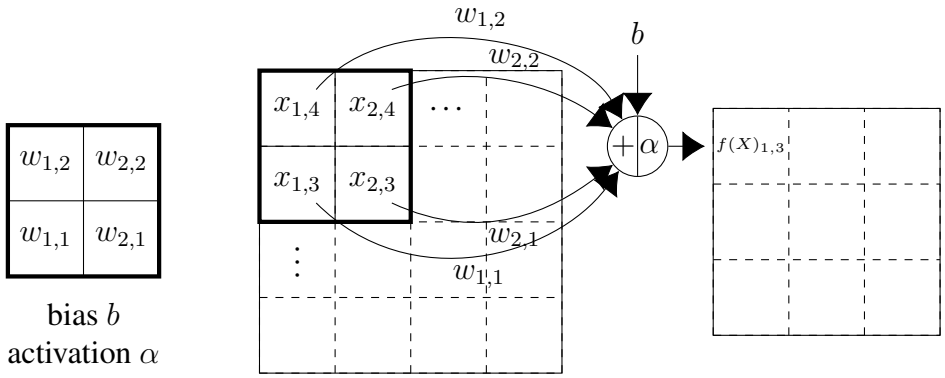
$$W' = \prod_{i=1}^{l} W_l \qquad b' = \sum_{i=1}^{l} \vec{b}_i \prod_{j=i+1}^{l} W_j$$

□

We can replace an FFN with multiple layers without non-linear activation functions by an FFN with a single layer. If we can merge those networks to a single layer, do we require networks with multiple layers? In theory, no! By the universal approximator theorem, any continuous function can be approximated to arbitrary precision by an FFN with a single layer and sufficiently many neurons with sigmoid activation function (Cybenko, 1989). But why do we use then FFN with multiple layers? A single layer FFN is sufficient to represent any function f, but this single layer FFN might require significantly more neurons than a neural network with multiple layers. In general, the more neurons an NN has, the harder it is to train.

Describing the information flow as in Figure 4.2b is infeasible for large networks and describing it in textual form can be tedious. The architecture of a network is regularly visualized in a flow chart as in Figure 4.2c. When a network is trained, its parameters, i.e., the weights and biases of its neurons are updated such that the network approximates a desired function. In most network architectures, the parameters approximate the desired function. He et al. (2016) showed that it can be easier for a network to learn the residuals between the input and the desired outcome. A residual block (see Figure 4.2d) approximates the difference between its input and the desired output. The output of a residual block is the sum of the input and the approximated difference. A residual network is a subgroup of FFN.

Definition 4.8. Dropout Layer
Let L be a layer with n outputs. Let D be a layer with n inputs, n outputs, and a dropout probability of p. Every input of D is connected to a different output of D. During training, the value received by an input is replaced by 0 with probability p. With probability $1 - p$ the input is forwarded. After training, all inputs are forwarded to their outputs. Then D is called a dropout layer.



(a) Convolution filter **(b)** Application of filter

Figure 4.3: (Left) Visualization of a convolution filter and (Right) a single application of that filter.

Another type of layers are *dropout layers*. During training, they randomly suppress some outputs of their previous layer. Thus, the NN cannot put too much focus on single neurons. This can improve the robustness of the final model.

Feed-forward networks accept only inputs of a fixed dimensionality. Many data sets violate this constraint. For example, if the inputs are images, then the width and height of the images can vary. We only know that images have a 2d grid structure. Convolutional neural networks (CNNs, see LeCun, Bengio, and Hinton, 2015) can be constructed for arbitrary d-dimensional grid structured inputs. For a concrete CNN the dimensionality of this grid structure is fixed, but the size of each grid dimension can vary. For this purpose, a CNN has a *convolution layer* with *convolution filters*. Each convolution filter is a neuron that is reused on multiple parts of the input (see Figure 4.3a). We call this reusing *weight sharing*. If the CNN accepts d-dimensional inputs, then the convolution filter has a d dimensional weight matrix. As a side effect of the weight sharing, the number of trainable weights to process a concrete d-dimensional input is a lot smaller than the number of weights necessary if a dense layer would be used for the same image. The reduced number of features means the filter is easier to train, faster to evaluate, and requires less memory.

Definition 4.9. Convolution Filter
Let $W \in \mathbb{R}^{N_1 \times \ldots \times N_d}$ be the weight matrix, $b \in \mathbb{R}$ be the bias term, and $\alpha : \mathbb{R} \mapsto \mathbb{R}$ be the activation function of a convolution filter f. Let $X \in \mathbb{R}^{M_1 \times \ldots \times M_d}$ be some input with $M_i \geq N_i$ and $O_i = M_i - N_i + 1$. The filter f applied on X is a function $f : \mathbb{R}^{M_1 \times \ldots \times M_d} \mapsto \mathbb{R}^{O_1 \times \ldots \times O_d}$.

Let $E = [0, N_1] \times \ldots \times [0, N_d]$ be the set of indices for all individual weights in W and $\vec{p} \in \mathbb{N}^d$ with $1 \leq \vec{p}_i \leq O_i$ be the indices in the filter output. The application of f is

defined as

$$f(X)_{\vec{p}} = \alpha(b + \sum_{\vec{e} \in E} X_{\vec{e}+\vec{p}} * W_{\vec{e}+\vec{p}}).$$

Figure 4.3b visualizes for a 2×2 filter on a 4×4 input the computation of the output value at position $\langle 1, 3 \rangle$. The filter is "in" the top left corner of the input. The "covered" values are multiplied by the weights covering them. These products are summed up, and the sum is transformed with the activation function α. The output of the activation function is the output of the convolution filter for that position. Afterwards, the filter is shifted to a new location. We described convolution filters with outputs that are smaller than their inputs. Using a technique called *padding*, the input and output size can be the same. For brevity, we do not explain it here.

We often use *pooling* in combination with convolution filters. A pooling layer combines spatially neighboring inputs to a single output. A simple example is a 2×2 *max pooling*. Like the convolution filter, it moves a 2×2 square over its inputs. The output of each pooling calculation is the maximum over the inputs "covered" by the square.

CNNs proved useful for many applications, like text or object recognition from images. But they are restricted to grid-structured inputs. Many use-cases have data with arbitrary graph structures, i.e., the data is represented by a graph $G = \langle V, E \rangle$ where V are the vertices of the graph and E the edges. There is a set of features $\mathcal{F}$ and each vertices v is annotated with a feature vector $\vec{x}_v \in \mathbb{R}^{|\mathcal{F}|}$. Thus, graph neural networks (GNNs) emerged as a generalization of CNNs (Bruna et al., 2014; Defferrard, Bresson, and Vandergheynst, 2016; Li et al., 2016; Kipf and Welling, 2017; Hamilton, Ying, and Leskovec, 2017; Gilmer et al., 2017; Veličković et al., 2018).

Remember, let $G = \langle V, E \rangle$ be a graph. Then it has an *adjacency matrix* $A \in \mathbb{B}^{|V| \times |V|}$ with $A_{i,j} = 1$ iff $\langle i, j \rangle \in E$ and $A_{i,j} = 0$ otherwise. The *in-degree* of the i-th node $d(i) = \sum_j A_{j,i}$ denotes the number of incoming edges of the node. The *degree matrix* $D \in \mathbb{N}_0^{|V| \times |V|}$ is a diagonal matrix with $D_{i,i} = d(i)$ and $D_{i,j} = 0$ otherwise. The *identity matrix* $I \in \mathbb{B}^{|V| \times |V|}$ has $I_{i,i} = 1$ and $I_{i,j} = 0$ otherwise.

A convolution filter combines the features of neighboring nodes in the grid-graph to construct its output. A layer in a GNN follows the same idea, but generalized to arbitrary graphs. One instance of such a generalization are AC-GNN layers.

Definition 4.10. AC-GNN layer (Hamilton, 2020)
Let $G = \langle V, E \rangle$ be an arbitrary graph. Let $\mathcal{F}$ be a set of features and for every $v \in V$ let $\vec{x}_v \in \mathbb{R}^{|\mathcal{F}|}$ be the feature vector of vertex v. The output o_v of a AC-GNN layer for a vertex v is

$$o_v = \textsc{Combine}(f_v, \textsc{Aggregate}(\{\{f_{v'} \mid v' \in \mathcal{N}(v)\}\}))$$

where $\mathcal{N} : V \mapsto 2^V$ defines the neighborhood of a vertex, Aggregate *aggregates multiple feature vector into one, and* Combine *computes the new feature vector o_v for v by combining its old feature vector with the aggregated feature vector.*

Countless realizations for $\mathcal{N}$, AGGREGATE, and COMBINE exist. We refer the interested reader to Hamilton (2020). An example is

$$\mathcal{N}(v) = \{v' \mid (v', v) \in E\}$$
$$\text{AGGREGATE}(v) = \sum_{v' \in \mathcal{N}(v)} \vec{x}_{v'}$$
$$\text{COMBINE}(v) = \vec{x}_v W_{self} + \text{AGGREGATE}(v) W_{others} + b$$

where W_{self} and W_{others} are two trainable weight matrices and b is a trainable bias term. We call a GNN consisting only of AC-GNN layers an AC-GNN.

4.3. Training Paradigms

Remember, a model which predicts a continuous value is a regression model. In contrast, a model that predicts one label out of a finite set of labels is classification model. In the setting of classification, we use the words "label" and "class" interchangeably. A neural network can have multiple continuous value outputs. We call this *multi-target regression*. Nevertheless, if we have a finite set of labels $\mathcal{L}$, the network has one output o_l for each $l \in \mathcal{L}$, and we interpret the output of the network such that the prediction of the network is the label with the greatest value o_l, then we call it a classification network. A common loss function for a classification network is the *cross entropy*.

Definition 4.11. Cross Entropy (Rubinstein, 1997)
Let $D = \langle X, \vec{y} \rangle$ be a data set with N samples for a classification task with K classes. $X \in \mathbb{R}^{N \times M}$ is its feature matrix and $\vec{y} \in \{1, \ldots, K\}^N$ is its label vector. Let $Y \in \mathbb{R}^{N \times K}$ be a one-hot encoding of the label vector, i.e., $Y_{i,k}$ is 1 iff $\vec{y}_i = k$. Otherwise $Y_{i,k}$ is 0. Let m be a function $m : \mathbb{R}^M \mapsto \Delta^K$ where Δ is the K-simplex. The cross entropy loss of m on D is

$$CE(m, D) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{K} Y_{i,c} \log(m(X_i)_c).$$

Throughout this thesis, we train all models using *supervised learning*. That means, we train the model using feature vector - label pairs. This is in contrast to *unsupervised learning* where the model is trained using only feature vectors without labels or *reinforcement learning* where the network receives for a feature vector - prediction pair a *reward signal*. *Curriculum learning* is a special form of supervised learning. The network is initially trained on "simple" examples and over time the difficulty of the samples increases.

We use the terms *training*, *validation*, and *test* data to refer to three different data sets. The training data is used during training to update the parameters of the model. The

validation data is used during training to monitor the training progress or afterwards to select hyperparameters. The test data is used for the final performance evaluation of the trained model.

If the training process is non-deterministic, like with neural networks, then the training has to be repeated multiple times and the final test performance is the average over the individual test performances. Furthermore, the individual test performances depend on the training and test data. Repeating the training with the same data sets does not show how reliable the model performs. *k-fold cross validation* solves this problem. It takes a data set D and splits it in k folds $D_1, \ldots, D_k$ each containing approximately the same number of samples. It trains k models. For the i-th model, the i-th fold is used as test data and the $(i+1) \mod k$ fold is optionally used as validation data. All other folds are used as training data. The final test performance is an average over the k individual test performances.

5. Description Logic

Description logic (DL) is a family of knowledge representation formalisms related to FOL (see First-Order Logic). We use DL to represent knowledge about a world. In our setting, the world is a state of a $PDDL$ planning task. We do not use the ability of description logic to reason about the world.

5.1. Concepts, Roles & Individuals

Description logic describes the world using *individuals*, *concepts*, and *roles*. An individual represents an object and relates to a constant in FOL; a concept describes a set of objects, which share some property and relates to a unary predicate in FOL; a role describes a binary relation between objects and relates to binary predicates in FOL.

At the heart of every description logic are *atomic concepts* and *atomic roles*. Atomic concepts describe the fundamental properties of the objects and atomic roles describe the fundamental relations between objects. An interpretation assigns meaning to the individuals, atomic concepts, and atomic roles. Let N_I be the set of all individuals, N_C be the set of all atomic concepts, and N_R be the set of all atomic roles. Let $\mathcal{I} = \langle \Delta^\mathcal{I}, \cdot^\mathcal{I} \rangle$ be an interpretation, then $\Delta^\mathcal{I}$ is the universe. $\cdot^\mathcal{I}$ defines for all individuals $I \in N_I$ a constant value $I^\mathcal{I} \in \Delta^\mathcal{I}$, for all atomic concepts $C \in N_C$ a set of objects $C^\mathcal{I} \subseteq \Delta^\mathcal{I}$, and for all atomic relations $R \in N_R$ a relation $R^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$. We call the interpretation of an individual $I^\mathcal{I}$, a concept $C^\mathcal{I}$, and a role $R^\mathcal{I}$ a denotation.

Furthermore, a specific description logic fragment defines rules to inductively construct new concepts and roles and to evaluate their denotations. We call these constructed concepts and roles *complex*. Let C and C' be concepts, then negation ($\neg C$) and intersection ($C \sqcup C'$) are two examples for such rules (Baader et al., 2003).

5.2. Description Logic for Planning

If we want to use description logic for a planning task, then we have to define the individuals, atomic concepts, atomic roles, and the rules to use. Furthermore, we have to define a method to construct interpretations for the states of the planning task. For this purpose, we follow the instructions of Drexler, Francès, and Seipp (2022).

Let $\Pi_{PDDL} = \langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_\mathrm{I}, \delta \rangle$ be a $PDDL$ planning task with a set of states $\mathcal{S}$ and a state $s \in \mathcal{S}$. We construct an interpretation $\mathcal{I} = \langle \Delta^\mathcal{I}, \cdot^\mathcal{I} \rangle$ for s with $\Delta^\mathcal{I} = \mathcal{O}$. For every

Name	Syntax	Complexity $\mathcal{K}$	Denotation
Top	$\top$	1	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	1	$\emptyset$
One-of	a	1	$\{a\}$
Negation	$\neg C$	$1 + \mathcal{K}(C)$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Intersection	$C \sqcap D$	$1 + \mathcal{K}(C) + \mathcal{K}(D)$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$C \sqcup D$	$1 + \mathcal{K}(C) + \mathcal{K}(D)$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Existential	$\exists R.C$	$1 + \mathcal{K}(C) + \mathcal{K}(R)$	$\{a \mid \exists b : \langle a, b\rangle \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
Value restriction	$\forall R.C$	$1 + \mathcal{K}(C) + \mathcal{K}(R)$	$\{a \mid \forall b : \langle a, b\rangle \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
Role-value-map	$R \subseteq S$	$1 + \mathcal{K}(R) + \mathcal{K}(S)$	$\{a \mid \forall b : \langle a, b\rangle \in R^{\mathcal{I}} \rightarrow \langle a, b\rangle \in S^{\mathcal{I}}\}$
Composition	$R \circ S$	$1 + \mathcal{K}(R) + \mathcal{K}(S)$	$\{\langle a, c\rangle \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \langle a, b\rangle \in R^{\mathcal{I}} \wedge \langle b, c\rangle \in S^{\mathcal{I}}\}$
Inverse	R^{-1}	$1 + \mathcal{K}(R)$	$\{\langle b, a\rangle \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \langle a, b\rangle \in R^{\mathcal{I}}\}$
Transitive closure	R^{+}	$1 + \mathcal{K}(R)$	$\bigcup_{n \geq 1} (R^{\mathcal{I}})^n$

Table 5.1: The rules we use to construct complex concepts and roles with their syntax, their complexity, and the explanation to calculate their denotations.

predicate symbol $p \in \mathcal{P}$ with $k = arity(p)$ and every value $1 \leq i \leq k$ we define an atomic concept $C_{p,i}$ with the denotation

$$C_{p,i}^{\mathcal{I}} = \{o_i \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}.$$

Furthermore, for every predicate symbol $p \in \mathcal{P}$ with $k = arity(p)$ and every pair i, j with $1 \leq i, j \leq k$ we define an atomic role $R_{p,i,j}$ with the denotation

$$R_{p,i,j}^{\mathcal{I}} = \{\langle o_i, o_j\rangle \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}.$$

Often it is insufficient to reason only about the state. We also have to reason about the goal. Thus, we define additional atomic concepts $C_{p_g,i}^{\mathcal{I}}$ and an atomic roles $R_{p_g,i,j}^{\mathcal{I}}$ which consider the goal instead of the current state:

$$C_{p_g,i}^{\mathcal{I}} = \{o_i \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in \delta\}$$
$$R_{p_g,i,j}^{\mathcal{I}} = \{\langle o_i, o_j\rangle \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in \delta\}.$$

Whether and which individuals we define depends on the task.

To construct complex concepts and roles, we use the rules shown in Table 5.1. We use the same measure as Drexler, Francès, and Seipp (2022) for the complexity $\mathcal{K}$ of concepts and roles. The complexity of an atomic concept or role is 1. For the complex ones, it is recursively calculated (see Table 5.1).

Boolean Features. Later, we learn formulas in disjunctive normal form using the generated concepts and roles. Thus, we convert the denotation of a concept or role to a Boolean value following Drexler, Francès, and Seipp (2022). Let X be a concept or role, we define a *description logic feature* (in short just *feature*) $|X| > 0$, which checks if the denotation of X is not empty. Furthermore, let C_1, C_2 be concepts and R be a role. The *concept distance* between C_1 and C_2 using R is the smallest $n \in \mathbb{N}_0$ such that $x_0 \in C_1^{\mathcal{I}}$, $x_n \in C_2^{\mathcal{I}}$, and $\langle x_i, x_{i+1} \rangle \in R^{\mathcal{I}}$ for $0 \leq i < n$. The complexity of the size features is $\mathcal{K}(|X| > 0) = 1 + \mathcal{K}(X)$ and the complexity of the concept distance features is $\mathcal{K}(concept\_distance(C_1, R, C_2)) = 1 + \mathcal{K}(C_1) + \mathcal{K}(R) + \mathcal{K}(C_2)$.

5.3. Relationships

Most description logics express a fragment of first-order logic (FOL). Given an interpretation for such a description logic, we can express their concepts and roles as FOL formulas. This especially holds for a description logic which uses the rules, except for the transitive closure, from Table 5.1.

Theorem 5.1. Conversion from Description Logic to FOL (Baader et al., 2003)
Let C be a concept constructed with the rules from Table 5.1 except for the transitive closure, and $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ be an interpretation. Then we can efficiently construct a formula $\phi_C(x)$ such that $x \in C^{\mathcal{I}}$ iff $\phi_C(x)$.

Proof sketch:
The universe for the FOL formula $\phi_C(x)$ is $\Delta^{\mathcal{I}}$. If C is an atomic concept, then we define an unary predicate symbol ϕ_C with $\phi_C(x)$ holds iff $x \in C^{\mathcal{I}}$. Let D, E be two concepts. If $C = D \sqcap E$ is constructed using the intersection rule, then we first construct ϕ_D and ϕ_E and define $\phi_C(x) = \phi_D(x) \wedge \phi_E(x)$. The construction for negation and union can be done analogously.

Let R be an atomic role. We define a binary predicate symbol ϕ_R with $\phi_R(x, y)$ holds iff $\langle x, y \rangle \in R$. If $C = \exists R.D$ is a complex concept using the existential rule, then we first construct ϕ_R and ϕ_D and define $\phi_C(y) = \exists x : R(y, x) \wedge \phi_C(x)$. For all complex rules a translation exists. We refer the interested reader to Baader et al. (2003).

□

Counting-variable logic introduces *counting quantifiers*. A counting quantifier $\exists^{\geq N} \phi$ expresses that there exists at least N distinct objects, which satisfy ϕ. In k-variable logic (C_k) counting quantifiers with $N \leq k$ can be used. Counting variable logic is a subset of FOL (Grädel, Otto, and Rosen, 1997).

Barceló et al. (2020) showed AC-GNN can represent a fragment of C_2 logic, which is equivalent to $\mathcal{ALCQ}$ description logic. $\mathcal{ALCQ}$ description logic also corresponds to *graded model logic* (de Rijke, 2000). Furthermore, let us introduce a global aggregate function AGGREGATEGLOBAL which aggregates the feature vectors over *all* vertices

in the graph and change the COMBINE function to

$$o_v = \text{COMBINE}(f_v, \text{AGGREGATE}(\{\{f_{v'} \mid v' \in \mathcal{N}(v)\}\}), \text{AGGREGATEGLOBAL}()).$$

We call a GNN which such layers an ACR-GNN. Barceló et al. (2020) also showed that such a GNN can represent any C_2 formula.

Part I.

Learning Heuristics

For this part, I collaborated with the following people (listed in alphabetical order):
Florian Geißer, Malte Helmert, Jörg Hoffmann, Felipe Trevizan

6. Introduction to Learning Heuristics

Heuristic search, especially *greedy best-first search* (GBFS) and A^* *search*, is one of the most successful approaches for solving classical planning tasks. The heuristic estimates for every state the cost of length of a path to the next goal state. The search uses these estimates to order the encountered states. Greedy best-first search orders the encountered state solely on the heuristic estimates. It expands first states with lower estimates. The performance of heuristic search depends strongly on the quality of the used heuristic. Over the years, many powerful, admissible heuristics were designed to find optimal plans with A^* search (Helmert and Domshlak, 2009; Helmert et al., 2014; Seipp, 2017). Often, optimal plans are not required, but any plan suffices. We call this setting *satisficing planning*. Many strong heuristics were designed for satisficing planning (Hoffmann and Nebel, 2001; Richter and Westphal, 2010; Domshlak, Hoffmann, and Katz, 2015). For satisficing planning, the heuristics can be indadmissible or even accurate. Especially for GBFS holds, if the state ordering of a heuristic is consistent with the state ordering of the perfect heuristic, then GBFS behaves identical with both heuristics (up to random tie-breaking).

Besides predicting accurate estimates or correct state orderings, a second important property is the *evaluation speed*. GBFS commonly generates millions or more states during search. The heuristic is computed on each generated state. In practice, a better informed heuristic can perform worse because it is too slow to evaluate. Designing a well-informed and fast heuristic is a difficult endeavor. The LAMA planning system Richter and Westphal (2010) participated already in the International Planning Competition 2008 and is still used as a baseline.

It is easier to design a heuristic if it does not need to perform well on all domains, but only on a single domain or even on a single task from a domain. This restriction could enable a human to reason about the properties of a concrete task and design a specific heuristic. In practice, even a single task can be too complex for a human and human labor is valuable. With the breakthroughs in machine learning in the last decades, the question arose whether we can automatically learn heuristics. Early approaches evaluated existing heuristics in a domain and learned to combine them into more powerful heuristic (Arfaee, Zilles, and Holte, 2011; Virseda, Borrajo, and Alcázar, 2013).

Heuristics are not only used in planning, but also in many classical games like chess (Knuth and Moore, 1975) as well as in computer games (Sturtevant, 2012). In the last decades, we witnessed breakthroughs in training and applying neural networks. Recently, Silver et al. (2016) trained neural networks as heuristics for the game of Go and combined this with other planning techniques. The product is *AlphaGo*. AlphaGo

is an AI agent, which plays not only the game of Go, but also won against Lee Sedol, one of the best Go players at that time. From these results sprung a series of research in which trained neural networks as heuristics (Silver et al., 2017, 2018; Agostinelli et al., 2019). In all of these projects, they have a single task (or a small set of tasks) which they want the computer to solve. They manually design and tune every step in the pipeline (input representation, network architecture, ...) for this task. This contradicts our perspective in planning. In planning, we want to design a general purpose solver. Give it any task, and it finds a solution without human interaction.

Nevertheless, those projects show the potential of well-trained neural networks as heuristics. In the last few years, research was inspired by the hand-tailored NN approaches. Planning researchers designed workflows for training and using NN on any domain and task without human interaction. A key difficulty is to sample good training data. In contrast to reinforcement learning, we do not get rewards after every step. In classical planning, we receive a single reward, after solving the task. Simply applying random actions is unlikely to end in a goal state. Furthermore, it is unclear which network architecture is beneficial for planning. Every published paper uses a new architecture.

Some research uses GNN (Issakkimuthu, Fern, and Tadepalli, 2018; Toyer et al., 2020; Garg, Bajpai, and Mausam, 2019; Rivlin, Hazan, and Karpas, 2020; Shen, Trevizan, and Thiébaux, 2020) to learn knowledge, which generalizes across a whole domain. Many of those works are situated in probabilistic planning instead of classical planning (Issakkimuthu, Fern, and Tadepalli, 2018; Toyer et al., 2020; Garg, Bajpai, and Mausam, 2019). A possible reason is that probabilistic planning tasks in the *RDDL* formalism (Sanner, 2010) provides a reward after every action. This facilitates training policies with reinforcement learning. Indeed, most of them learn policies on *RDDL* tasks (Issakkimuthu, Fern, and Tadepalli, 2018; Garg, Bajpai, and Mausam, 2019). Only Rivlin, Hazan, and Karpas (2020) and Shen, Trevizan, and Thiébaux (2020) use classical planning. We compare the performance of our networks against the networks of Shen, Trevizan, and Thiébaux (2020) [1].

[1] Rivlin, Hazan, and Karpas (2020) informed us that they were unable to reproduce their results and recommended not to use their framework as comparison.

7. Neural Network Heuristics

We limit ourselves to *reset problems*[1]. That means we train NN as heuristics for a single state space with a fixed goal. Examples of reset problems are Rubik's cube, the sliding tile puzzle, and scenarios in which a system has to be reset to a default state. Restricting ourselves to reset problems, allows us to tackle the challenge of learning heuristics from its base. Starting with FFNs avoids the biases and parameter choices of CNNs and GNNs before we even know that learning good heuristics is possible. The FFN for a reset problem receives a Boolean input for each fact of the task. In a given state, the input of a fact is true if the fact is in that state and false otherwise. Our initial study evaluates the feasibility and the hyperparameter options of such an FFN. Building upon this, we present new approaches, which solve some observed limitations.

Our learned heuristics have almost no guarantees on the predictions. They are safe, because our neural networks simply cannot express infinity. They are goal-aware, because we check for every evaluated state if it is a goal state and only if it is not a goal state, we evaluate the network. They are neither admissible nor consistent. Thus, all our evaluations use the satisficing setting, i.e., we accept any plan, not just optimal plans. Our methods can be applied to unit and to non-unit cost tasks. In general, solving the unit cost version of a non-unit cost task terminates faster. Thus, we transform all our tasks to their unit cost version. Consequently, all goal cost estimators are also goal distance estimators.

We evaluate our methods against each other, against Shen, Trevizan, and Thiébaux (2020)'s state of the art in learning heuristics, and against model-based heuristics.

7.1. Progression Heuristics

We present now our initial framework and list parameter options. The framework consists of two steps. First, we generate the training data. Then, we train the heuristics. Afterwards, we can use the heuristic in search. A key property is that it uses a progression random walk to generate the training data. In our experiments we evaluate the possible parameter choices.

Sampling Data. Once per task Π, we train an FFN as heuristic with supervised training. We require state, goal cost pairs as training data. Thus, we have two challenges:

[1]Thanks to Rob Holte for suggesting this name.

How to sample states? How to label sampled states with their cost to the goal? For many domains, there exists no tool to generate new states, which we could use as samples. Furthermore, in a real world scenario a user might provide a planning task, but they might be unable to describe the distribution of valid states and how to generate them. Additionally, to obtain good goal cost estimates is expensive.

To solve the first challenge, we take inspiration from Arfaee, Zilles, and Holte (2011). In their setting, the goal δ of a task Π is a valid state and all actions are reversible, i.e., for every action a there exists an action a' which reverses the changes caused by a. Starting from the goal, they apply a random sequence of applicable actions and end in a state s. If they apply a sufficiently long sequence of actions, then any state can be reached, and the chance of ending in the same state twice is negligible. In our setting, the goal is *not* a valid state, but a partial state. Thus, we start our random walks from the only known valid state, the initial state s_{I}. Furthermore, actions are not necessarily reversible. Thus, we cannot generate states which are not reachable from the initial state. We do not expect our heuristics to generalize on those states.

Now we can sample a states. Our next challenge is to label them. Let s be a sampled state. It would be simplest, if we pick an existing heuristic h, evaluate it on s, and use this value as label. If we do this, the FFN will just imitate h. It would be more sophisticated to choose a set of *admissible* heuristics H and label the state s by the maximum estimate $h(s)$ over the heuristics $h \in H$. Again, the network imitates just a function, which is the maximum over a set of heuristics, and we could replace it by the maximum over these heuristics. Instead, we choose a planner, which we call *teacher*. For every sampled state s we execute the teacher. If the teacher finds an s-plan $\pi = \langle a_1, \ldots, a_n \rangle$ within some resource limits, then we add *all* states $s_i = s[\![\langle a_1, \ldots, a_i \rangle]\!]$ along the plan to our data set (for $0 \leq i \leq n$). Each state s_i is labeled by the remaining cost $c_i = \sum_{j=i+1}^{n} \mathit{cost}(a_j)$. This has the advantage that the relatively fast to evaluate FFN imitates a slow function, which produces high quality labels.

Input and Output Representations. Knowing our training data, we specify the concrete network architecture. Let Π be a planning task with a set of facts F. Many tasks contain facts which *never* change their value. A task from the Transport domain has facts, which describe a road map. For a specific Transport task the road map is static. There is no need to inform the FFN about the road map in every state. The road map does not change and the FFN should learn it during training. We identify for every task Π the set of dynamic (non-static) facts F_D. Our FFN for Π has $|F_D|$ input neurons, one per dynamic fact. To evaluate the network on a state s, we check for every fact $f \in F_D$ if it is in the current state ($f \in s$). If it is in the current state, we set the associated neuron to 1, otherwise we set it to 0.

The output representation has a huge impact on the success of the NN. We require an output, which represents an integer. Intuitively, we would use a *regression* encoding, i.e., a single output neuron with ReLU activation. The learning objective does not

sound like a classification task, but classification networks often perform unexpectedly well. Thus, we additionally evaluate classification networks. Let H be the maximum heuristic value in the sampled data. Our classification networks have $H+1$ output neurons, each one is associated with an observed heuristic value $0 \leq h \leq H$. Almost all classification networks encode the labels using a *one-hot* encoding and *softmax* activation for the output neurons. During training, the heuristic value h is encoded as a vector $\langle y_0, \ldots, y_H \rangle$ where y_h is set to 1 and all other entries are set to 0. When evaluating, the predicted heuristic value h is the output y_h with maximal value.

The one-hot encoding does not represent the relationship between the classes. For a classification network with one-hot encoding the difference between the classes $h = 0$ and $h = 1$ is the same as the difference between the classes $h = 0$ and $h = 100$. But there is a relation between our classes. The heuristic value of 1 is closer to the value of 2 than it is to the value of 100. If we use a *unary* encoding with sigmoid activation on the output neurons, then this relation is preserved (Cheng, Wang, and Pollastri, 2008). During training, the unary encoding for a heuristic value of h is a vector $\langle y_0, \ldots, y_H \rangle$ where all entries y_i with $i \leq h$ are set to 1 and all other entries are set to 0. When evaluating, all outputs with a value greater than a threshold t are interpreted as 1. All other outputs are interpreted as 0. The prediction of the FFN is the number of consecutive ones at the beginning of the output vector minus one.

Network Layers. We specified the network input and listed possible output encodings. But also the inner layers provide options for tweaking. Anything that can be done in an FFN is possible. We present and evaluate later only the most common parameters. First, we evaluate different numbers of hidden layers. We expect that more difficult tasks require more powerful FFN to learn their heuristics. Thus, we adapt the number of neurons in the hidden layers to the task. We scale the number of neurons in each network layer in equally sized steps down from the input layer, which has one neuron per dynamic fact, to the output layer, which has either 1 or $H+1$ neurons. Secondly, different activation functions like sigmoid and ReLU can be used for the hidden layers. To prevent overfitting, regularization can be incorporated. One common option is to add a dropout layer after each hidden layer. Remember a dropout layer has a dropout probability p. With p probability, a neuron of the dropout layer does not forward its value to the next layer during training. This prevents the FFN from relying too much on a single neuron. A second regularization option adds the L2 norm of the trainable weights as loss to the objective function to prevent overfitting with large weight values.

Data Distributions. The final set of knobs we propose and evaluate concern the data distribution during training. While generating training data, we solved planning tasks and stored all states along the *entire plan* found. This is the largest data set we have. It has some drawbacks. The states extracted from a single plan highly correlate. Between two consecutive states of a plan, only a few facts differ. This could introduce a bias to

the network. Thus, we propose to train on a subset of the sampled data, which consists of one *random state* per plan. Every plan ends in a goal state. Thus, the entire plan data set contains for every plan a goal state, a state which is one step away from the goal, a state which is two steps away from the goal, and so forth. There will be approximately the same number of goal states than states of intermediate distance to the goal or states far away from the goal. At the same time, most tasks have significantly fewer states around the goal than states far away from the goal. Thus, states close to the goal will appear multiple times in the training data and are more important during training. To counteract this bias, we suggest a third data set consisting of only the *initial state* of every plan.

Another option to solve the overrepresetation of few states (especially those close to the goal) is to prune duplicates in the training data. It is likely that this stronger affects states with low heuristic value. Thus, the higher heuristic values will now be overrepresented. This could also be problematic. To solve this overrepresentation, the samples can be weighted. Let N_h be the number of samples in the training data labeled with h. If we weight all samples with a label of h by $1/N_h$, then all heuristic classes have the same impact. Because pruning and weighting influence each other, we evaluate them in combination.

7.2. Regression Heuristics

We observe that the data generation is a bottleneck. Sampling states via a progression random walk from an initial state results in new states, which are far away from the goal. We are especially interested in learning heuristics for difficult tasks, but for those tasks the teacher will frequently be unable to find plans for the sampled states. A solution could be a better teacher. But this changes only our definition from hard tasks to even harder tasks. Then we again need a better teacher. At one point, there is no better teacher to pick.

As a remedy, we propose to iteratively scale the difficulty of the training samples, as in curriculum learning, and to use the current state of the network together with some other techniques to generate the labels. While we train our FFN they become better informed. Thus, they can solve harder tasks. Thus, we get training data from harder tasks. Thus, we can further improve our FFNs. To scale the difficulties of the samples, we propose a new sampling procedure.

Sampling States. Let Π be an FDR task for which we train an FFN as heuristic. In the beginning, our FFN is too uninformed to find plans for most states. Thus, we need to generate states which are close to the goal. The better informed the network becomes, the more difficult should be the state we sample. Our sampling procedure requires a parameter which influences the difficulty of the sampled states. Arfaee, Zilles, and Holte (2011) suggest to sample states with random walks from the goal. The walk

length correlates loosely with the difficulty of the sampled state. In their setup, the goal is a state. Thus, they can apply actions on it. In our setup, the goal is a partial state. We cannot apply actions to partial states. Instead, we regress over actions (see Definition 3.8). If we regress over a partial state p, like the goal, with an action a, the result is another partial state p'. For all states $s' \in \mathcal{S}$ described by p' ($s' \supseteq p'$), it is guaranteed that a is applicable in s' and that applying it leads to a state s which is described by p ($s'[\![a]\!] \in S_P$). In this way, we can use the length of the regression random walk to influence the difficulty of our samples. The maximum distance of a sampled partial state to the goal is the length of the walk. Using the regression operator resolves the limitation of the forward random walk that all valid states have to be reachable from the initial state.

For training, we do not require partial states, but states. Thus, we have to pick a state s from the set of states S_p described by the sampled partial state p. Remember, an FDR task has a set of finite-domain variables. A state is an assignment to all variables, a partial state is an assignment to a subset of those variables. To get a state s described by a partial state p, we can randomly assign facts to the variables without assignment. This can produce unintended states. For example, a task has two robots, which can never be at the same location. The FDR representation of this task has one variable for each robot to represent its location. If we randomly assign a value to each variable, it can happen that both robots are at the same location in our completed state. In practice, it is often possible to identify *mutexes*, i.e., groups of facts which are mutually exclusive. A state which contains two mutually exclusive facts is an *invalid state*. We uniformly at random assign facts to the unassigned variables such that no known mutex is violated.

Labeling States. Next, we present three methods to use the current state of an FFN to label the sampled states. The first two are based on *bootstrapping*, i.e., we use the FFN during actual searches to generate labels. The last one uses *approximate value iteration*.

Our first method labels a sampled state with an estimated *cost* to the goal. Let s be a sampled state and h be the heuristic modeled by the current state of our FFN. Then we execute a GBFS with the heuristic h. If it finds a plan within a resource limit, then we add all states along the plan together with their costs along the plan to the training data. If it does not find a plan within the resource limit, we ignore the state. If we train the FFN on these samples, then it learns to estimate the cost from a state to the goal. If the task has unit-costs, then it learns to estimate the distance to the goal.

We want to find plans quickly. If two heuristics are equally fast to evaluate, then we prefer the one which requires fewer expansions with GBFS. Thus, we do not want the network to predict for a state its goal-cost or goal-distance, but the number of states GBFS has to expand until it finds a plan. As a second method, we propose to label a sampled state s with the number of states expanded by a GBFS which uses as heuristic

h the FFN.

$$L(s) = \begin{cases} \#\text{expansions of GBFS}(s, h) & \text{if } s \text{ is solvable} \\ \infty & \text{otherwise} \end{cases}$$

In practice, we have to enforce a resource limit on these GBFS. We cannot label every state for which we fail to find a solution by infinity. This label would be widely inaccurate for solvable states. The simplest option is to ignore these states. In exploratory experiments, we observed that the information that a state is solvable, but still too difficult to solve is beneficial. Thus, we label those states by the number of expansions GBFS performed until the resource limit was reached. This number is often much higher than the labels for samples with solutions.

Under mild assumptions, training the FFN as search space size estimator converges to the perfect heuristic.

Theorem 7.1. Convergence of Search Space Size Estimators
Let Π be a unit cost task with the set of states $\mathcal{S}$. If a heuristic uses a lookup-table $G \in \mathbb{N}^{|\mathcal{S}|}$ to store its heuristic estimates and updates the table for all states simultaneously using the equation above, then it converges to the perfect heuristic.

Proof:
Denote by H_n^* the set of states whose heuristic value is their perfect heuristic value after n updates: $H_n^* := \{s \mid s \in \mathcal{S}, G_n(s) = h^*(s)\}$. For convenience, we start counting the update iterations with $n = 0$. The initial lookup-table G_0 is arbitrarily initialized. We show by induction that $H_n^* \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$.
Induction basis: After iteration $n = 0$, all goal states have the value 0, so, $H_0^* \supseteq \{s \mid s \in \mathcal{S}, h^*(s) = 0\}$
Induction step: s is a state with $h^*(s) = n$. Then s has a successor s' with $h^*(s') = n - 1$. By induction hypothesis, we have $H_{n-1}^* \supseteq \{s \mid s \in \mathcal{S}, h^*(s) \leq n-1\}$ and $s' \in H_{n-1}^*$. Thus, there is a path P from s' to the goal with G_{n-1} decreasing by 1 in each step. A GBFS run on s generates s' when expanding s, and afterwards follows P (or another path of the same length) resulting in n expansions. Hence $H_n^* \supseteq \{s | s \in \mathcal{S}, h^*(s) = n\}$. Because of the induction assumption, the same argument applies to all states t where $h^*(t) < n$: GBFS follows a direct path to the goal. So we have $t \in H_n^*$, and hence $H_n^* \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$ as desired.

In all updates, GBFS executed on a dead-end state s proves that s is a dead-end. Thus, all states s with infinite h^* value also satisfy $h^*(s) = G_n(s)$, for all n. This proves the claim.

Proof taken from Ferber et al. (2022b)
□

Inspired by Agostinelli et al. (2019) our final method uses approximate value iteration (Bertsekas and Tsitsiklis, 1996, AVI). Approximate means we use a function approximator, like an FFN, instead of a lookup table to store the estimates for a state.

Value iteration means that we calculate in every iteration i new estimates for our states using the current estimates. As the update function, we use the Bellman update:

$$h_{i+1}(s) = \max_{a \in A, applicable(s,a)} R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') h_i(s')$$

where $R(s,a)$ is a reward for applying action a in state s and $P(s,a,s')$ is the probability that applying action a in state s ends in the state s'. γ is a discount factor to determine the trade-off between current rewards $R(s,a)$ and future rewards $h(s')$.

We can simplify the Bellman update in the context of classical planning. We have no rewards which depend on $\langle state, action \rangle$ tuples, but negative reward (action costs) which depend only on the action. Furthermore, the outcome of an action is not probabilistic, but deterministic. Let a be an action and s, s' be states. If a is applicable in s and applying a in s results in s', then $P(s,a,s')$ is 1, otherwise $P(s,a,s')$ is 0. Thus, we can simplify the equation to

$$h_{i+1}(s) = \max_{a \in A, applicable(s,a)} -\mathit{cost}(a) + \gamma h_i(s[\![a]\!]).$$

As we work only with negative rewards, we can replace the maximization by a minimization and the negative reward by the cost. Furthermore, we have no trade-off between current costs and future costs. All costs are the same for us. Thus, we do not discount future costs. Without discounting the future costs, the updates can increase the values indefinitely. As a remedy, we introduce a ground truth. The resulting formula is

$$h_{i+1}(s) = \begin{cases} 0 & \text{if } s \supseteq \delta \\ \min\limits_{a \in A, applicable(s,a)} \mathit{cost}(a) + h_i(s[\![a]\!]) & \text{otherwise.} \end{cases}$$

AVI does not require a plan for the sampled states, thus, it is significantly faster to evaluate, but also every update is less informative. As with the state-space size estimator, the heuristic function represented by an FFN trained with AVI converges to the perfect heuristic (Bertsekas and Tsitsiklis, 1996).

Workflow. In the previous workflow, we first sampled and labeled data and then trained our networks. Now, both happen in parallel. One process samples and labels states using the FFN with its current weights. Another process updates the FFN. Because a maximum heuristic value is unknown during training, the FFN requires a regression output. We can neither use classification networks with one-hot encoding nor with unary encoding. For both bootstrapping methods, the sampling has to start with short random walks. Once GBFS consistently solves the sampled states, the *maximum* random walk length is increased. For AVI, the walk length is irrelevant. Thus, we always allow a long maximum random walk length. When sampling, the actual walk length is randomly picked between 0 and the maximum walk length.

Validation. We observe that the performance of our FFN is not robust. Retraining an FFN for the same tasks can drastically change the number of tasks it solves. Thus, we sample a set of validation states from the same distribution as the test states. After training, we evaluate the performance of the FFN as heuristic for GBFS on the validation states. If it solves too few tasks, we retrain the FFN.

8. Experiments

We presented our methods to train a neural network as a heuristic for reset problems. For our first method, we list many parameter choices. These choices are at least partially applicable for the later methods. In the following, we first explain the setup of our experiments. Then, we show that simple machine learning models are not powerful enough to represent good heuristics and that neural networks are required. Afterwards, we evaluate the parameter choices for the progression heuristic. Building upon this, we train and evaluate our regression heuristics and finally compare them against other state-of-the-art techniques. For readability, we say "the coverage for h" instead of "the coverage for GBFS when using h as heuristic".

8.1. Setup

Tasks. We use the following nine domains from the IPC: Blocksworld, Depots, Grid, Pipesworld-NoTankage, Rovers, Scanalyzer, Storage, Transport, VisitAll, as well as the NPuzzle domain. All those domains are dead-end free. Thus, our forward sampling approach cannot generate unsolvable states. If a task has non-unit action costs, we transform it to unit action costs. We classify the published tasks of these domains into the categories *trivial*, *moderate*, and *hard*. If GBFS with the FF heuristic (Hoffmann and Nebel, 2001, h^{FF}) solves a task in less than a second, it is trivial. If the search requires at least a second, but less than 900 seconds, then we say the task has a moderate difficulty. Otherwise, the task is hard. We ignore all trivial tasks. We remark that these classifications are only guidelines. Rerunning the searches on different hardware, with different implementations, or even just with a different random seed, produces different results.

For all non-trivial tasks, we use the forward sampling approach with a walk length of 200 to generate 200 test states. Even though it is unlikely that we sample a state twice, we verified that this indeed never happens.

Planners & Heuristics. In our experiments, we use our progression heuristic (h^P) and our regression heuristics (goal-distance estimator h^{GD}, search-space size estimator h^{SE}, and approximate value iteration h^{AVI}). As a learning-based comparison, we use the STRIPS-HGN heuristic (Shen, Trevizan, and Thiébaux, 2020). STRIPS-HGN is trained on a few simple tasks of a domain and generalizes to any task of the same

Domain	# Samples	Teacher Values Min Max	Max Max
Blocksworld	504K	145	327
Depots	98K	46	414
Grid	123K	72	112
Pipesworld-NT	87K	91	411
Scanalyzer	43K	19	100
Storage	10K	107	133
Transport	152K	56	130

Table 8.1: For each domain, the median number of data samples generated on the moderate tasks. Furthermore, the minimal over the maximal teacher values (min max) observed on the moderate tasks and the absolute maximum over the observed teacher values (max max).

domain. As a model-based comparison, we use the FF heuristic. We evaluate each heuristic in a GBFS.

Furthermore, we compare against Mercury (Domshlak, Hoffmann, and Katz, 2015) and the first iteration of LAMA (Richter and Westphal, 2010), two strong planners from previous IPCs.

To generate the training data for our progression heuristic, we perform a forward random walks with a maximum length of 200. We note that in practice some steps undo previous steps, thus, the effective walk length may be shorter. We use GBFS with the FF heuristic as the teacher. We execute the teacher for at most 30 minutes with 3.8 GB of memory on a sampled state. We use a generous time limit of 400 hours for each task to generate the training data. In most tasks, this time limit is excessive, but in some tasks it is expensive to generate enough training data. Table 8.1 shows the median number of samples generated on the moderate tasks of each domain. The number of training states generated per task ranges from 3,000 to 500,000. Only in three tasks we generated fewer than 20,000 states. Furthermore, Table 8.1 shows over all moderate tasks of a domain, the minimal maximal teacher value, and the maximal maximal teacher value. This shows that in almost all tasks the range of heuristic values to predict is large.

To train our supervised learning heuristics, we split the sampled data into 10 folds, and we train ten models. Each model uses a different fold as validation data and is trained on the remaining folds. During training, we optimize for the mean squared error using a batch size of 100. The training stops after at most 48 hours, at most 1,000 epochs, or once the error on the validation data converged. For training, we use 12 GB of memory and 4 CPU cores. All our NN have only fully connected layers. If not otherwise specified, the NNs have 3 hidden layers with sigmoid activation function without any regularization, pruning, or weighting. We select from every sampled plan a single

random state for training. At the end of this chapter, we compare multiple learning-based heuristics. To make their computational resources comparable, we reduce the sampling time to 56 hours and the training time to 2.8 hours. We observed that for most tasks, fewer training data is sufficient, and the training finishes long before the time limit (see Table 8.11).

For our regression heuristics, sampling runs in parallel to training. For the bootstrapping based approaches, we start with a maximum regression random walk length of 5. On every sampled state, we execute GBFS with a time limit of 10 seconds. Whenever we observe that GBFS using the current network as heuristic solves at least 95% of the sampled states, then we double the maximum walk length. We double the walk length at most 8 times. Thus, we walk for at most $1,280$ steps. We observe that further extending the walk does not lead to new states. Instead, it only costs time. For the approximate value iteration approach, iteratively extending the walk length is not necessary. Thus, we start with a maximum walk length of $1,280$. To complete the partial states, we use those mutexes identified by Fast Downward (Helmert, 2006). Furthermore, we extend the Bellman update to check not only the direct successors of the sampled state, but to propagate the estimates from the successors of the successors. Using this two-step lookahead turned out to be beneficial. As the predictions of the network become better over time, the labels of old samples become deprecated. Thus, we use experience replay, i.e., the sampled data is pushed into a *first-in-first-out* buffer of size 25,000. In each training epoch, we randomly pick batches of 250 samples from the buffer. We optimize the network using the mean squared error. To prevent instabilities during training, we update the model for labeling if at least 50 epochs passed and the validation error is below 0.1. We train for 28 hours on 4 cores. Training *cannot* stop early. Like Agostinelli et al. (2019), we use residual neural networks. Our networks consist of two dense layers, followed by two residual blocks, followed by a single output neuron. Each residual block contains two dense layers. Each dense layer has 250 neurons with ReLU activation.

STRIPS-HGN (Shen, Trevizan, and Thiébaux, 2020) learns a network, which generalizes across a domain. To enable this generalization between state spaces of different sizes, STRIPS-HGN uses internally a generalization from GNN to hypergraphs. In contrast to a normal graph, the edges in a hypergraph can have multiple start and multiple end points. Because STRIPS-HGN generalizes across all tasks within a domain, it is trained on a set of simple tasks. Thus, labeling states from these tasks is quick. In the original setup, it samples and trains for 90-1,200 seconds. To account for the computational resources required by our approaches, we carefully tune its parameters to benefit from the extended resource limits. We generate data for 10 hours and then train a single model for 2.8 hours on 4 cores. We sample training states from the moderate and hard tasks using regression random walks. The length n of the regression random walks is randomly chosen from the interval $\tilde{n} \leq n \leq \bar{n}$. We initialize $\tilde{n}$ to 50 and $\bar{n}$ to 500. Like Shen, Trevizan, and Thiébaux (2020) we execute on every sampled state A^* with the LM-Cut heuristic. We enforce a time limit of 30 minutes for the search. If the search

finds a solution, we use all states along the plan as training data together with their cost along the plan and training samples. As A^* with LM-Cut produces optimal plans, we know that all states are labeled with their true goal distance. Whenever the search finds a solution in less than 5 minutes, the sampled state was too easy. Thus, we increase the minimum walk length $\tilde{n} = (\tilde{n} + 3n)/4$. If the search times out, the state was too hard. Then, we decrease the maximum walk length $\bar{n} = (\bar{n} + n)/2$. We observe in the Blocksworld, Scanalyzer, and Transport domains that the original training setting performed better than ours. Thus, we use the original setting for those domains.

We additionally compare to well-established model-based planners. First, we use GBFS with the FF heuristic. This is also the teacher for our supervised learning heuristic. The FF heuristic is a delete relaxation heuristic, i.e., internally it uses a simplified version of the actions where all delete effects are removed. Mercury is a planner based on partial delete relaxation. It uses a heuristic, which removes only some delete effects. Furthermore, it uses a short-cut technique. This short-cut technique recognizes relaxed plans which successfully execute, or which can be repaired to execute successfully. Finally, we use the first phase of the LAMA planning system. This phase is a GBFS with 4 open lists. One open list for the FF heuristic, one for the LM-Count heuristic, one for the preferred operators of h^{FF}, and one for the preferred operators of LM-Count. Röger and Helmert (2010) showed that alternating between multiple open lists for selecting the next state to expand is beneficial. If one heuristic is uninformed at the current part of the state space, the other heuristic can continue to progress. Some heuristics compute preferred operators in addition to the heuristic value for a state. Those are actions, which the heuristic deems important (Hoffmann and Nebel, 2001). The preferred operators queue of the FF heuristic (resp. LM-Count) is also sorted by the FF heuristic (resp. LM-Count), but contains only states reached via actions marked as preferred operators.

8.2. Simple Machine Learning Models

Neural networks and deep learning are trending buzzwords. However, they are complex models and expensive to train. Simpler machine learning techniques require fewer data, require fewer computational resources, and are simpler to understand. Following Occam's Razor, if they work equally well, they should be preferred. Thus, we first evaluate if FFNs are needed at all.

As simple models, we use linear regression (LR) without regularization, random forests (RF), and support vector regression (Drucker et al., 1996, SVR) with *radial basis function* as kernel. An SVR fits a line to the data points by minimizing the L2 norm of the line and ensuring that the error for each data point is smaller than ϵ. As this is not always possible, there is a slack parameter C. The error of data points can be larger than ϵ, but this additional error is weighted by C, and added to the L2 norm. The larger C, the fewer impact has the L2 norm of the fitted line on the minimization. Large

	Reg	LR	RF	$SVR_{C=10}$	$SVR_{C=100}$
Reg	-	71	62	69	50
LR	7	-	21	26	14
RF	16	57	-	50	42
SVR_{10}	9	52	28	-	0
SVR_{100}	28	64	36	78	-

Table 8.2: Pairwise comparison between regression networks (Reg), linear regression (LR), random forests (RF), and support vector regression (SVR) with a C value of 10 and 100. For each moderate task, a model of each technique is trained using 35,000 samples and evaluated using the MSE on 5,000 samples. Each row indicates on how many tasks out of 78 a technique is better than the other techniques.

C values lead to overfitting. As FFN, we use our default setting for the progression heuristics, but with a single regression output.

We train a model for every moderate task. As the training time of SVRs scales cubically in the number of samples, we train all models with at most 35,000 samples. We train the simple models on all training samples. We train the FFN on 30,000 training samples and use the remaining 5,000 samples as validation data. We evaluate the models on another 5,000 samples using the mean squared error.

Table 8.2 shows how often a technique is better than another. The regression FFN is significantly better at predicting the right heuristic estimates than any other technique. Furthermore, it is not only more accurate, but also faster. Except for the linear regression model, evaluating the FFN is always faster than evaluating the other models. We note that the evaluation speed could partially be due to the libraries used. As conclusion, it is justified to use complex machine learning models like FFNs.

8.3. A Survey in Hyperparameter Space

We showed that it makes sense to use NN. Now, we study the hyperparameters to pick. It is computationally too expensive to generate enough data for all tasks. Thus, we restrict ourselves to the moderate tasks. For each task and parameter setting, we train ten models. Each model uses a different part of the sampled data for validation. For each task, we generated 200 test states. It is also too expensive to evaluate each of the ten models on each test state. Thus, we partition the test states into ten sets and evaluate each model on a different set. This still produces results, which are robust with respect to retraining the model and changing test states. To evaluate the quality of a model, we use it as heuristic in a GBFS to solve the test states. If it solves the test state within 30 minutes and 3.8 GB of memory, we count this as success. The

Domain	cls_{OH}	cls_{U}	reg
Blocksworld	93.4	97.2	65.3
Depots	87.7	76.2	77.3
Grid	44.8	93.2	71.0
Pipesworld-NT	84.3	89.6	82.0
Scanalyzer	96.2	94.6	80.3
Storage	14.5	95.5	98.5
Transport	92.2	99.1	88.9
Average	73.3	92.2	80.5

Table 8.3: Coverage (in %) of the progression heuristic with regression (reg), one-hot (cls_{OH}), and unary (cls_{U}) output encoding on the moderate tasks.

NPuzzle, Rovers, and VisitAll domain are especially difficult for our approach. In none of these domains, we learned meaningful heuristics. Thus, we exclude them for the hyperparameter evaluation and evaluate them at the end with the final parameter setting. We do not have the computational resources to evaluate the combination of all parameter choices. Thus, we evaluate each parameter choice individually and use the result of the previous evaluations for the next one. For each experiment, we report for each domain the percentage of test states solved. Remember that a domain has for each task 200 test states, and we have multiple tasks for each domain. The absolute average over all domains is calculated by equally weighting all domains.

Output Encoding. The output encoding of our FFN is one of the most fundamental architecture choices. To interpret the output of a classification network with unary encoding, we use a threshold of 0.01. We found this value in exploratory experiments. Table 8.3 shows the fractional coverage results.

The regression encoding is in general inferior to both classification encodings. Only twice it is better than the one-hot encoding and only twice better than the unary encoding. The one-hot encoding performs well in most domains, but there are two outliers where it does not work well at all. This reduces its average coverage drastically. Using a classification output *and* still encoding the numeric relationship between the classes performs best. With one exception, the unary encoding is always best or close to best. In the remaining experiments, we consider the unary encoding.

Architecture. Next, we evaluate possible choices in the network architecture. Increasing the number of hidden layers adds more trainable weights to the network. Thus, it can approximate a larger set of functions. But more weights make it harder to train and slower to evaluate. Our networks with one hidden layer evaluate on average 1,000 states per second. Our networks with three hidden layers evaluate around 370 states per

	Coverage				# Expansions		
Domains	0	1	3	5	1	3	5
Blocksworld	30.7	100.0	97.2	83.3	585	899	2122
Depots	70.7	80.6	76.2	75.9	499	182	129
Grid	41.8	94.0	93.2	47.8	14K	373	2761
Pipesworld-NT	70.9	88.9	89.6	74.8	818	697	699
Scanalyzer	49.0	88.2	94.6	83.2	360	173	160
Storage	100.0	100.0	95.5	55.0	7155	1917	25K
Transport	58.3	98.3	99.1	99.7	47K	2910	719
Average	60.2	92.9	92.2	74.3	-	-	-

Table 8.4: Coverage (in %) and median number of expansions of the progression heuristics with varying number of hidden layers on the moderate tasks.

		Activation	Dropout Rate		L2 Weight		
Domain	Base	ReLU	0.2	0.4	0.1	1	10
Blocksworld	97.2	100.0	95.7	90.9	0.0	0.0	0.0
Depots	76.2	77.6	81.2	77.8	0.2	0.2	0.2
Grid	93.2	78.2	77.0	72.2	0.0	0.0	0.0
Pipesworld-NT	89.6	84.2	89.3	90.1	7.2	7.2	7.4
Scanalyzer	94.6	96.5	81.8	73.8	15.4	15.4	15.4
Storage	95.5	99.5	56.0	31.5	0.0	0.0	0.0
Transport	99.1	89.6	99.9	99.9	0.0	0.0	0.0
Average	92.2	89.4	83.0	76.6	3.3	3.3	3.3

Table 8.5: Coverage (in %) of the progression heuristics with ReLU activation function on the hidden layers, dropout layers, and L2 regularization on the moderate tasks.

Domain	Init-State	Random-State	Entire-Plan	Random-State [# Plans]
Blocksworld	0.0	97.2	81.6	42.2
Depots	28.6	76.2	71.2	78.6
Grid	12.2	93.2	48.0	71.5
Pipesworld-NT	90.6	89.6	75.8	63.8
Scanalyzer	16.8	94.6	79.4	60.0
Storage	13.5	95.5	52.0	21.0
Transport	17.8	99.1	93.7	96.1
Average	25.6	92.2	71.7	61.9

Table 8.6: (Column 2–4) Coverage (in %) of the progression heuristics on the moderate tasks when using different sample selection strategies. (Right) Coverage (in %) on the moderate tasks of the random-state sample selection strategy when selecting the same number of samples as the entire-plan strategy.

second. Our networks with five hidden layers evaluate only 160 states per second.

Table 8.4 shows the coverage for networks with 0, 1, 3, and 5 hidden layers. Networks without hidden layers perform worst in almost all domains. They obtain an average coverage of just 60%. Networks with 1 or 3 hidden layers perform best with an average coverage of around 92%. In some domains the former is better, in other domains the latter is better. Networks with 5 hidden layers are again worse with only 74% average coverage. Table 8.4 also shows the median number of expansions per hidden layers and domain. Increasing the number of hidden layers from 1 to 3 reduces the required number of expansions in all except for one domain. The increased network complexity can be used to better approximate the perfect heuristic. Further increasing the number of hidden layers is beneficial in some domains, but is detrimental in others. In the later course, we train networks with 3 hidden layers, because they perform almost as good as the networks with 1 hidden layer, and they are much better informed.

Another architectural choice is the activation function used in the hidden layers. Switching from sigmoid to ReLU activation has a minor detrimental impact (see Table 8.5). Thus, we keep using the sigmoid activation function. Finally, we also add regularization to the network in the form of dropout layers or an L2 penalty for the weights. Neither of both performs consistently better than no regularization (see Table 8.5). Thus, we conclude our networks do not overfit in a manner that regularization prevents.

Data Distribution. Our final parameter set concerns the training data distribution. We proposed three strategies to select training states from sampled plans. Table 8.6 shows the average coverage for each strategy when training on the *same* number of states. The random state strategy clearly outperforms the other two. Thus, using uncorrelated states

Domain	P^+W^+	P^+W^-	P^-W^+	P^-W^-
Blocksworld	60.1	0.0	66.5	97.2
Depots	45.2	76.7	39.8	76.2
Grid	88.8	70.0	89.5	93.2
Pipesworld-NT	83.9	87.0	89.9	89.6
Scanalyzer	72.8	81.5	76.6	94.6
Storage	24.5	45.0	24.5	95.5
Transport	99.8	99.8	99.7	99.1
Average	67.8	65.7	69.5	92.2

Table 8.7: Coverage (in %) of the progression heuristics with and without pruning and sample weighting on the moderate tasks.

is indeed advantageous. The initial state strategy is clearly inferior. Taking a closer look at the data reveals that it performs best when there are states with small heuristic values in the data. Our assumption that we should focus on states far away from the goal is wrong. The network is now lacking states with low heuristic values. It remains future work to evaluate if a mixture of strategies improves performance.

Although the random state strategy performs best, the experiments hide that using only a single state per generated plan reduces the number of samples by a factor of 10–138. To generate the same number of samples, the random state strategy requires much more computational power. Thus, we train additional models with the random state strategy. This time, random states are only picked from those plans previously used as training data by the entire plan strategy. This simulates the performance of the random state strategy when the computational resources for sampling are a limiting factor. We see that the new models perform worse than the entire plan strategy. Thus, if we cannot generate enough plans to fill the memory with uncorrelated states, then using the entire plan is the best choice.

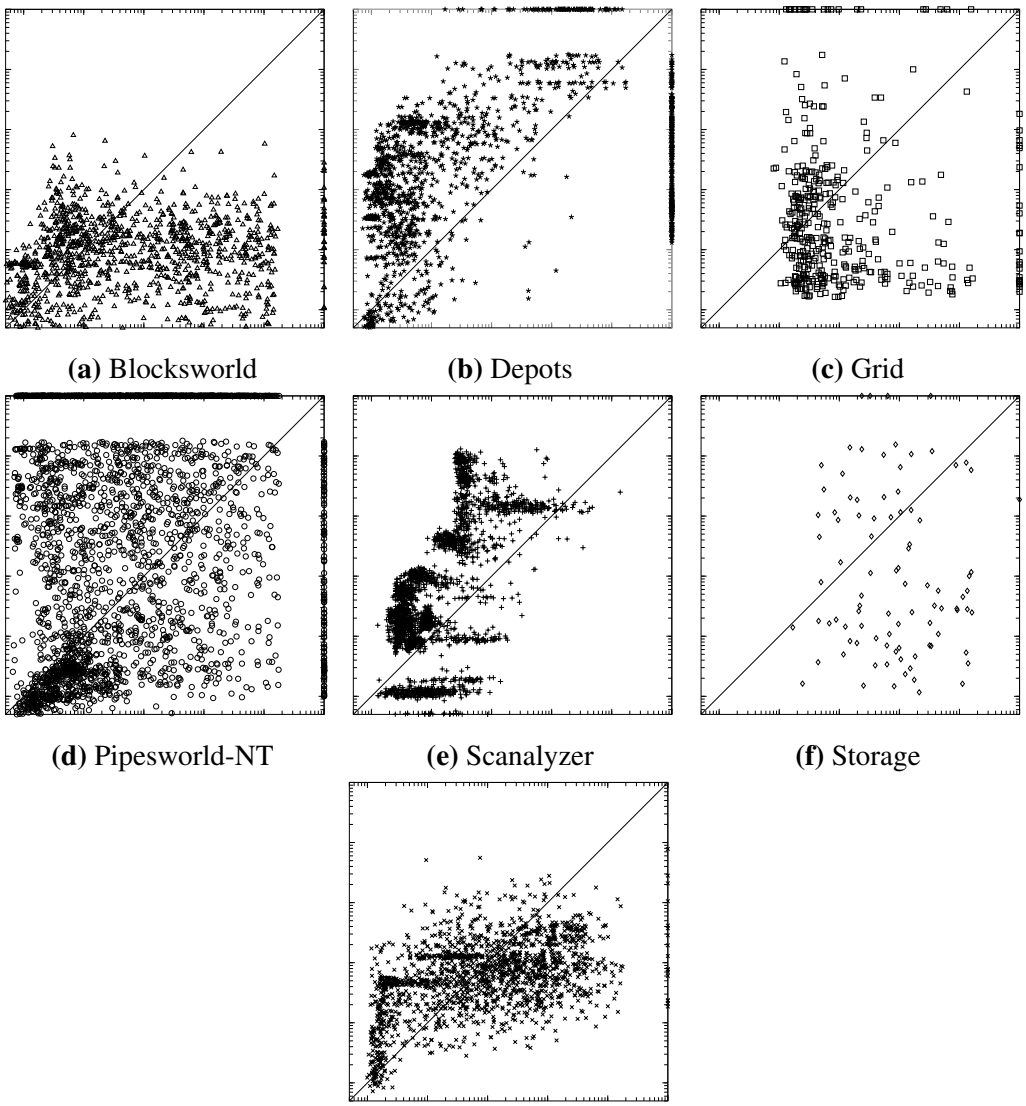
Table 8.7 shows the results when using pruning or weighting. If we prune duplicate states from the training data, this reduces the performance in half of the domains and in the other half it has almost no effect. There are much fewer states close to the goal than far away from the goal. Thus, most duplicates will have a low heuristic value. If we prune duplicates, states with low heuristic value become underrepresented. We observe that this is detrimental. This observation is also in line with our observation regarding the initial state strategy. We also observe that just weighting the samples such that all heuristic values are equally impactful during training does not work either. We know that for the high heuristic values, there are very few samples, sometimes only one. If each heuristic value has the same influence, then only a few states have almost all impact on the training outcome, and the network cannot generalize.

	Coverage					Median # Expansions				
Domain	h^P	h^{FF}	LAMA	Mercury	h^{P+}	h^P	h^{FF}	LAMA	Mercury	h^{P+}
Blocksworld	97.2	100.0	100.0	100.0	98.8	1381	12665	465	516	442
Depots	76.2	92.1	100.0	99.2	91.0	182	47731	8734	8746	65
Grid	93.2	93.2	100.0	100.0	94.0	493	2439	150	183	493
Pipesworld-NT	89.6	63.6	98.7	92.5	97.9	354	832	1676	138	103
Scanalyzer	94.6	97.8	100.0	100.0	98.2	269	482	208	217	97
Storage	95.5	94.5	96.0	97.0	98.0	4208	2089	24712	16055	4145
Transport	99.1	100.0	100.0	100.0	100.0	3002	4586	192	0	122
Average	92.2	91.6	99.2	98.4	96.8	1413	10118	5162	3693	781

Table 8.8: Coverage (in %) and median number of expansion for the progression heuristic (h^P), the teacher (h^{FF}), LAMA, Mercury, as well as the progression heuristic extended with the preferred operators of the FF heuristic (h^{P+}) on the moderate tasks.

	Median # Exp. Per Second				Median Runtime				
Domain	NN	h^{FF}	LAMA	Mercury	NN	h^{FF}	LAMA	Mercury	h^{P+}
Blocksworld	857	14536	19593	14875	1.7	0.8	0.1	0.8	0.9
Depots	530	1490	4792	3749	0.4	17.9	1.1	3.4	0.3
Grid	125	1810	4525	1739	4.1	1.5	0.2	4.9	3.2
Pipesworld-NT	295	752	3563	1640	1.2	1.5	0.6	2.7	0.8
Scanalyzer	183	38	1403	725	1.4	5.7	0.7	8.3	1.1
Storage	51	722	7309	5283	83.9	3.0	3.4	4.5	39.5
Transport	458	799	3415	0	8.0	6.3	0.2	2.8	1.1
Average	357	2940	6616	4116	14.4	5.2	0.9	3.9	6.7

Table 8.9: Median number of expansions per second and median runtime (in seconds) for the progression heuristic (h^P), the teacher (h^{FF}), LAMA, Mercury, as well as the progression heuristic extended with the preferred operators of the FF heuristic (h^{P+}) on the moderate tasks.



(a) Blocksworld **(b)** Depots **(c)** Grid

(d) Pipesworld-NT **(e)** Scanalyzer **(f)** Storage

(g) Transport

Figure 8.1: Runtime comparison between the progression heuristic and the teacher on the moderate tasks. Every test state is a data point. The runtimes of the progression heuristic is on the x-axis, the runtimes of the teacher is on the y-axis. Data points left of the diagonal indicate a better runtime for our progression heuristic.

Performance Comparison. Now that we have our final parameter setting, we compare GBFS with our heuristic against some state-of-the-art model-based planners. Table 8.8 and Table 8.9 show the results. The first obvious question is whether our heuristics solve more tasks than the teacher, which generated the data. In Pipesworld-NoTankage, we perform better than our teacher. In Depots, we perform worse. In the other domains, the results are approximately the same. If we take a look at the median number of expansions for each domain, which we use as a proxy for the informedness of the heuristics, we see that our trained heuristics are significantly better informed. Why are they not solving more tasks if they are better informed? We observe that our NN heuristics require a factor of 2–36 more time to evaluate a single state than our teacher. Often this additional time offsets the advantage of being better informed. Figure 8.1 provides a comparison between the runtime of GBFS with h^P and our teacher. Every test task is a point in the scatter plot. In Blocksworld and Storage, the teacher is in general faster. In Transport, our heuristic is faster on simple tasks, but on average, the teacher is faster. For Pipesworld-NoTankage the picture is not that clear, but it is inverted. The teacher is faster on simpler tasks, and our heuristic is faster on the harder tasks. For Scanalyzer and Depots our heuristic is fastest, and in Grid, the picture is mixed. These results can also be seen in the summarized median runtime values of Table 8.9. We conclude that our heuristic is better informed, but too slow to evaluate on a *single CPU core*.

And how do we compare against other state-of-the-art planners like LAMA and Mercury? Table 8.8 shows that our learned heuristics solve in general fewer tasks than those planners. It should be noted that both of them are not just GBFS with a heuristic, but incorporate other techniques. Mercury has a technique, which detects if it can extend a delete relaxed plan to a valid plan and LAMA combines two heuristics and their preferred operators (PO) using four open lists (also called *queue*). For a fair comparison, we also evaluate our heuristic in a dual queue approach. The second queue is filtered by the preferred operators of the FF heuristic. Whenever we extend a method with a second queue using the preferred operators of the FF heuristic, we extend its abbreviation by a "+". The dual queue approach improves the coverage in all domains. In two domains, we now solve more tasks than Mercury. Furthermore, we are now faster than Mercury in five out of seven domains. Although this extension shrinks the gap to LAMA, LAMA is still supreme in all domains. It solves more tasks, and it is faster. Our learning-based heuristic is not yet ready to be used in the real world.

Until now, we skipped the NPuzzle, Rovers, and VisitAll domains, because preliminary experiments did not learn meaningful heuristics on them. Table 8.10 shows the final test results for those three domains. Even with the optimized parameter setting, our heuristics do not work on these domains. They are less informed than even the teacher and slower to evaluate. Consequently, they solve only a few test states. A reason for NPuzzle and VisitAll could be that the teacher also performs badly in those domains. The teacher produces excessively large plans, sometimes with more than 2,500 actions, which leads to samples with low quality labels. A solution could be to use a differ-

Domain	Coverage NN	Coverage h^{FF}	# Expansions NN	# Expansions h^{FF}	# Exp. Per Sec. NN	# Exp. Per Sec. h^{FF}	Runtime NN	Runtime h^{FF}
npuzzle	0.0	97.3	–	–	–	–	–	–
rovers	53.0	73.9	3543	1165	43	1800	82.8	0.5
visitall	0.2	94.1	1065K	93K	893	35354	1251.5	2.6
Average	17.7	88.4	–	–	–	–	–	–

Table 8.10: Coverage (in %), median number of expansion, median number of expansions per second, and median runtime (in seconds) for the progression heuristic (h^P) and the teacher (h^{FF}).

	Coverage 100	Coverage 75	Coverage 50	Coverage 25	Coverage 2.5	Median # Expansions 100	Median # Expansions 75	Median # Expansions 50	Median # Expansions 25	Median # Expansions 2.5
Blocksworld	97.2	96.2	94.2	87.8	45.3	409	265	230	364	93K
Depots	76.2	89.3	90.6	90.8	82.8	181	78	94	150	883
Grid	93.2	64.2	70.8	70.8	72.5	367	1005	1266	1302	1948
Pipesworld-NT	89.6	95.4	93.6	93.1	78.4	462	121	136	215	3501
Scanalyzer	94.6	95.1	93.2	82.9	57.4	86	50	45	45	208
Storage	95.5	18.5	59.5	32.5	1	626	222K	37K	228K	-
Transport	99.1	99.8	99.0	97.7	94.4	2686	146	165	248	910
Average	92.2	79.8	85.8	79.4	61.7	688	32K	6K	33K	-

Table 8.11: Coverage (in %) and median number of expansions for the progression heuristic trained on fractions of the available training data.

ent teacher, e.g., a bounded suboptimal teacher. Indeed, exploratory experiments with weighted A^* as teacher increased the coverage from 0% to 40%. We learn that if the quality of the labels is too low, then the network does not learn a good heuristic.

We used a generous amount of time to generate our training samples. This computational cost should amortize if we use our learned heuristic frequently. Nevertheless, is it necessary to generate this much training data? Table 8.11 shows that in most domains a fraction of the training data is sufficient to learn high quality heuristics. In many domains, 50% or even 25% of the training samples suffice. In Transport the heuristic quality drops only slightly when training on 2.5% on the training data. In other words, for Transport we do not need to sample data for 400 hours, but only for 10 hours. In a concrete setting, the amount of training data to sample should be adapted to the domain.

	Moderate Tasks									
	w/o Validation			with Validation						
Domain	h^{GD}	h^{SE}	h^{AVI}	h^{GD}	h^{SE}	h^{AVI}	h^{P}	h^{HGN}	h^{FF}	LAMA
blocks	0	0	0	18	0	0	80	**100**	99	100
depots	32	18	**44**	60	33	55	**90**	0	98	100
grid	**100**	**100**	51	**100**	**100**	51	93	0	96	100
npuzzle	**27**	0	1	**28**	0	1	0	0	98	100
pipes-nt	36	**51**	21	58	68	50	**92**	8	82	99
rovers	**36**	15	34	**48**	22	45	26	14	84	100
scanaly.	33	60	**67**	33	71	67	**83**	11	98	100
storage	**89**	61	67	**89**	58	70	24	0	48	38
transport	**84**	80	70	**100**	**100**	88	99	95	98	100
visitall	**17**	0	0	55	0	0	0	**100**	93	100

Table 8.12: Coverage (in %) comparison between our goal-distance estimator (h^{GD}), our search space size estimator (h^{SE}), our approximate value iteration heuristic (h^{AVI}), our progression heuristic (h^{P}), STRIPS-HGN (h^{HGN}), the teacher (h^{FF}), and LAMA on the moderate tasks with validation and between h^{GD}, h^{SE}, h^{AVI} on the moderate tasks without (w/o) validation.

8.4. Regression Heuristics and Comparisons

To circumvent the computational bottleneck of the data generation, we proposed three methods, which use regression random walks to sample states, and label the states using the current state of the network. We now evaluate these three methods and compare them to STRIPS-HGN (h^{HGN}) and LAMA.

Because our new approaches avoid the data generation bottleneck, we also include the hard tasks in our evaluation. We previously showed that our NN are slow at evaluating a state on a single core. The evaluations could be sped up with parallel computing power, but the model-based planners do not support parallel computing. To make up for this disadvantage, we set a search time limit of 10 hours for all planners. This allows the NN heuristics to profit from their better informedness without being punished for running on a single core. Consequently, we use only 50 instead of 200 test states for each task. For the hard tasks, we generate those test states in the same manner as before: progression random walks from an initial state with a walk length of 200.

Validation. Preliminary experiments showed that the performance of our new approaches is brittle. For every task, we sample ten validation states from the same distribution as the test states and evaluate a trained model on them. If GBFS using those models finds plans for less than eight validation states, then we retrain the model. We retrain at most three times.

	Hard Tasks with Validation						
Domain	h^{GD}	h^{SE}	h^{AVI}	h^{P}	h^{HGN}	h^{FF}	LAMA
blocks	0	0	0	0	**50**	62	97
depots	8	4	13	**35**	0	36	83
grid	88	**95**	70	60	0	53	100
npuzzle	0	0	0	0	0	33	86
pipes-nt	23	19	8	**49**	0	27	69
rovers	3	1	**6**	2	0	14	100
scanaly.	3	0	**61**	60	0	98	100
storage	**27**	13	16	0	0	14	12
transport	0	0	**2**	0	0	0	93
visitall	28	0	0	0	**100**	74	100

Table 8.13: Coverage (in %) comparison between our goal-distance estimator (h^{GD}), our search space size estimator (h^{SE}), our approximate value iteration heuristic (h^{AVI}), our progression heuristic (h^{P}), STRIPS-HGN (h^{HGN}), the teacher (h^{FF}), and LAMA on the hard tasks with validation.

Table 8.12 shows the effect of the validation for the moderate tasks. For all our heuristics and most domains, using validation increases the coverage significantly. For example, in the Depots domain, the coverage of h^{GD} increases from 31.7% to 60.3%. Thus, we use the validation approach for all further learning-based experiments. To be fair to the other learning-based heuristics, we train 10 STRIPS-HGN models for each domain and 10 progress heuristic models for each task and evaluate the performance of those models on the same validation states. Then we pick the model which solves most of the validation states. If there is a tie, we prefer the network which fastest solves the validation states.

Performance. Table 8.12 shows the coverage on the test states for the moderate tasks. If we compare our regression heuristics, we observe that no approach dominates any other approach. h^{GD} is better than the other two in 8 domains, h^{SE} is best in 4 domains, and h^{AVI} is closest to the best in 3 domains. Adding our progression heuristic to the mix, it performs better than our other approaches in 4 domains, but worse in 6. Noteworthy, h^{GD} is able to learn something in all three domains where h^{P} cannot learn reasonable heuristics. STRIPS-HGN learns heuristics, which generalizes across the whole domain. It excels in Blocksworld and VisitAll and solves *all* test samples. In Transport, it also performs very well. In all other domains, it solves no or almost no task. At the core of STRIPS-HGN is a hypergraph. We identify this hypergraph as a main issue in the other domains. Either the hypergraph is so large that it exceeds the memory limits (Depots, Storage, Grid) or it is still so large that the network is very slow to evaluate. In the Blocksworld and VisitAll domains where STRIPS-HGN excels,

we observe that the hypergraph stays small (less than 1,000 nodes and less than 1,500 edges). If we compare all learning-based heuristics, we observe that they are highly complementary. No heuristic dominates the others. It depends on the domain, which method performs best. Due to the issue with the hypergraph size, learning a heuristic which generalizes only over the states of a task is often superior to learning a heuristic which generalizes across the domain.

If we add GBFS with the FF heuristic and LAMA to the comparison, we see that model-based approaches are still superior. Especially LAMA performs well on all domains. Only in the Storage domain, learning-based heuristics are able to solve more tasks than LAMA and h^{FF}. LAMA solves 38% of the test samples, GBFS with h^{FF} solves 48%, h^P and STRIPS-HGN are worse, but all three regression heuristics exceed the model-based heuristics. h^{GD} solves even 89% of the moderate Storage tasks.

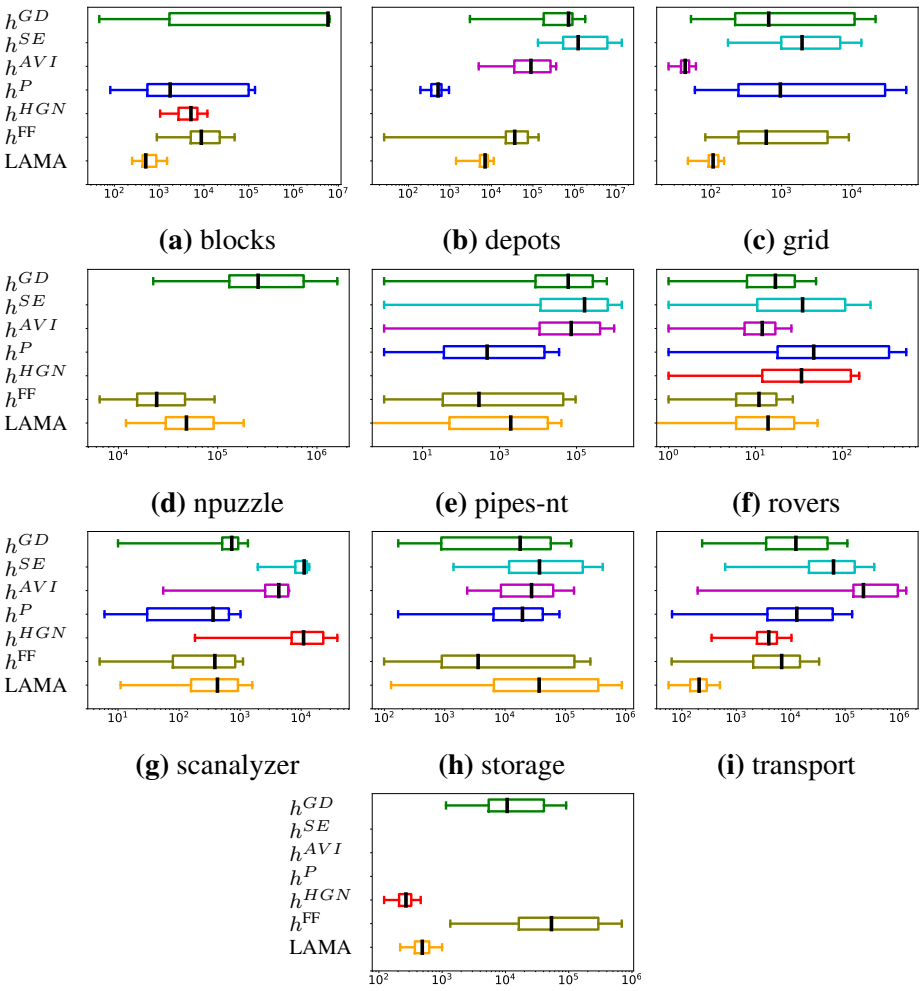
Figure 8.2 compares the expansions required by the different methods. We see again that all network based heuristics are highly complementary. Sometimes, one is better. Sometimes, another one is better. LAMA is often good, and we already know that it quickly expands states. Thus, LAMA performs well in most domains.

As we used a long time limit of 10 hours, we also evaluate how the coverage changes over time. Figure 8.3 shows that coverage superiority persists over time. For every domain holds if a method is better than another one after 30 minutes, then it is also better after 2 hours. On the other hand, LAMA quickly expands states. Thus, it solves a state quickly or reaches the memory limit and terminates without a solution. On the other hand, the neural network based heuristics are slower to evaluate and still solve tasks after multiple hours have passed. Nevertheless, even the neural network heuristics rarely solve a task after five hours. Cutting the maximum search time in half would not have changed the results significantly.

Finally, let us take a look at the coverage results for the hard tasks (see Table 8.13). The hard tasks are more difficult for *all* techniques. The coverage numbers decrease in general, but the results are qualitatively the same. STRIPS-HGN still excels in VisitAll, but solves only half the Blocksworld states. It fails on all other domains. Our progression heuristic still generates enough training data to solve some tasks, but fails in most tasks. The coverage numbers of our regression heuristics also decreased, but in some domains they still work. As with the moderate tasks, our regression heuristics still exceed the coverage of h^{FF} and LAMA in Storage. As before, we improve our heuristics with a second queue, which uses the preferred operators of the FF heuristic. They almost always improve the performance significantly. Our advantage in storage increases, and our disadvantage in the other domains decreases, but LAMA is still superior in nine out of ten domains.

	Hard Tasks with Validation									
domain	h^{GD}	h^{GD+}	h^{SE}	h^{SE+}	h^{AVI}	h^{AVI+}	h^{P}	h^{P+}	h^{FF}	h^{FF+}
blocks	0	+ 0	0	+ 0	0	+ 0	0	+ 0	62	+ 9
depots	8	+16	4	+13	13	+ 1	35	+22	36	+31
grid	88	+11	95	+ 4	70	+10	60	+19	53	+24
npuzzle	0	+ 0	0	+ 0	0	+ 0	0	+ 0	33	- 2
pipes-nt	23	+ 6	19	+10	8	+ 3	49	+10	27	+37
rovers	3	+34	1	+ 5	6	+ 0	2	+35	14	+82
scanaly.	3	+ 5	0	+ 3	61	+ 6	60	+29	98	+ 1
storage	27	+ 5	13	+ 6	16	+ 6	0	+ 0	14	- 5
transport	0	+ 0	0	+ 0	2	+30	0	+ 1	0	+26
visitall	28	+ 4	0	+ 0	0	+ 0	0	+ 0	74	+ 4

Table 8.14: Coverage (in %) comparison between our goal-distance estimator (h^{GD}), our search space size estimator (h^{SE}), our approximate value iteration heuristic (h^{AVI}), our progression heuristic (h^{P}), and the teacher (h^{FF}), on the hard tasks with validation without and with ("+") the preferred operators of the FF heuristic.



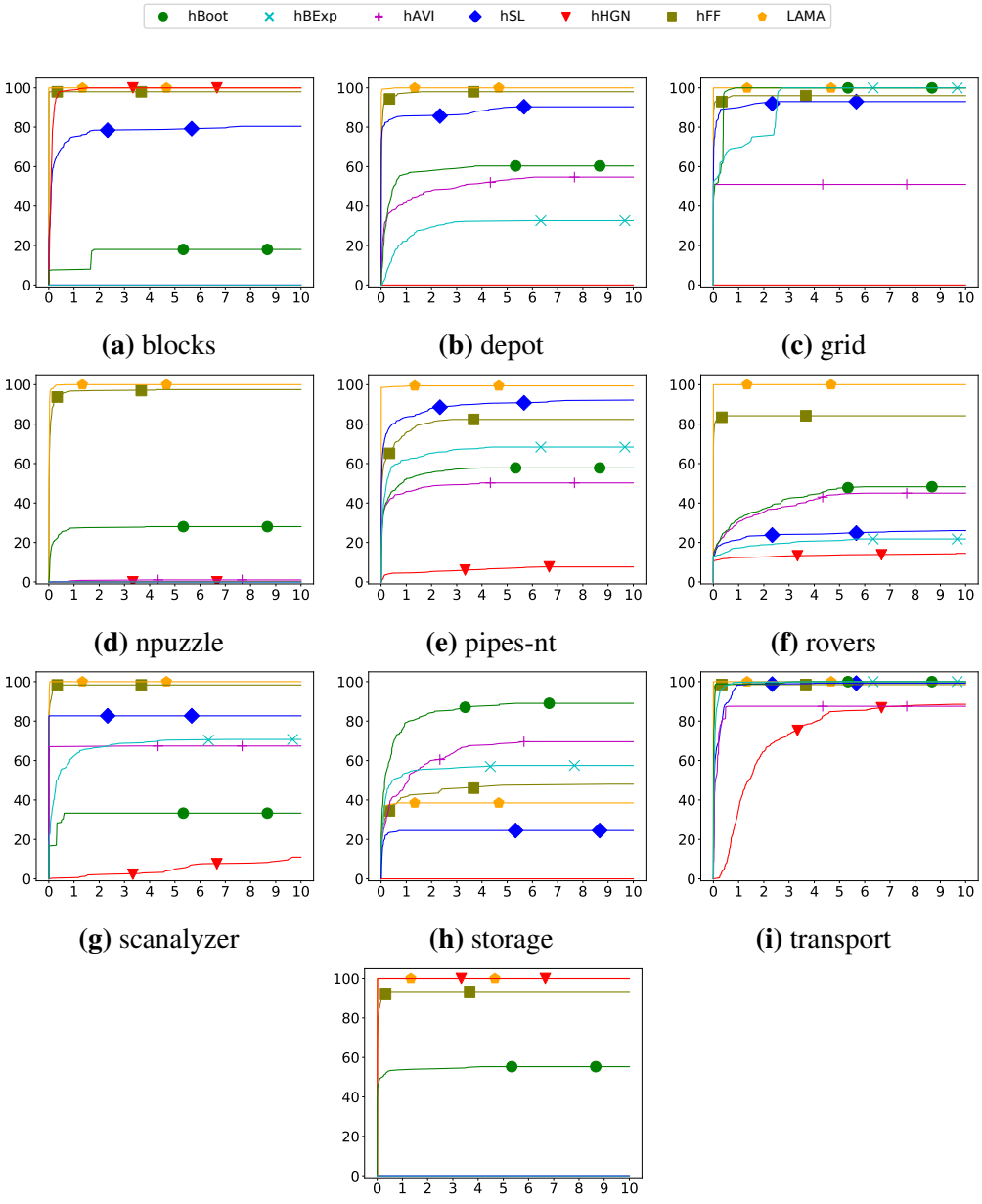
(a) blocks **(b)** depots **(c)** grid

(d) npuzzle **(e)** pipes-nt **(f)** rovers

(g) scanalyzer **(h)** storage **(i)** transport

(j) visitall

Figure 8.2: Expansion comparison between our goal-distance estimator (h^{GD}), our search space size estimator (h^{SE}), our approximate value iteration heuristic (h^{AVI}), our progression heuristic (h^{P}), STRIPS-HGN (h^{HGN}), the teacher (h^{FF}), and LAMA on the commonly solved moderate tasks. Algorithms which solved fewer than 10% of the tasks in a domain are skipped, to keep the number of the commonly solved tasks sufficiently large. The black line in the middle indicates the median, the boxes the interval between the 25 and 75 percentile. the whiskers extent to the 5 and 95 percentile.



(a) blocks **(b)** depot **(c)** grid

(d) npuzzle **(e)** pipes-nt **(f)** rovers

(g) scanalyzer **(h)** storage **(i)** transport

(j) visitall

Figure 8.3: Coverage (in %) over time (in hours) for our goal-distance estimator (h^{GD}), our search space size estimator (h^{SE}), our approximate value iteration heuristic (h^{AVI}), our progression heuristic (h^{P}), STRIPS-HGN (h^{HGN}), the teacher (h^{FF}), and LAMA on the moderate tasks.

9. Summary and Future Work

We presented four approaches to train neural networks for reset problems. The first uses progression random walks to generate the training states. However, it is expensive to label to states with a teacher. The latter three circumvent the expensive labeling by using approximate value iteration or by slowly scaling the difficulty of the samples and using the network itself as teacher heuristic. We observe that the performance of the three regression walk heuristics is brittle. We introduce a validation approach to increase robustness. We identified the effect of important hyperparameters on our NN and compare our approaches against STRIPS-HGN, a state-of-the-art learning-based planner, and some model-based planners on tasks of interesting difficulties. We show that all learning-based approaches are highly complementary. They excel in some domains and perform badly in others. It is a priori unclear how they perform in a new domain. In general, all learning-based heuristics perform worse than LAMA, a state-of-the-art model-based planner. A notable exception is that all our regression based heuristics solve more tasks than LAMA in the Storage domain. Until now, this was only achieved once by Karia and Srivastava (2021) on the Spanner domain. Furthermore, we want to remember that the neural networks were executed on a single CPU core, although their computations can easily be parallelized and sped up using multiple cores or even a GPU.

A first future step speeds up the network evaluation. Agostinelli et al. (2019) showed that evaluating multiple states at once improves the evaluation speed. Furthermore, we can adapt the implementation to GPUs.

Secondly, we can introduce better sampling techniques. We can replace the independent random walks by a search like algorithm with duplicate detection. Every new random walk starts at the same goal and walks over the same few states close to the goal. The unnecessary work of generating the same states multiple times will be skipped. Furthermore, identifying duplicates prevents us from choosing actions which reverse previous actions. Thus, leading us deeper in the state space. As we know how many steps we used to generate a sample, we have an upper bound on the goal distance. We could use this upper bound as label like Yu, Kuroiwa, and Fukunaga (2020) or at least use it to label a state if we are unable to find a plan for it with GBFS.

In satisficing planning, it is not relevant to predict an accurate goal distance or goal cost, but it suffices if the heuristic produces a similar state order as the perfect heuristic. Thus, we can also train a network by showing it two states and telling it which one is closer to the goal. Learning to rank states might be an easier task than predicting the correct goal distance. We implemented a prototype of such an approach using the

DirektRanker architecture of Köppel et al. (2019). The DirektRanker has one main network, which accepts a state and transforms it into a latent representation. To evaluate if one state is closer to the goal than another one, it transforms both states into the latent representation and then compares the representations to make its decision. With minor adaptions, we can split the network after training such that we can evaluate it on a single state and interpret the output as heuristic. In contrast to our definition, this heuristic can output negative values. As we use those values only to order states, this has no consequences.

Finally, we consider training a network, which generalizes across a domain. This allows us to train the network on simple tasks where we can generate the true labels. Most previous approaches do this with GNN (Shen, Trevizan, and Thiébaux, 2020; Rivlin, Hazan, and Karpas, 2020). Karia and Srivastava (2021) use canonical abstractions instead. We can show that their fragment of canonical abstraction is a subset of the description logic we use in later parts of this thesis. We refer the interested reader to Section A.1 of the appendix. Our idea is to build upon our experience with description logic and learn automatically description logic features on which the state ranking builds its decisions. GNNs and description logic can express similar functions (Barceló et al., 2020). The advantage of GNNs is that they learn by themselves what is important in the input, but we observed that the size of the underlying graphs make GNNs hardly usable on large tasks. If we use description logic, we do not have the scalability issue. Instead, we need a good procedure to build meaningful inputs for the network.

Part II.

Learning Portfolios

For this part, I collaborated with the following people (listed in alphabetical order):
Jie Chen, Siyu Huo, Michael Katz, Tengfei Ma, Horst Samulowitz, Jendrik Seipp, Silvan Sievers, Shirin Sohrabi

10. Introduction to Portfolios

Solving classical planning tasks is difficult, EXPSPACE-complete difficult (Erol, Nau, and Subrahmanian, 1995). Thus, the planning community developed many sophisticated techniques to solve planning tasks. While the community constructed new planners, we observed that no single planner dominates all other planners. In fact, even though modern planners perform better and better on our benchmark sets, older planners still have their unique characteristics, which are useful for some tasks (Roberts and Howe, 2009). Prior work observed that given a time limit a planner solved a task either quickly or it exceeds the time limit (Helmert et al., 2011). We also observe that most tasks in our experiments are solved quickly or not at all (see Figure 13.1). Consequentially, the idea emerged that we can combine multiple planners to add their strengths. We call the combination of multiple planners into a single planning system a *portfolio* (Vallati, 2012). A portfolio is again a planner.

The idea of portfolio planners opens a new research direction. *Which planners* should be combined? *How* should they be combined? *Which properties* influence the best combination of planners?

The first emerging class of portfolios executes sequentially a fixed schedule of planners (Helmert et al., 2011; Núñez, Borrajo, and Linares López, 2014; Seipp et al., 2012; Seipp, Sievers, and Hutter, 2014a,b,c; Seipp, 2018a,b). The schedule is pre-computed and does *not* adapt to the task to solve. We call such a portfolio *offline portfolio*.

Definition 10.1. Offline Portfolio
Let $P = \{P_1, \dots, P_n\}$ be a set of planners and T be a time limit. An offline portfolio *is a schedule $S = \langle\langle p_1, t_1\rangle, \dots, \langle p_k, t_k\rangle\rangle$ of planner $p_i \in P$, time limit $t_i \in \mathbb{R}^+$ pairs with $\sum_{i=1..k} t_i \leq T$.*

Fast Downward Stone Soup is an example offline portfolio (Helmert et al., 2011). Given a set of planners P and a set of training tasks $\mathcal{T}$, it evaluates the performance, e.g., coverage, of every planner on every task. It splits the time limit T into equally sized fragments and initially assigns every planner zero seconds of runtime. Iteratively, it assigns one additional time fragment to the planner which improves the overall performance of the portfolio the most. Once all time fragments are assigned, the final schedule is constructed. Ties are broken using an arbitrary preference order between the planners.

Offline portfolios have the advantage that they are easy to execute and require a negligible overhead during execution, but they do not adapt to the current situation. If we want to solve a specific task, it is beneficial to adapt the schedule for that task. If we

know that some planners will perform poorly on the given task, we can remove those planners from the schedule and assign longer runtimes to the other planners. We call a portfolio which adapts its schedule to the current situation an *online portfolio* (Roberts and Howe, 2006; Cenamor, de la Rosa, and Fernández, 2013, 2014, 2016, 2018; Seipp et al., 2015).

Definition 10.2. Online Portfolio
Let $P = \{P_1, \ldots, P_n\}$ be a set of planners and T be a time limit. Given a task Π and some history $\mathcal{H}$, an online portfolio *is a function $s(\Pi, \mathcal{H}) = \langle p, t \rangle$ which predicts the next planner $p \in P$ to run and a time limit t for this planner.*

A simple online portfolio predicts a performance for each planner in the portfolio and selects the planner with the best predicted performance. As almost all non-portfolio planners are sequential algorithms, most competitions also execute planners only on a single core. Thus, most portfolios, as well as ours are sequential portfolios.

In the last years, dynamic algorithm configuration (DAC) was introduced for heuristic search (Speck et al., 2021; Biedenkapp et al., 2022). DAC executes a single search, but its parameters can change on the fly. For example, if the controller realizes that the current heuristic is not the best choice anymore, it changes the heuristic without starting a new search. Many of the planners in our portfolio differ only by their heuristic. For those planners, using DAC is more general than an online portfolio. We can switch at any point during search *and* keep the current search progress. This ability comes with an overhead. Every optional heuristic requires bookkeeping. This costs time and memory. The more heuristics DAC uses, the more expensive it becomes. Portfolios cannot switch between planners and keep the search progress, but they require no bookkeeping. Furthermore, portfolios allow combining arbitrary planners. For DAC, this is difficult, but not necessarily impossible, to combine fundamentally different planners, like a heuristic search planner and a SAT based planner.

In our work, we will train sequential online portfolios.

11. Competitive Online Portfolios

A few online portfolios for planning exist. All of them use a set of complicated, handcrafted features (Roberts and Howe, 2006; Cenamor, de la Rosa, and Fernández, 2014, 2016, 2018). Given a planning task, they extract the values for those features and use them as input to their model. Handcrafting features is problematic for two reasons. First, the selection of features introduces a bias for the model. Secondly, we can forget important features. Thus, we asked ourselves, *can we reduce the bias of handcrafted features? Can we reduce the burden of coming up with good features? And can we solve the problem of forgetting important features?*

The answers to all these questions is "yes". By changing the encoding of the planning task, we can directly apply machine learning techniques to the task without manually extracting features. That means, the machine learning model learns the features and how to weight them by itself.

In an initial endeavor, we show how to convert a planning task to a graph and then to an image and use this as input for a CNN. The CNN automatically extracts features from the images and predicts a single planner to run. The conversion from graph to image is ad hoc, loses information, and still introduces a human bias. Thus, we improve the approach and feed the graphs directly to a GNN. It rarely happens, but it can happen that our models select the wrong planner. We enable our portfolio to reconsider its first choice and *optionally* switch to a second planner. The choice of the second planner depends only on the first chosen planner. We do not yet include additional features from the run of the first planner.

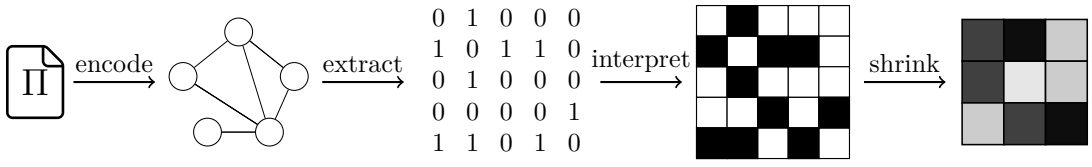


Figure 11.1: Workflow to convert a planning task Π into an image. First, the task is encoded as graph. Then, its adjacency matrix is extracted and interpreted as image. Finally, this image is postprocessed.

11.1. Graph Representations

Let $P = \langle P_1, \ldots, P_n \rangle$ be the set of planners we include in our online portfolio. We train our online portfolios on the data set $D = \{\langle \Pi_1, \vec{y}_1 \rangle, \ldots, \langle \Pi_k, \vec{y}_k \rangle\}$ where Π_i is a $PDDL$ planning tasks, and $\vec{y}_{ij}$ indicates the time the planner P_j requires to solve task Π_i. Most machine learning techniques do not accept a planning task as input. Thus, all previous online portfolios settle for a set of handcrafted features, which describe a planning task. They extract for each task Π_i the value of their features. To remove the need of handcrafting features, we encode a $PDDL$ task in such a way that the machine learning model can parse it. If this encoding is lossless, then the model parses the full task and identifies on its own the important features.

Loreggia et al. (2016) solve the input representation question for SAT and CSP problems by taking the textual description of a task, reading it character by character, interpreting every character as a pixel where its ASCII code represents its gray-scale color, and arranging the pixels in a grid. Then, they scale the resulting image down to a fixed size. The image can easily be processed by a CNN. Inspired by this, we set up the pipeline (see Figure 11.1) for our first online portfolio. First, we encode a task as graph using either the *abstract structure graph* (Sievers et al., 2019b, ASG) or the *problem description graph* (Pochter, Zohar, and Rosenschein, 2011, PDG). These graphs contain meaningful information about the task. Then we extract the adjacency matrix of the graphs and interpret them as black and white images. Finally, we apply some post-processing (see Section 11.3)

Abstract Structure Graph. The ASG is used to find structural symmetries and encodes a $PDDL$ tasks. Because the ASG encodes a lifted planning formalism, we call it a *lifted encoding*. An abstract structure is an inductively defined concept.

Definition 11.1. Abstract Structure (Sievers et al., 2019b)
Let T be a set of symbol types*, S be a set of* symbols*, and $t : S \mapsto T$ be a function, which associates every symbol with a type.*

- *A symbol $s \in S$ is an abstract structure.*
- *A tuple $A = \langle A_1, \ldots, A_n \rangle$ of abstract structures A_i is an abstract structure.*
- *A set $A = \{A_1, \ldots, A_n\}$ of abstract structures A_i is an abstract structure.*

We can easily turn any abstract structure into a graph.

Definition 11.2. Abstract Structure Graph (Sievers et al., 2019b)
Let A be an abstract structure over a set of symbols S with symbol types T and a function $t : S \mapsto T$. Let $G = \langle V, E \rangle$ be its abstract structure digraph.

- *If $A \in S$, then V contains a single node n_A for the symbol A and E is empty.*

- *If $A = \langle A_1, \ldots, A_n \rangle$, then V contains a main node n_A. Furthermore, G contains the nodes and edges for the ASG of $A_1, \ldots, A_n$ and an auxiliary node $n'_{A,i}$ for every structure A_i. Let $n_{A,i}$ be the main node for the ASG of A_i. E contains edges $n'_{A,i} \to n_{A,i}$. Finally, E contains an edge $n_A \to n'_{A,1}$ and for $1 \leq i < n$ an edge $n'_{A,i} \to n'_{A,i+1}$.*
- *If $A = \{A_1, \ldots, A_n\}$, then V contains a main node n_A. Furthermore, G contains the nodes and edges for the ASG of $A_1, \ldots, A_n$. Let $n_{A,i}$ be the main nodes for the ASG of A_i. For every A_i, there is an edge $n_A \to n_{A,i}$.*

If multiple sub-structures use the same symbol $s \in S$, then N contains exactly one node n_s for the symbol s. The ASG is acyclic.

Additionally, we define a node coloring function $color$. If $A \in S$, then $color(A) = t(A)$; if $A = \langle A_1, \ldots, A_n \rangle$, then $color(A) = tuple$; if $A = \{A_1, \ldots, A_n\}$, then $color(A) = set$; and if A is an auxiliary node, then $color(A) = auxiliary$.

Now, we show how to recursively construct an abstract structure for a $PDDL$ task $\Pi = \langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_\text{I}, \delta \rangle$ and how to derive a graph. For simplicity, we skip the notion of functions and axioms of a $PDDL$ task. We refer to Sievers et al. (2019b) for the complete definition. The abstract structure for a planning task has the symbol types

$$T = \{Object, Variable, Predicate, Negation\} \cup \mathbb{N}$$

and the symbols

$$\begin{aligned} S = &\{s_o \mid o \in \mathcal{O}\} \cup \{s_{a,v} \mid \langle V, pre, eff \rangle \in \mathcal{A}, v \in V\} \cup \\ &\{s_{a,v} \mid \langle V, pre, eff \rangle \in \mathcal{A}, \langle V^i : cond^i \triangleright eff^i \rangle, v \in V^i\} \cup \{s_p \mid p \in \mathcal{P}\} \cup \\ &\{s_a \mid a \in \mathcal{A}\} \cup \{s_\neg\} \cup \mathcal{N} \end{aligned}$$

where $\mathcal{N}$ is the finite set of all costs $c \in \mathbb{N}$ which are assigned by the *cost* function to an action. The construction rules are:

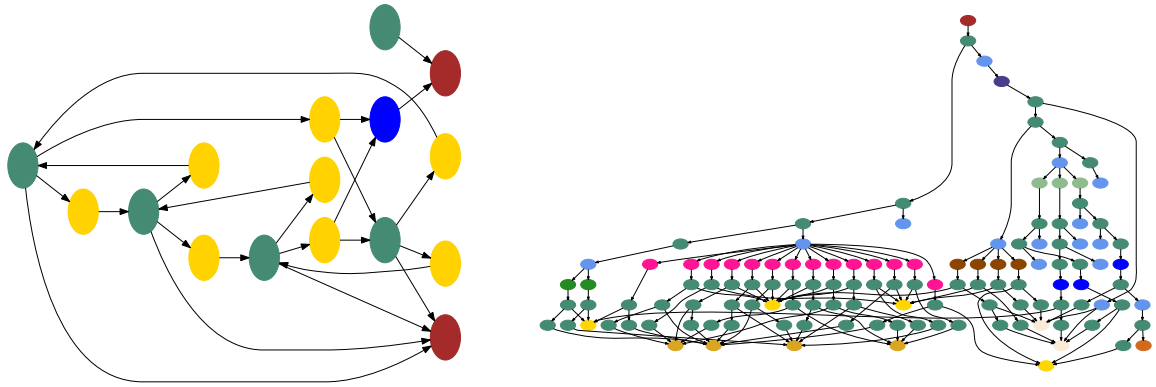
- $AS(\Pi) = \langle AS(\mathcal{A}), AS(s_\text{I}), AS(\delta) \rangle$
- $AS(\mathcal{A}) = \{AS(a) \mid a \in \mathcal{A})\}$ where a is an action schema
- $AS(s_\text{I}) = \{AS(f) \mid f \in s_\text{I}\}$ where f is a fact
- $AS(\delta) = \{AS(l) \mid l \in \delta\}$ where l is a literal
- $AS(f) = \langle s_p, AS(o_1), \ldots, AS(o_n) \rangle$ where $f = p(o_1, \ldots, o_n)$ is a fact with $p \in \mathcal{P}$ and $o_i \in \mathcal{O}$
- $AS(l) = AS(f)$ where $f \in F$ and $l = f$
- $AS(l) = \langle s_\neg, AS(f) \rangle$ where $f \in F$ and $l = \neg f$

- $AS(a) = \langle cost(a), AS(V_a), AS(pre_a), AS(\mathit{eff}_a)\rangle$ where $a = \langle V_a, pre_a, \mathit{eff}_a\rangle$ is an action schema
- $AS(V_a) = \{s_{a,v} \mid v \in V_a\}$ where V_a is a set of variables
- $AS(pre_a) = \{AS(l) \mid l \in pre_a\}$ where l is a literal $p(o_1, \ldots, o_n)$ or $\neg p(o_1, \ldots, o_n)$ and all o_i are either in $\mathcal{O}$ or in V_a
- $AS(\mathit{eff}_a) = \langle AS(\mathit{eff}_a^i) \mid \mathit{eff}_a^i \in \mathit{eff}_a\rangle$
- $AS(\mathit{eff}_a^i) = \langle AS(V^i), AS(cond_a^i), AS(e_a^i)\rangle$ where $\mathit{eff}_a^i = V_a^i : cond_a^i \triangleright e_a^i$
- $AS(V_a^i) = \{s_{a,v} \mid v \in V_a^i\}$ where V_a^i is a set of variables
- $AS(cond_a^i) = \{AS(l) \mid l \in cond_a^i\}$ where l is a literal $p(o_1, \ldots, o_n)$ or $\neg p(o_1, \ldots, o_n)$ where all o_i are either in $\mathcal{O}$, V_a, or V_a^i

Problem Description Graph. Alternatively to the ASG which encodes a $PDDL$ task, we use the PDG. The PDG can also be used to detect structural symmetries (Shleyfman et al., 2015). It encodes an FDR task. As FDR is a grounded formalism, we call this graph a *grounded encoding*. For the PDG we should keep in mind that we need a function, which compiles the given $PDDL$ task to an FDR task. This introduces the bias of the function and requires time and memory. Especially for some tasks of the 2018 International Planning Competition (IPC 2018) this is problematic. The IPC 2018 included some tasks which take a lot of time and memory to ground.

Definition 11.3. Problem Description Graph (Pochter, Zohar, and Rosenschein, 2011) *Let* $\Pi = \langle \mathcal{V}, A, s_\mathrm{I}, \delta\rangle$ *be an* FDR *task. The problem description graph of* Π *is a digraph* $PDG(\Pi) = \langle V, E\rangle$ *with nodes*

$$
\begin{aligned}
V = & V_I \cup V_\delta \cup V_v \cup V_d \cup V_a \cup V_e, \text{ where} \\
& V_I = \{n_I\} \\
& V_\delta = \{n_\delta\} \\
& V_v = \{n_v \mid v \in \mathcal{V}\} \\
& V_d = \{n_v^d \mid v \in \mathcal{V}, d \in dom(v)\} \\
& V_a = \{n_a \mid a \in A\} \\
& V_e = \{n_a^e \mid a \in A, e \in \mathit{eff}_a\}
\end{aligned}
$$



(a) Grounded **(b)** Lifted

Figure 11.2: Visualization of a grounded and lifted graph including node coloring for the same task from the VisitAll domain.

and edges

$$
\begin{aligned}
E = &\{\langle n_I, n_v^d\rangle \mid v \in \mathcal{V}, d = s_{\mathrm{I}}(v)\} \cup \\
&\{\langle n_\delta, n_v^d\rangle \mid v \in vars(\delta), d = \delta(v)\} \cup \\
&\{\langle n_v, n_v^d\rangle \mid v \in \mathcal{V}, d \in dom(v)\} \cup \\
&\{\langle n_a, n_v^d\rangle \mid a \in A, v \in vars(pre_a), pre_a(v) = d\} \cup \\
&\{\langle n_a, n_a^e\rangle \mid a \in A, e \in \mathit{eff}_a\} \cup \\
&\{\langle n_v^d, n_a^e\rangle \mid a \in A, e \in \mathit{eff}_a, e = cond_a^i \triangleright \mathit{eff}_a^i, v \in vars(cond_a^i), cond_a^i(v) = d\} \cup \\
&\{\langle n_a^e, n_v^d\rangle \mid a \in A, e \in \mathit{eff}_a, e = cond_a^i \triangleright v \to d\}.
\end{aligned}
$$

Every node set V_x is associated with a color. The color of a node $n \in V$ is the color of the set V_x for which holds $n \in V_x$.

Both graph encodings are lossless, i.e., given the graph and node coloring, we can reconstruct the initial $PDDL$ respectively FDR task. Figure 11.2 shows an example graph for each formalism.

11.2. Label Representations

We showed how to represent a task as a graph. For our CNN based approach, we later convert these graphs to images. For our GNN based approach, we use these graphs directly. Now we explain possible output encodings for our NN and how we act on them. Let P be the set of planners, $\mathcal{I}$ be the input to our model and X the output. Our model is the function $f : \mathcal{I} \mapsto X$. We propose five different output encodings.

1. For every planner $p \in P$ the model predicts the time p requires to solve the task Π. The model has one regression output per planner. Thus, $f : \mathcal{I} \mapsto \mathbb{R}^{|P|}$. We refer to this encoding as *time*.

2. For every planner $p \in P$ the model predicts a normalized runtime for p to solve Π. The model has one regression output per planner. Thus, $f : \mathcal{I} \mapsto [0, 1]^{|P|}$. We refer to this encoding as *normalized*.

3. For every planner $p \in P$ the model predicts a log-scaled runtime for p to solve Π. The CNN has one regression output per planner. Thus, $f : \mathcal{I} \mapsto \mathbb{R}^{|P|}$. We refer to this encoding as *logtime*.

4. For every planner $p \in P$ the model predicts whether p requires between $0 - 600s$, $601 - 1200s$, $1201 - 1800s$, or does not solve the task. The model has four sigmoidal outputs for each planner. Thus, $f : \mathcal{I} \mapsto [0, 1]^{4\times|P|}$. We refer to this encoding as *discretized*.

5. For every planner $p \in P$ the model predicts whether p solves Π. The model has a single sigmoidal output per planner. Thus, $f : \mathcal{I} \mapsto [0, 1]^{|P|}$. We refer to this encoding as *binary*.

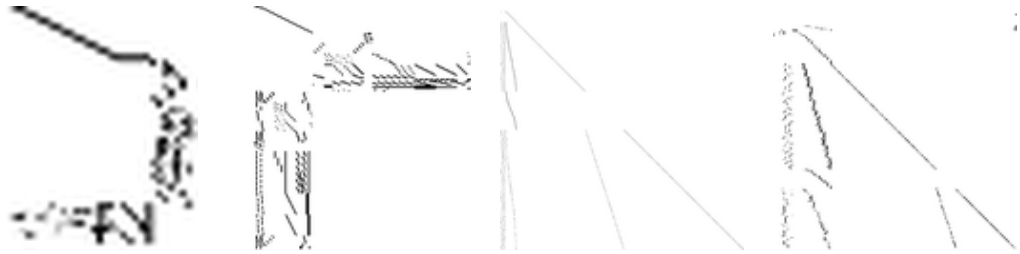
The *time* encoding is intuitive, but predicting the exact runtime could be too difficult for the model. To simplify the task, we normalized the runtimes into the range 0 to 1. We observe that the runtime distribution is heavily skewed towards short runtimes (see Figure 13.1), to counter-act this imbalance, we include the *logtime* encoding.

On a first glance, our *discretized* encoding is counter-intuitive, but there are reasons to include it. First, NN have shown that phrasing a regression problem as a classification problem can perform well. Second, the training data might be insufficient to learn the nuances need to predict the exact time, but it is sufficient to differentiate four classes. Thirdly, the training labels are noisy and discretization hides this noise. Lastly, there is no relevant difference whether a planner took 5 s or 10 s. As long as the runtime has the same magnitude, it does not matter.

Finally, the *binary* encoding seems to contain less information, but it most closely encodes the final metric. We do not want the fastest planner, but any planner which solves the task. This is a minor detail, but one planner could be slower in general, but more robust than the other planners.

We act on the predictions of our models. For the first three encodings, our online portfolio executes the planner p with minimum predicted runtime. For the *discretized* encoding, let $o_1, \ldots, o_4$ be the four outputs of a planner p. We calculate a weighted sum $\sum_{i=1,\ldots,4} i * o_i$ and execute the planner with the smallest weighted sum. For the *binary* encoding, we execute the planner p with the highest expected chance of solving the task.

Remember, we have a set $P = \{P_1, \ldots, P_k\}$ of planners and every sample $\langle \Pi, \vec{y} \rangle \in D$ has a label $\vec{y}$ where $\vec{y}_i$ describes the time P_i takes on Π. In practice, we have a time



(a) Grounded Airport **(b)** Grounded Barman **(c)** Lifted Airport **(d)** Lifted Barman

Figure 11.3: Example of the grounded and lifted images we construct for a task from the Airport and the Barman domain.

limit T. If a planner P_i did not solve Π within T, we count the task as unsolved. We convert for each output encoding the runtime $\vec{y}_i$ as training label as follows:

$$\vec{y}_i^{\text{time}} = \begin{cases} \vec{y}_i, & \text{if } \vec{y}_i \leq T, \\ 2T, & \text{otherwise} \end{cases}$$

$$\vec{y}_i^{\text{logtime}} = \begin{cases} log(t), & \text{if } \vec{y}_i \leq T, \\ log(2T), & \text{otherwise} \end{cases}$$

$$\vec{y}_i^{\text{normalized}} = \begin{cases} \frac{t}{T}, & \text{if } \vec{y}_i \leq T, \\ 2, & \text{otherwise} \end{cases}$$

$$\vec{y}_i^{\text{discretized}} = [e_1, e_2, e_3, s] \text{ with } e_i := \frac{(i-1)T}{3} < \vec{y}_i \leq \frac{iT}{3}, s := \vec{y}_i > T$$

$$\vec{y}_i^{\text{binary}} := \vec{y}_i \leq T$$

11.3. Image Based Planner Selection

Our initial online portfolio uses CNNs. Thus, we need to convert the graphs to images. Let $G = \langle V, E \rangle$ be a graph, which encodes a planning task. We extract its adjacency matrix $A \in \mathbb{B}^{|V| \times |V|}$. The nodes of a graph have no order. Nevertheless, we sort the nodes by their color. The sorting has a recognizable effect on the images and could affect portfolio quality. The adjacency matrix is a two dimensional grid like structure. We interpret it as a black & white image of size $|V| \times |V|$. Next, we *bolden* the picture, i.e., every pixel adjacent (left, right, above, below) to a black pixel is also drawn black. Now, we shrink the image. We partition it into 3×3 squares and replace every square by its average grayscale value. Then, we simply shrink the resulting image down to 128×128 pixels. Those images are the input to our CNNs. Figure 11.3 shows two examples for each graph encoding. In all images, we recognize structures. For the grounded

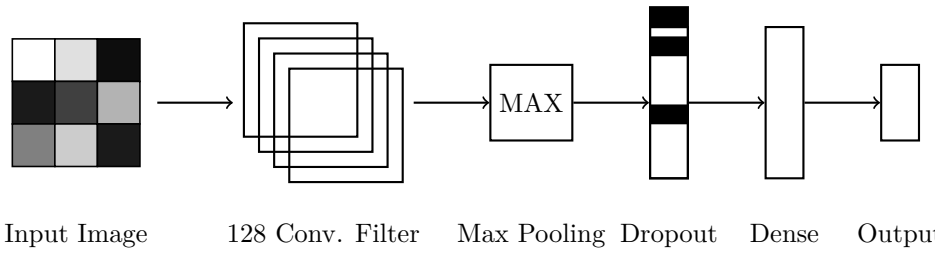


Figure 11.4: Visualization of the network architecture of our image based portfolios.

encodings, the structures are more diverse; for the lifted encodings the structures show more similarities.

Our CNNs consist of a convolution layer with 128 filters, a max pooling layer, a dropout layer, a dense layer and finally the output layer. Our CNNs are small. This makes them quick to train and evaluate and reduces the chance of overfitting. Figure 11.4 visualizes the CNN architecture. Delfi, a preliminary version of this online portfolio was submitted to the optimal track of the IPC 2018 (Katz et al., 2018)[1].

11.4. Graph Based Planner Selection

The image based approach should remove the need for handcrafting features. Although our approach relies less on handcrafted features, the conversion from graph to image is a handcrafted feature. We decided how the image loses data. In recent years, the machine learning community focused on *graph neural networks* (GNN). As the name suggests, GNNs receive graph-structured data as input. This achieved great successes in many applications. In our next step, we replace the CNN with a GNN. This allows us to skip the conversion from graph to image. Instead, we feed the graph to the NN.

In our experiments, we use the two previously defined graph encodings. For the ASG, we change the node colors in such a way that all nodes representing a number have the same color assigned. Let C be the list of all colors assignable to graph nodes. Then every node $v \in V$ has the initial feature vector $\vec{x}_v = [C_i = color(v)]_{i=1,\dots,|C|}$. We use Graph Convolution Networks (Kipf and Welling, 2017, GCN) and Gated Graph Neural Networks (Li et al., 2016, GGNN).

Definition 11.4. Graph Convolution Layer (Kipf and Welling, 2017)
Let $G = \langle V, E \rangle$ be a graph with an adjacency matrix A, a degree matrix D, and an identity matrix I. Let $\mathcal{F}$ be a set of (implicit) node features. The input to the GCN layer is a matrix $X \in \mathbb{R}^{|V| \times |\mathcal{F}|}$ which contains for every node a row with its feature values. The layer has a matrix $W \in \mathbb{R}^{|\mathcal{F}| \times |\mathcal{F}'|}$ of trainable weights and outputs a new matrix $H \in \mathbb{R}^{|V| \times |\mathcal{F}'|}$. Let $\tilde{A} = A + I$, $\tilde{D} \in \mathbb{B}^{|V| \times |V|}$ with $\tilde{D}_{i,j} = 0$ except for $\tilde{D}_{i,i} = \sum_j \tilde{A}_{j,i}$, and α be an element-wise activation function. Then a GCN layer is defined as

[1] This was before I joined the project.

$$H = \alpha(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}XW).$$

A GCN consists of one or more GCN layers placed after another. Each layer with its own trainable weights.

Definition 11.5. Gated Graph Neural Network Layer (Li et al., 2016)
Let $G = \langle V, E\rangle$ be a graph with $in(v) = \{v' \mid (v', v) \in E\}$ is the set of nodes leading to v and $out(v) = \{v' \mid (v, v') \in E\}$ is the set of nodes directly reachable from v. Let W_{in} and W_{out} be two matrices of trainable weights, which are used to combine the feature vectors of the in-coming and out-going nodes. Let $v \in V$ be a node. A GGNN layer first combines the feature vectors of the neighbors of v. Then, it forwards these values to a Gated Recurrent Unit (Cho et al., 2014, GRU). For every vertex $v \in V$, the vector $\vec{h}_v$ is initialized with its feature vector.

$$\vec{m}_v = \sum_{u \in in(v)} W_{in}\vec{h}_u + \sum_{u \in out(v)} W_{out}\vec{h}_u$$
$$\vec{h}_v = GRU(X_i, \vec{m_v})$$

A GGNN has a single GGNN layer, which is repeated a fixed number of times, i.e., it has one layer with one set of trainable weights. The output of the layer is passed as new input to the same layer, and this repeats.

Both types of GNN produce for every node $v \in V$ as output a vector $\vec{h}_v$. We do not need one output vector for each node, but a single vector $\vec{h}_g$ for the graph. Thus, we combine the individual vectors. For every node $v \in V$, we calculate an attention $a_v = \vec{w}_{gate}[\vec{x}_v\ \vec{h}_v]$, where $\vec{w}_{gate}$ is a trainable weight vector, $\vec{x}_v$ is the initial feature vector for v and $\vec{h}_v$ is the output of the GNN for v. The combined output features for the graph are $\vec{h}_g = \sum_{v \in V} a_v\vec{h}_v$. To get one output per planner p in our portfolio, we add a single dense layer with sigmoid activation at the end of our GNN: $\vec{o} = sigmoid(W_{logit}\vec{h}_g)$.

11.5. Adaptive Planner Selection

Ideally, our models pick the best planner for a task. But even good models pick wrong planners. As we select only a single planner, this can be fatal. Offline portfolios are more robust, because they always execute a set of *complementary* planners. If one planner is bad for the current task, another planner of the schedule is hopefully better. We improve our online portfolios such that they reconsider their choices and select a second planner.

The simplest idea is to switch after half the time limit passed to the planner with the second best predicted performance. Our experiments show that this is a bad idea. Instead, we train a second model to pick a complementary planner. We train a first model

as before. If the initially selected planner finds no solution within half the time limit, then the second model decides to keep the initial planner running or to switch to another planner. Our second model receives as additional input a vector $\vec{c} = [c_1 \ \dots \ c_{|P|}]$ with $c_i = 1$ if the first model chose the planner P_i and $c_i = 0$ otherwise. The final layer has a new trainable weight matrix W_{fail} to incorporate the knowledge of the previously executed planner:

$$\vec{o} = sigmoid(W_{logit}\vec{h_g} + W_{fail}\vec{c})$$

Let $D = \{\langle \Pi_1, \vec{y}_1 \rangle, \dots, \langle \Pi_k, \vec{y}_k \rangle\}$ be our original data set, $P = \{P_1, \dots, P_n\}$ be our set of planners, and T be our time limit. Then we construct a new data set D' to train the second model. For every task Π and every planner $P_i \in P$ which does not solve the task Π within half the time limit T, we construct a sample. The new sample has as features the task Π and the planner p. For all planners P_j other than P_i we increase their stored runtime $\vec{y}_j$ by $T/2$. This time passed when the second planner will start. For P_i, we do not change its runtime, because the time $T/2$ passed, but P_i was also executed during this time. Formally, we define D' as

$$D' = \{\langle \Pi, i, \vec{y}_i^{'} \rangle \mid \langle \Pi, \vec{y} \rangle \in D, P_i \in P, \vec{y}_i > T/2\}$$

with

$$\vec{y}_i^{'} = \langle \vec{y}_1 + \frac{T}{2}, \dots, \vec{y}_{i-1} + \frac{T}{2}, \vec{y}_i, \vec{y}_{i+1} + \frac{T}{2}, \dots, \vec{y}_n + \frac{T}{2} \rangle.$$

To train our second model, we convert for each sample the task Π to a graph, we construct for the planner index i the vector $\vec{c}$, and we construct the appropriate output encoding from the planner runtimes $\vec{y}'$.

12. Explainable Online Portfolios

In the previous chapter, we got rid of handcrafted features and optimized our portfolios for coverage. But we lost all explainability. *Why do we select a certain planner? Which features are important for the selection? Which properties of a task are detrimental for a given planner?* Sometimes trust is more important than coverage. If all planners in our portfolio are complete, then we can always trust them to find a solution. But if we require a solution within limited resources, we want to know if we can trust the predicted planner to find that solution within the resource limit or if it is better to use a backup planner. There exist methods to explain the shapes the filters in a CNN identify, but even if we know the shapes, which the filters detect it is unclear how to interpret them for our images. For GNNs there exist even fewer methods to explain its features.

Rudin (2019) stated, explainability is needed to build trust and we should not try to explain a black-box model. Instead, we should use an interpretable model. Thus, we return to classical, explainable machine learning techniques. We revisit the idea of handcrafted features. We restrict ourselves to simple ones. We show that simple models with simple features are competitive and additionally explainable. The explainability opens doors for future applications.

12.1. Building Trust

Simple Features & Models. We use four feature sets. Fawcett et al. (2014) crafted a huge set of features to describe planning tasks and to predict the runtime of a planner on a task. Their feature set contains simple $PDDL$ related features like the number of actions in a task, more complicated features related to the compilation from $PDDL$ to FDR, like the number of mutex groups, complicated features related to the conversion of a task to a SAT formula, and more. We call this feature set FAWCETT. The $PDDL$ related features are simple, understandable, and easy to calculate. They count things in the task, e.g., the number of actions. We call the $PDDL$ related features of Fawcett et al. (2014) FPDDL. As we deemed some $PDDL$ related features missing, we construct a feature set PDDL. This set extends FPDDL by adding ratios between some $PDDL$ features and more information about the distribution of $PDDL$ features, e.g., the minimum, mean, and maximum number of prevail conditions over all actions. For completeness, the UNION feature set contains the features of all three previous feature sets.

As simple models, we use linear regression (LR) and random forests (RF). Furthermore, we train FFNs. FFNs are not explainable, they are a middle ground between the simple ML models and the CNNs. Both, linear regression and random forests, output a single value per model. Thus, they learn internally one model per planner.

Single Decision Tree. Let Π be a task and A, B be two planners. If we want to know why our model prefers A over B for Π, we cannot answer the question. We can only say that we predict for A a better value than for B. We present a new procedure, which trains a single decision tree. The resulting tree outputs the planner to execute, and the decision trajectory informs the user why that planner is preferable over the others.

The standard decision tree training procedures expect one label per sample. We have for each sample one label *per planner* in the portfolio. Thus, we have to transform our training data. We first describe the training procedure, which corresponds to the *binary* label encoding. Let $P = \{P_1, \ldots, P_k\}$ be our set of planners, $D = \{\langle \Pi_1, \vec{y}_1 \rangle, \ldots, \langle \Pi_k, \vec{y}_k \rangle\}$ be our training data, and T be the time limit. For every sample $\langle \Pi, \vec{y} \rangle \in D$ we define $\mathcal{P}(\vec{y}) = \{P_i \mid \vec{y}_i < T\}$ the set of planners which solve the task Π within the time limit. Our new data set duplicates a sample for every planner which solves the sample $D' = \{\langle \Pi, p \rangle \mid \langle \Pi, \vec{y} \rangle \in D, p \in \mathcal{P}(\vec{y})\}$. Training on D' favors tasks which are solved by many planners, as they are more frequent in the training data. To counteract this, we weight the samples $D'' = \{\langle \Pi, p, \frac{1}{|\mathcal{P}(\vec{y})|} \rangle \mid \langle \Pi, \vec{y} \rangle \in D, p \in \mathcal{P}(\vec{y})\}$. Now, we can use D'' in combination with any standard decision tree learning algorithm.

If we want to incorporate the runtime information (like *time*, *normalized*, or *logtime*), then we add a second factor f to the sample weight. Let $\langle \Pi, \vec{y} \rangle \in D$ be a sample with $p_1, p_2 \in \mathcal{P}(\vec{y})$. Let t_1 (resp. t_2) be the time p_1 (resp. p_2) takes to solve Π and f_1 (resp. f_2) be their additional factor. If p_1 is n times faster than p_2, then f_1 is n times greater than f_2: $f_1/f_2 = t_2/t_1$. Additionally, the runtime factors f for samples associated with the same task are scaled to sum up to 1. The new data set is $D''' = \{\langle \Pi, p, \frac{1}{|\mathcal{P}(\vec{y})|} * f_{\vec{y},p} \rangle \mid \langle \Pi, \vec{y} \rangle \in D, p \in \mathcal{P}(\vec{y})\}$.

Building Trust. To build trust into a model, it helps to know which features it uses and to understand whether this makes sense. Let $\mathcal{F}$ be a set of features and m be a model. To understand whether m requires a feature $f \in \mathcal{F}$, we retrain m, but exclude f and all other features which correlate with it. The stronger the model depends on f, the more its performance drops.

A further method to understand the model and to build trust checks if the model correctly picks a planner. Let D be some test data and m be a model with a set of planners P. For every planner $p \in P$ we can calculate the fraction of tasks it solves from the test data (Cov_D). Furthermore, we can calculate the fraction of tasks it solves on those tasks for which it is chosen (Cov_C). If m picks a planner p randomly, then Cov_D and Cov_C are approximately the same. On the other hand, if the model m learned when to use p, then Cov_C is larger than Cov_D. If we show that a model correctly

identifies when a planner p performs well, then we can trust its predictions in p. On the other hand, if we see that it does not know when to use p, then we can intervene and maybe execute another planner instead.

Intuitively, tasks from the same domain are structurally similar. We would expect that a planner which works well on one task also works well on the other tasks. As a consequence, we would expect that an online portfolio which correctly learns why a planner is good for a domain predicts the same planner for all tasks within the domain. Of course, the tasks in a domain have some varying parameters and there can be a phase transition. When increasing some parameters, another planner could be preferable, but we would expect this to be rare. If we show that an online portfolio identifies domains, then this builds further trust in its learned rules. If a model does not identify domains, then this does not mean its rules are bad, but this does not increase our trust in them.

13. Experiments

In the previous chapter, we suggested many approaches to train online portfolios. Now, we train, evaluate, and compare them. In most parts, our evaluation metric is the coverage on the test tasks. A few times, we use the metric from the IPC 2018, which is similar to the coverage.

13.1. Data

Planners. The most important ingredient of a portfolio is the set the planners. If the portfolio contains only well performing planners, then the performance of the portfolio will be good. Even if it selects planners at random. At the same time, a planner with bad average performance has its place in a portfolio, if the portfolio selects it at the right moment.

We experiment with three planner collections. The *Delfi* collection $\mathcal{C}_D$ was curated for the planner Delfi (Katz et al., 2018) which participated in the IPC 2018. It includes 17 Fast Downward based planners. 16 planners are forward A^* searches (Hart, Nilsson, and Raphael, 1968). All of them prune states using strong stubborn sets (Alkhazraji et al., 2012; Wehrle and Helmert, 2014). If less than 10% of the first 1,000 states are pruned, then pruning is disabled. All searches use either DKS structural symmetry pruning (Domshlak, Katz, and Shleyfman, 2012; Shleyfman et al., 2015) or orbital space search (OSS) structural symmetry pruning (Domshlak, Katz, and Shleyfman, 2015). Those searches use one of the following eight heuristics: blind heuristic, LM-Cut (Helmert and Domshlak, 2009), iPDB (Haslum et al., 2007) with a time limit of 900s for the pattern generation, a zero-one cost partitioning pattern database (ZO-PDB) which generates the patterns using a genetic algorithm (Edelkamp, 2006), and four Merge-And-Shrink (M&S) heuristics (Dräger, Finkbeiner, and Podelski, 2006; Helmert et al., 2014). All M&S heuristics use full pruning (Sievers, 2017) and exact label reduction using Θ-combinability (Sievers, Wehrle, and Helmert, 2014). Some M&S heuristics use exact bisimulation (BS, Nissim, Hoffmann, and Helmert, 2011) for shrinking. The others use a greedy variant of bisimulation (GBS). All M&S heuristics base their merging either on DFP (Sievers, Wehrle, and Helmert, 2014), strongly connected components (Sievers, Wehrle, and Helmert, 2016, SCC), MIASM (Fan, Müller, and Holte, 2014), or score-based MIASM (Sievers, Wehrle, and Helmert, 2016). Additionally, most configurations use the h^2 mutexes to prune actions (Alcázar and Torralba,

Structural Symmetry Pruning	h^2	BS	GBS	Heuristic
OSS	x			LM-Cut
OSS	x			iPDB
OSS	x			ZO-PDB
OSS	x			Blind
OSS	x	x		M&S, MIASM, DFP
OSS	x	x		M&S, sb-MIASM
OSS		x		M&S, SCC, DFP
OSS	x		x	M&S, SCC, DFP
DKS	x			LM-Cut
DKS	x			iPDB
DKS	x			ZO-PDB
DKS	x			Blind
DKS	x	x		M&S, MIASM, DFP
DKS	x	x		M&S, sb-MIASM
DKS		x		M&S, SCC, DFP
DKS	x		x	M&S, SCC, DFP

Table 13.1: List of Fast Downward configurations in the Delfi planner collection $\mathcal{C}_D$. $\mathcal{C}_D$ contains additionally the planner $SymBA^*$.

2015). Table 13.1 lists the exact 16 configurations. The 17th planner is $SymBA^*$ (Torralba et al., 2017), a symbolic bi-directional A^* search.

Intuitively, adding new planners to the portfolio, which solve additional tasks should be beneficial. Thus, we define a second collection $\mathcal{C}_A$ which includes additionally the planners from the optimal, classical track of the IPC 2018. Those are Complementary1 (Franco et al., 2018), Complementary2 (Franco, Lelis, and Barley, 2018), DecStar (Gnad, Shleyfman, and Hoffmann, 2018), FDMS1 and FDMS2 (Sievers, 2018), Metis 2018 1 and Metis 2018 2 (Sievers and Katz, 2018), Planning-PDBs (Moraru et al., 2018), Scorpion (Seipp, 2018b), and SYMPLE 1 and SYMPLE 2 (Speck, Geißer, and Mattmüller, 2018). This is our largest planner set $\mathcal{C}_A$.

On the other hand, increasing the set of planners makes the learning objective more difficult. There are now more planners and for each one, we have to learn a performance profile. Thus, we define a last planner collection $\mathcal{C}_C$ which consists of the minimum set of planners which covers the tasks solved by all planners from $\mathcal{C}_A$. This set contains $SymBA^*$, iPDB with OSS pruning, M&S MIASM with DFP and DKS pruning, Complementary1, Complementary2, Metis 2, Planning-PDBs, Scorpion, and the symbolic-bidirectional baseline.

Planning Tasks. The quality of a machine learning model depends heavily on the quality of the training data. Thus, we train our models on a diverse collection of plan-

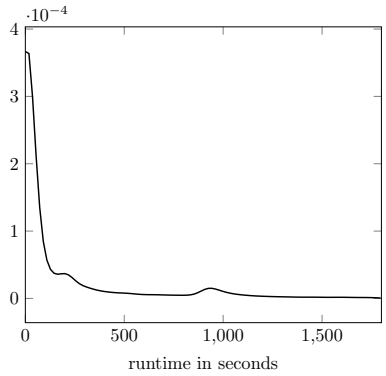


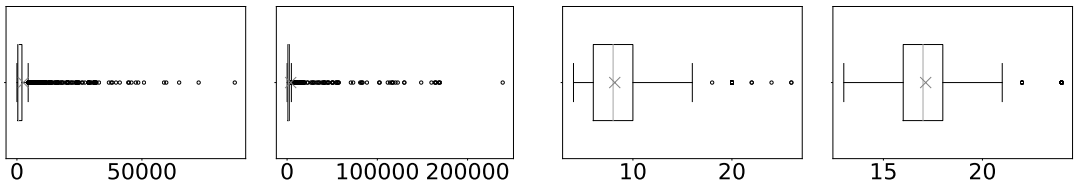
Figure 13.1: Density distribution of the runtime in seconds over all planners and those tasks which terminate within 1,800 seconds.

ning tasks. We select our training tasks from the following sources:

- all domains from the classical tracks of the International Planning Competitions (IPCs) until (including) 2018
- some domains from the learning tracks of the IPCs until (excluding) 2018
- BriefcaseWorld, Ferry, and Hanoi from the IPP benchmark collection (Köhler, 1999)
- genome edit distance (GEDP) domain (Haslum, 2011)
- domains from the conformant-to-classical planning compilation (Palacios and Geffner, 2009)
- domains from the finite-state controller synthesis compilation (Bonet, Palacios, and Geffner, 2009)

Some IPCs reused domains from previous iterations, but generated new tasks. In this case, we use only the tasks from the latest IPC. For some domains, a generator to generate new tasks exists. If necessary, we generated additional tasks for a domain. This leaves us with 2439 tasks in total.

The IPC 2018 introduced ten new, independent domains, each with 20 tasks. To simplify the grounding of two domains, the organizers included for each task within those domains a reformulation. This results in 240 tasks from 12 domains from the IPC 2018. The IPC score counts how many independent tasks a planner solved, i.e., if there are two tasks which are reformulations. If a planner solves at least one of them, then this is counted once.



(a) Grounded, size **(b)** Lifted, size **(c)** Grounded, diameter **(d)** Lifted, diameter

Figure 13.2: Distributions of graph size and graph diameter for the grounded and lifted graphs generated from our task set. The gray line in the center is the median, the cross the mean, the box spans from the 25 percentile to the 75 percentile, the whiskers extent to the 5 percentile and the 95 percentile.

Labels. To train and evaluate our portfolios, we require the runtime of each planner on our tasks. We execute each planner on a single core of an Intel Xeon Silver 4114 with 7744 MiB of memory for up to 1,800 s for each task. If the planner finds a solution, we store its runtime, otherwise we store that it failed. When we evaluate our portfolios, we do *not* run the predicted planners on the test tasks. Our data sets contain the runtime every planner required on every test task. We just look up if the planner solves the task within the time limit.

We observe that in 31% of all planner - task pairs the search ends unsuccessfully. Only in 69% of the pairs it finds a solution within the time limit of 1,800s. Figure 13.1 shows the density distribution of the runtimes for those data points where we find a solution. Like Helmert et al. (2011), we observe that most tasks are quickly solved. Our data show that a significant number of tasks is solved right after 900 seconds. Some planners perform up to 900 seconds of precomputations. Once their precomputation phase ends, their search phase starts and again they either quickly find a solution, or they find no solution.

Features. For every task we construct their PDG and ASG. From those graphs, we construct their image representations as described in Section 11.3. Furthermore, we extract the handcrafted features described in Section 12.1 from the tasks.

We observe that the generated graphs are large. For the PDG, 39% of the graphs contain more than 1,000 nodes. For the ASG even 63% of the graphs contain more than 1,000 nodes. Furthermore, the size distribution is heavily skewed. Most graphs are large, but some graphs are huge (see Figure 13.2). Compared to other graph data sets used in combination with GNNs our graphs are significantly larger (see Figure B.1). This leads to difficulties when training the GNNs.

Figure 13.3 shows for all tasks the size of their ASG relative to the size of their PDG. The tasks are sorted by the size of their PDG. For most tasks, the ASG is significantly larger than the PDG. But the larger the PDGs are, the more the ratio changes in favor

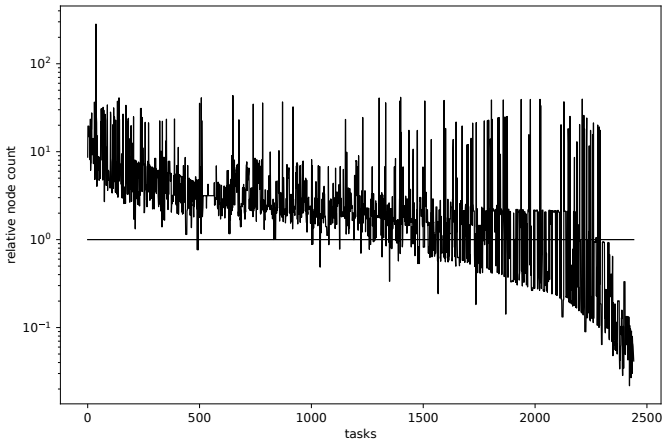


Figure 13.3: Size of the lifted graphs relative to the size of the grounded graphs over all tasks.

of the ASGs. At some point, the ASGs are smaller than their PDG counterpart. This suggests that for difficult tasks the ASG is smaller, and thus, preferable.

The diameter of a graph is the longest shortest path between any two of its nodes. In contrast to the enormous graph sizes, we observe that the *diameters* of our graphs are reasonably small (see Figure 13.2) and comparable to other graph data sets (see Figure B.2). As GNNs pass information between neighboring nodes, a short diameter ensures that the information of every node quickly spreads through the network.

Data Splits & Robustness. To evaluate our online portfolios, we require training, optionally validation, and test data. We split our data set in multiple parts. The *IPC split* resembles the setting of the IPC 2018. All tasks published before the IPC 2018 belong to the training data and all tasks from the IPC 2018 belong to the test data. We optionally split some validation data from the training data.

In machine learning, there is often the assumption that the samples in the data sets (training, validation, test) are independent and identically distributed. For planning, this does not hold. Tasks from different domains differ greatly, and even within a domain tasks are not identically distributed. Especially for the IPC 2018, the organizers curated a challenging set of domains, which differs greatly from previous domains. For example, some new domains have an especially high number of conditional effects and other domains are especially hard to ground. Nevertheless, we evaluate a *random split*, which assigns the samples randomly to the training, validation, and test sets.

The random split resembles closest the machine learning practice, but does not reflect the use case in planning. In planning, we train on tasks from some set of domains and during evaluation we expect the user to provide us a tasks from an unseen domain. Thus, we suggest a *domain-preserving* split. In the domain preserving split the data is split such that all tasks from the same (or closely related) domains are assigned to the

same new data set.

Our experiments verify that our approaches are robust, i.e., the reported values are not due to chance. Chance has two facets. First, the training of many ML techniques contains non-deterministic elements. Secondly, the performance of a model depends on the training and test data. If we use the IPC 2018 split, then we cannot change the test data. To show robustness with respect to non-determinism, we train all models 10 times and report the average performance. For the other two splits, we tackle both facets with cross-validation. For every reported performance metric, we split the data set into 10 parts using either the random or the domain-preserving split. We train and evaluate our models 10 times. Every time a different part of the data is used as test data.

All our data sets and data splits are publicly available[1].

13.2. Image Based Planner Selection

How well performs our image based portfolio? Is it competitive? To answer the first question, we take a look into the parameter choices. At the IPC 2018, we saw that the Delfi planner collection ($\mathcal{C}_D$) performs well. We test our intuition that more planners ($\mathcal{C}_A$) make the portfolio more powerful and that fewer, better planners ($\mathcal{C}_C$) simplify the learning objective. Furthermore, the *binary* encoding resembles most closely our evaluation metric. But it abstracts all time information. Providing time information for training could improve the portfolio. Thus, we also evaluate the *discretized*, *normalized*, and *time* output encoding. We do not have independent and identically distributed data. Thus, *how should we separate the data?* Should we randomly split validation data off training data or domain-preserving split better, because it resembles more closely the final evaluation (the test set contains only tasks from unseen domains)? And finally, *should we use the lifted or grounded graphs?*

To answer these questions, we train each model using an NVIDIA Tesla K80 GPU for 250 epochs using the mean squared error for the *time* and *normalized* outputs and the binary cross entropy for the *discretized* and *binary* outputs. During training, we measure the training progress using validation data. We store the state of the model with the minimum loss on the validation data. As training is a matter of minutes and never takes longer than an hour, we train and evaluate all combinations of the enumerated parameters.

We evaluate all planners in the setting of the IPC 2018. That means we train all models on the task known prior to the IPC 2018 and evaluate the portfolios and baselines on all 240 tasks from the IPC 2018. We report the fraction of the maximum IPC score.

Before we start training the final models, we perform for every configuration automatic hyperparameter optimization (HPO). We adapt the approach of Diaz et al. (2017).

[1] `https://github.com/IBM/IPC-graph-data`
`https://github.com/IBM/IPC-image-data`
`https://doi.org/10.5281/zenodo.5749959`

		domain-preserving split				random split			
		validation		no validation		validation		no validation	
		mean	std	mean	std	mean	std	mean	std
time	$\mathcal{C}_D$	50.0	4.4	57.3	1.6	57.5	1.5	57.5	0.0
	$\mathcal{C}_A$	48.7	4.4	49.9	2.7	50.8	3.4	48.8	0.9
	$\mathcal{C}_C$	52.6	3.9	50.5	2.2	50.7	3.9	50.3	2.3
normalized	$\mathcal{C}_D$	50.9	4.4	53.8	2.0	55.4	3.1	54.9	3.1
	$\mathcal{C}_A$	51.8	3.7	50.5	2.6	48.8	1.2	49.3	1.8
	$\mathcal{C}_C$	49.5	5.6	50.2	2.1	50.0	1.3	50.3	1.8
discrete	$\mathcal{C}_D$	49.5	4.0	53.7	5.9	53.9	3.3	54.1	3.0
	$\mathcal{C}_A$	55.4	3.4	52.7	2.2	53.9	3.8	53.7	5.1
	$\mathcal{C}_C$	50.5	1.6	51.6	3.1	58.3	5.2	53.3	1.4
binary	$\mathcal{C}_D$	49.6	4.0	50.2	1.4	52.0	3.3	50.3	1.1
	$\mathcal{C}_A$	50.4	4.7	48.9	1.8	49.9	2.2	49.6	1.5
	$\mathcal{C}_C$	53.4	3.0	49.2	2.2	52.3	2.7	51.7	3.6

Table 13.2: IPC 2018 score for the lifted image based online portfolios trained on the pre-IPC 2018 tasks.

For example, the dropout rate and the size of the convolution filter are hyperparameter. The automatic optimization selects values for the hyperparameters and then performs 5-fold cross validation using only the training data to evaluate the chosen parameters. At the end of the process, we have for every configuration a set of good hyperparameters. For reproducibility, we report the hyperparameters in the appendix (see Table B.1).

For every configuration c with data split s, we train 10 models using 10-fold cross validation, i.e., the training data is split into 10 parts. In every iteration a different part is used as validation data. The other parts are used as training data. The disadvantage with the cross validation is that one part of the data is not used for training. Thus, we train additionally 10 models per configuration using the full training data for updating the model and using the full training data to select the best model after training. This means the impact of the data split on these models is only due to the hyperparameter optimization.

With the data on all these configurations, we observe that the lifted graphs are always preferable over the grounded graphs. We confirm this also in later experiments (see Table 13.6). Thus, we restrict ourselves now to the lifted configurations. The results are qualitatively the same for the grounded configurations.

Table 13.2 shows the average IPC score for all 48 lifted configurations. Those are too many configurations to compare them individually. Thus, we summarize the results. For every parameter (e.g., the planner collections), we pair-wise compare its options. Let c_1, c_2 be two configurations, which differ only in one parameter. If c_1 performs on

	T	*N*	*D*	*B*
Time	-	**7**	5	**7**
Normalized	4	-	4	**7**
Discrete	**7**	**8**	-	**10**
Binary	5	5	2	-

(a) Label encodings

	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$
$\mathcal{C}_D$	-	**12**	**10**
$\mathcal{C}_A$	3	-	6
$\mathcal{C}_C$	6	**10**	-

(b) Planner collections

	Domain-preserving Split		Random split	
	Validation	No Validation	Validation	No Validation
Dom-pres. Split & Val.	-	5	5	5
Dom-pres. Split & No Val.	**7**	-	2	3
Random Split & Val.	**7**	**10**	-	**8**
Random Split & No Val.	**7**	**9**	3	-

(c) Data splits

Table 13.3: For the lifted image based online portfolio the pair-wise comparison between the options of its training parameters (label encodings, planner collections, and data splits) using the data from Table 13.2. For each option pair it counts how often one option (row) obtains a higher IPC 2018 score than the other option (column).

average better than c_2, then we count this as win for the option by c_1 over the option used by c_2.

Table 13.3a shows the results of the pair-wise comparison for the output encodings. We see immediately that the *discretized* encoding compares favorable against all other encodings. The combination of classification network and encoding time information is advantageous. Unexpectedly, the *binary* encoding, which is closest to our metric, performs the worst on average. In two out of three comparisons, it is close to a tie.

Moving to the planner collections (see Table 13.3b), we see that the Delfi planner collection performs best. Although more planners could improve the portfolio because there are now more choices, it seems that the additional planners hinder the training. This is astonishing as the larger planner collection $\mathcal{C}_A$ has a higher average quality. Picking random planners from $\mathcal{C}_A$ leads to better results than picking random planners from $\mathcal{C}_D$ (see Table 13.4). The collection $\mathcal{C}_C$ contains only those planners required to cover all tasks and picking random planners from this collection is significantly better than picking random planners from $\mathcal{C}_D$. Nevertheless, $\mathcal{C}_C$ wins only against $\mathcal{C}_A$, but not against $\mathcal{C}_D$. We learn that the initial $\mathcal{C}_D$ collection was a lucky choice and that there is no simple rule for selecting the right planners for our online portfolio. It is future work to understand precisely the conditions, which make up a good planner selection.

Let us answer the last question in this series: How should we treat the training data? Although in planning the samples are not independent and identically distributed, Table 13.3c shows that it works best to randomly split the training data for HPO and for

Rnd. $\mathcal{C}_D$		Rnd. $\mathcal{C}_A$		Rnd. $\mathcal{C}_C$		Oracle			Best		
Mean	Std	Mean	Std	Mean	Std	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	Sym	C2	Delfi1
42.8	8.3	45.0	8.8	50.3	9.8	67.9	72.1	70.8	57.1	58.3	60.0

Table 13.4: IPC 2018 score when randomly picking planners from the planner collections (Rnd. $\mathcal{C}_D$, Rnd. $\mathcal{C}_A$, Rnd. $\mathcal{C}_C$), when picking the best planner for each task from the planner collections (Oracle), and for state-of-the-art planners (Best).

the final training. Our test data from the IPC 2018 consist of new domains, which differ greatly from the previously known domains. An explanation could be that the random split allows us to train on as many domains as possible. Thus, the model generalizes better. The domain-preserving split reduces the number of domains in the training set, but has a more realistic validation loss. Remember, after training we pick the state of the model with the smallest validation loss. The advantage of the better generalization outweighs the advantage of a better validation loss. This explanation also fits the observation that after optimizing the hyperparameters for the domain-preserving split, it is better not to use validation data.

Finally, we compare our image based portfolios against some baselines (see column 1-3, Table 13.2). For every planner collection, we calculate the IPC score when randomly picking planners from the collection. We repeat the calculations 1,000 times and average over the computed scores. If the models learned to select the right planners or at least avoid bad planners, then they pass this baseline. For the Delfi and the extended collection we see that all configurations exceed their baseline. The minimum planner collection has a significantly higher random baseline, which makes it harder to pass. Most configurations pass, but some do not. As summary, most models learned useful knowledge.

If we compare our portfolios against their theoretical maximum score (see column 4, Table 13.4) we observe that there is still room for 10-20% of improvement. We reduce this gap in the next section.

Using hindsight knowledge, the best planner for the IPC 2018 domains in the Delfi collection would be $SymBA^*$ and the best planner in the other two collections is Complementary2. We see that our models rarely exceed those planners. Thus, we see again there is room for improvement. For fairness, the IPC 2018 domains differ greatly from the previously published domains, thus, the training data would not have suggested executing only those planners.

Our last comparison is Delfi1. Delfi1 is a preliminary version of this approach and was submitted to the IPC 2018 and won the optimal track. It uses the lifted graphs, *binary* output encoding, a single domain-preserving split for the hyperparameter optimization and does not split the final training data. Although Delfi1 used suboptimal parameters, it dominates all shown configurations and all other planners from the IPC

2018. At the end, we conclude that the non-determinism during training was lucky for Delfi1.

13.3. Graph Based Planner Selection

The image based portfolios performed better than the random baselines, but worse than some other planners and especially worse than Delfi1. This motivated us to improve our approach. Furthermore the conversion from graph to image is ad-hoc, lossy, and in general just not satisfying, thus, we proposed the graph based online portfolio. Additionally, we proposed an adaptive mechanism, which predicts a second planner if the first one does not terminate within half the time limit. First, we show that both modifications perform consistently better than their predecessors. Then, we show that we finally outperform Delfi1.

We use a parameter setting, which keeps us comparable to Delfi1. We train models using the lifted and the grounded graphs. We use only the Delfi planner collection. Although it did not perform best, we use only the *binary* output encoding, which was used by Delfi1. Our GCNs have 4 layers and our GGNNs perform 4 repetitions. All GNNs use 100 node features in the hidden layers. We do not perform hyperparameter optimization, instead we execute an early exploratory experiment where we also tested models with 2 and 6 layers, as well as, 150 and 200 node features in the hidden layers. We train all models with binary cross entropy loss and Adam optimizer (Kingma and Ba, 2015) for up to 300 epochs. We use early stopping. If the validation loss does not improve for 25 epochs, then training stops prematurely. Once training stopped, we return the state of the model with the best validation loss. Training is executed on 8 CPU cores and a GPU. We use 10 GB of memory. In general, one would train with a fixed batch size, but some of our graphs are exceptionally huge. Thus, we dynamically adapt the batch size. In every iteration we add graphs to the batch until the memory is full. The training terminates within 10 minutes.

We limit our training and test data to those tasks which are solved by at least one planner of the Delfi collection $\mathcal{C}_D$ within the time limit. There is no reason to train on samples which have the same signal for all planners and there is no reason to test on a sample which we cannot solve. 145 samples remain from the IPC 2018. We use these as test data. All reported coverage numbers are fractions of these 145 tasks.

To show that we consistently outperform the image based portfolios, we use the random and the domain-preserving splits to partition the pre-IPC 2018 data in ten parts. For each part, we train models using that part as validation data and the other as training data. Table 13.5 reports for the lifted graphs the average fraction of test tasks our methods solve. As a comparison, we also list how many tasks our image based portfolio with the same parameters solve. For both data splits, the GCNs performs a lot better than the GGNNs. But more importantly, feeding the graph directly to a GCN solves even more tasks than feeding the image to a CNN. If we additionally allow reconsidering the

	Domain-preserv.		Random	
	Mean	Std	Mean	Std
Image based, CNN	82.1%	6.6%	86.1%	5.5%
Graph based, GCN	**85.6%**	**5.5%**	**87.2%**	**3.5%**
Graph based, GGNN	76.6%	5.8%	74.4%	2.7%
Adaptive, GCN	**91.1%**	**3.8%**	**92.1%**	**3.2%**
Adaptive, GGNN	83.0%	5.8%	86.6%	2.0%

Table 13.5: Average coverage (in %) over 10 models on the solvable IPC 2018 task for the lifted online portfolios trained using a domain-preserving or random data split.

planner choice after half of the time limit, we see that this consistently improves the performance even further by 5-12%. The adaptive approach on a GCN with the lifted graphs is only 10% worse than the oracle. As a final note, we see again that the random split performs better than the domain-preserving split.

Knowing that the graph and the adaptive approaches work, we verify that they are the new state of the art. We train our methods again, using the exact training/validation split used by Delfi. Table 13.6 reports the performance of our approaches against baselines and other state of the art planners.

All shown planners are significantly better than randomly picking a planner from $\mathcal{C}_D$ and they are better than picking the single best planner from the training data.

The next three planners are top performers from the IPC. Some of them are better than some of our graph based online portfolios, but all of them are worse than the portfolio which uses GCN on the lifted graphs. The next planner is an online portfolio described by Fawcett et al. (2014) which we trained. This portfolio performs surprisingly well, but again not as the lifted GCN portfolio.

The next two planners are the Delfi portfolios from the IPC 2018. Remember, when retraining the lifted Delfi portfolio, we did not reach similar performance values. All graph portfolios using the grounded graphs are superior to the grounded Delfi. Our lifted GCN portfolio is superior to the lifted Delfi.

The simplest approach to execute two planners with our online portfolios is to pick the two planners with the best predicted performance and execute both of them for half the time limit. The next four rows show that this simple idea does not work. The planner with the second best predicted performance is often similar to the planner with the best predicted performance. This approach executes two similar planners each for half the time limit.

The final four rows of the table show the effect of the adaptive approach. We see that reconsidering significantly improves the performance in all settings. Three out of our four adaptive portfolios are superior to *all* other planners. Only the grounded GGNN, which we expected to perform worst, is inferior to our lifted, non-adaptive

Method	Solved	Eval. Time (in s)
Random planner	60.6%	0
Single planner for all tasks	64.8%	0
Complementary2	84.8%	0
Planning-PDBs	82.0%	0
Symbolic-bidirectional	80.0%	0
Enhanced features + random forest	82.1%	0.5
Delfi (CNN), grounded	73.1%	11.0
Delfi (CNN), lifted	86.9%	3.2
GCN, grounded	80.7%	23.2
GCN, lifted	**87.6%**	9.4
GGNN, grounded	77.9%	14.5
GGNN, lifted	81.4%	11.4
Top-2, GCN, grounded	62.1%	23.2
Top-2, GCN, lifted	65.5%	9.4
Top-2, GGNN, grounded	82.8%	14.5
Top-2, GGNN, lifted	76.6%	11.4
Adaptive, GCN, grounded	88.3%	-
Adaptive, GCN, lifted	89.7%	-
Adaptive, GGNN, grounded	84.8%	-
Adaptive, GGNN, lifted	89.0%	-

Table 13.6: Coverage (in %) on the solvable IPC 2018 tasks for our online portfolios and comparison planners.

GCN portfolio and the lifted Delfi.

As a final note, the table also reports the evaluation time for our models. For the adaptive models, we expect the evaluation time to be twice the non-adaptive evaluation time. For all portfolios, their overhead is negligible compared to the time limit of 1,800 seconds.

13.4. Explainable Planner Selection

We showed that we exceed the state of the art. Now we return to explainable machine learning models. Are they competitive? Which knowledge and trust can we extract?

As input we use the four described data sets FPDDL, FAWCETT, PDDL, and UNION. We compare for the FAWCETT and PDDL data set, as well as the image from our initial portfolio, the resource consumption to construct the model input (see Table 13.7).

	FAWCETT			*PDDL*			Images		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
Time	0.1	0.2	10.8	0.2	0.3	11.0	0.4	0.8	50.2
Memory	16	17	200	24	25	138	26	69	3023

Table 13.7: Minimum, mean, maximum time (in seconds) and memory usage (in MB) to extract the features from the IPC 2018 tasks.

We average the values over the IPC 2018 tasks. The handcrafted features require fewer resources, but unless we use a resource constrained device, the difference is negligible.

As output encoding, we use again *time*, *binary*, and now also *logtime*. We train linear regression models without L1 regularization (L1 weight of zero), and with L1 weights of 0.1, 1.0, 2.0, and 5.0. Our random forests consist of 50 decision trees. We observed a diminishing effect of adding more trees. Our FFNs have 3 or 5 hidden layers. They use a ReLU activation function and the mean squared error loss for the *time* and the *logtime* labels. For the *binary* labels, they use a sigmoid activation function and the cross entropy loss. All FFNs are trained with Adam.

We first show that those simple ML techniques learn good models. As training these models is a question of seconds to minutes, we train all combinations of models, feature sets, and output encodings, leading to 60 configurations. We take the whole data set, including IPC 2018, and split it domain preservingly to generate 10 different test sets. We train and evaluate one model per test set.

Table 13.8 shows the average coverage for each configuration. The standard deviation is large, because the tasks in the test sets are not identically distributed. For this data set, the random baseline solves 67.2% of tasks. All configurations exceed this baseline, i.e., all portfolios learn something useful. Selecting in every of the 10 folds the single best planner on the training data leads to a coverage of 73.5%. Almost all portfolios exceed this baseline. Many planners solve more than 80% of the tasks, some up to 88%. Thus, simple ML techniques learn good portfolios.

Additionally, we proposed a single decision tree, which predicts directly the planner to choose. We train it on the same 10 folds as described above with varying tree depth. Figure 13.4 shows the training and test performance for increasing maximum tree depth. Increasing tree depth increases the number of decision nodes. A tree of depth i perfectly differentiates up to 2^i samples. Thus, we quickly overfit. The coverage for the *binary* encoding alternates around 80%, the coverage for the time based labels is a bit lower. Again, the portfolio obtains a good performance and exceeds both baselines. As the test performance for time label does not improve much with more than 2 layers, we visualize the associated decision tree (see Figure 13.5). The tree is easy to interpret. $SymBA^*$ is preferable to the other two planners whenever the number of atoms and objects is small. In personal communications, the authors of $SymBA^*$ confirmed that this matches their experience.

		Linear Regression										MLP				Rnd. Forest	
		0.0		0.1		1.0		2.0		5.0		3		5		50	
FAWCETT	bin	78.6	(8.3)	77.2	(10.5)	82.1	(8.7)	82.4	(9.4)	80.9	(9.4)	**87.1**	**(6.1)**	78.2	(15.3)	**84.8**	**(7.5)**
	logt	79.3	(9.2)	79.0	(10.0)	81.5	(7.7)	81.7	(6.5)	83.6	(5.2)	82.2	(8.4)	82.2	(8.4)	84.1	(7.1)
	time	78.6	(8.2)	81.8	(7.1)	80.5	(7.5)	80.4	(7.2)	80.3	(7.9)	82.2	(7.6)	**85.3**	**(6.7)**	81.8	(15.7)
FPDDL	bin	**87.7**	**(7.2)**	74.3	(15.1)	72.7	(15.7)	74.3	(16.6)	71.4	(15.4)	81.0	(8.0)	81.5	(7.3)	77.5	(16.0)
	logt	82.5	(11.8)	84.0	(6.8)	78.5	(8.3)	77.7	(9.0)	80.3	(8.4)	78.2	(6.2)	79.7	(7.6)	82.0	(6.1)
	time	86.5	(7.8)	**86.5**	**(7.8)**	**86.5**	**(7.9)**	**86.6**	**(7.8)**	**86.6**	**(7.8)**	80.2	(6.6)	81.9	(6.0)	78.8	(15.5)
PDDL	bin	81.4	(9.3)	75.7	(12.1)	72.6	(16.3)	74.1	(16.6)	71.4	(15.4)	78.1	(9.9)	79.8	(6.8)	80.2	(13.3)
	logt	82.1	(8.4)	79.7	(11.2)	80.4	(9.1)	79.8	(8.7)	77.8	(13.0)	79.5	(8.2)	78.0	(7.5)	82.8	(7.0)
	time	81.6	(8.9)	82.0	(9.1)	81.2	(10.1)	79.0	(10.9)	78.7	(11.6)	77.8	(11.0)	78.4	(10.0)	79.7	(16.7)
UNION	bin	74.8	(9.7)	81.0	(8.3)	79.4	(11.1)	82.4	(9.3)	80.9	(9.4)	84.7	(7.7)	78.3	(13.6)	82.1	(8.5)
	logt	75.6	(10.2)	80.0	(9.2)	80.7	(7.9)	81.8	(6.7)	83.4	(5.8)	82.2	(8.4)	82.2	(8.4)	84.7	(7.6)
	time	74.8	(8.8)	77.3	(11.9)	75.7	(11.0)	76.1	(11.5)	77.1	(10.3)	84.3	(6.8)	83.6	(7.7)	84.0	(13.9)
	avg	80.3		79.9		79.3		79.7		79.4		81.5		80.8		81.9	

Table 13.8: Mean coverage and in brackets standard deviation (in %) over ten domain-preserving test folds for linear regression models with different L1 regularization weights, MLPs with 3 and 5 layers, and a random forest with 50 trees trained on the features of (FAWCETT Fawcett et al., 2014) (FAWCETT), the PDDL features of (PDDL Fawcett et al., 2014), the extended set of PDDL features (PDDL), and the union of all features (UNION) using the binary (bin), logtime (logt), and time label encoding. The best setting in each column is highlighted.

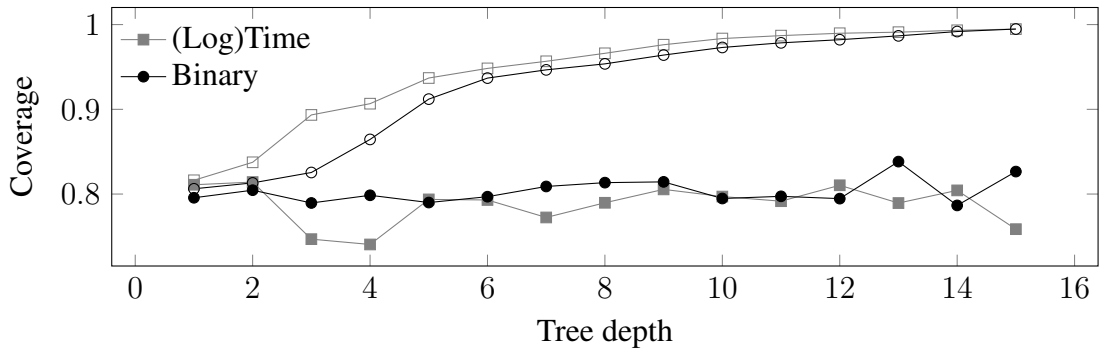


Figure 13.4: Mean coverage over 10 domain-preserving folds on training (hollow) and test (filled) data using single decision trees trained on binary (circle) and time/logtime labels (square) for increasing tree depth on the $PDDL$ features.

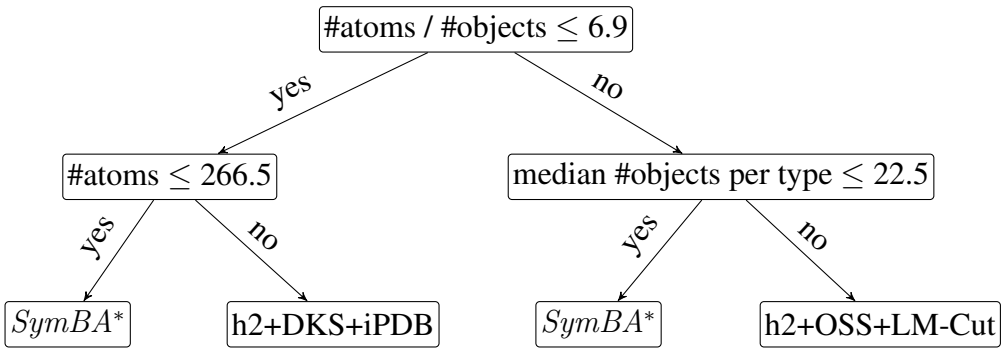


Figure 13.5: Example decision tree of depth two using time labels from one cross-validation fold.

Method	Solved	Eval. Time (in s)
Random planner	60.6%	0
Single planner for all tasks	64.8%	0
Delfi (CNN), lifted	86.9%	3.2
GCN, lifted	87.6%	9.4
Adaptive, GCN, lifted	89.7%	-
Linear Regression	86.2%	3.8
Decision Tree	82.7%	4.6
Random Forest	76.8%	4.1
MLP	70.8%	3.9

Table 13.9: Coverage (in %) on the solvable IPC 2018 tasks for some online portfolios and our explainable machine learning models.

From the previous results, we pick the best configurations for linear regression, random forest, FFN, and the single decision tree and train a new model using the IPC setting. The results are shown in Table 13.9. None of the explainable ML portfolios is superior to any of our image or graph based portfolios. The linear regression configuration which performed excellent on the previous data split performs again excellent in the IPC setting and solves almost as many tasks as Delfi! We conclude that explainable models can be competitive.

Our further experiments inspect a single model and improve the trust. We pick the best trained linear regression model for the domain-preserving split (FPDDL feature set, *binary* output, without L1 regularization). First, we execute the procedure to identify the features on which it relies. We say two features are correlating if their absolute Pearson correlation is greater than 0.95. This results in 47 feature groups. The Pearson correlation captures only linear dependencies, but this is acceptable, because linear regression exploits only linear correlation. Table 13.10 shows the coverage degradation

Feature	Degradation		Feature	Degradation	
req. neg. pre.	4.4	(10.0)	# goals	1.2	(7.6)
max params per pred.	2.7	(7.0)	has types	1.0	(7.9)
mean neg. per eff	2.6	(10.6)	min pred. per pre.	0.9	(8.2)
mean pred. per eff	2.4	(10.2)	# predicate symbols	0.9	(7.2)
req. cond. eff.	2.1	(9.1)	req. ADL	0.8	(6.9)
req. equality	1.8	(8.9)	max neg. per eff.	0.8	(6.1)
max pred. per eff.	1.8	(8.3)	min neg. per eff.	0.7	(8.0)
# types	1.6	(9.9)	# actions	0.7	(7.4)
min pred. per eff	1.6	(7.7)	# initial conditions	0.6	(7.0)
frac. actions w. neg. eff.	1.5	(9.8)	max pred. per pre.	0.5	(8.6)
req. STRIPS	1.5	(7.7)	mean pred. per pre.	0.4	(10.3)
req. typing	1.4	(8.1)	req action costs	0.2	(6.8)
mean params per pred.	1.4	(8.0)	# initial functions	0.1	(6.9)

Table 13.10: Mean coverage degradation and (in brackets) standard deviation (in %), over ten domain-preserving test folds, when ignoring a single group of correlated features of the FPDDL feature set for training a linear regression model without L1 regularization on the binary labels. Groups without performance degradation are omitted.

when excluding feature groups. We learn that the most important feature is whether a task uses negative preconditions. Without this feature the coverage drops by 4.4%. Intuitively, some planner configurations in our portfolio do not work too well with negative preconditions. Another useful insight is that 21 out of 47 feature groups have no impact on the performance and can be excluded. Excluding features speeds up the evaluation.

Secondly, we investigate whether a model selects a planner correctly. Table 13.11 shows the planners selected by our LR model, as well for every planner its chance of solving a task on the whole test set and its chance of solving the task when selected. The LR model heavily relies on $SymBA^*$ (43% of all tasks). We see that whenever it chooses $SymBA^*$, we should trust this choice. For most planners, it learned correctly when to choose them. Thus, we can trust the model in general. Only when it chooses the iPDB search with DKS pruning, we should ask for a second opinion. This could be another model or just a good baseline planner.

Thirdly, if a model learned to identify a domain, this builds additional trust. Figure 13.6 shows for our portfolios from Table 13.9 for each IPC 2018 task the selected planner. The tasks are grouped by domains and within a domain naturally sorted. The last row "Opt" is an oracle, which selects the best planner for each domain. The oracle solves all except for one task. This confirms our intuition that the right planner solves all tasks within a domain. Interestingly, those are almost exclusively the same planners

Usage	Cov_D	Cov_C	Planner
43.7	80.1	94.4	$SymBA^*$
12.3	82.4	89.9	h2 + OSS + LM-Cut
9.7	78.7	54.5	h2 + DKS + iPDB
9.4	78.8	88.5	h2 + OSS + iPDB
8.1	82.7	78.1	h2 + DKS + LM-Cut
5.4	67.9	74.8	DKS + M&S-MIASM-DFP
3.3	74.8	97.5	h2 + DKS + M&S-BS-sbMIASM
2.8	65.9	86.6	h2 + OSS + M&S-SCC-DFP
2.1	75.8	100.0	h2 + DKS + M&S-BS-SCC-DFP
1.0	67.7	84.0	OSS + M&S-MIASM-DFP
0.8	72.2	75.0	h2 + OSS + M&S-BS-sbMIASM
0.7	68.4	6.2	h2 + DKS + ZO-PDB
0.4	67.6	60.0	h2 + DKS + M&S-SCC-DFP
0.2	68.6	100.0	h2 + OSS + ZO-PDB
0.1	62.3	100.0	h2 + DKS + Blind
0.0	62.5	–	h2 + OSS + Blind
0.0	75.2	–	h2 + OSS + M&S-BS-SCC-DFP

Table 13.11: Planners selected by the linear regression model without L1 regularization trained on the FPDDL features and optimizing the *binary* labels. The columns show how often each planner is chosen (in %), the coverage (in %) for the planner on the whole data set (Cov_D), and the coverage (in %) on tasks for which the model chooses the planner (Cov_C). Figure 13.6 uses the assignment between planners and colors from this table.

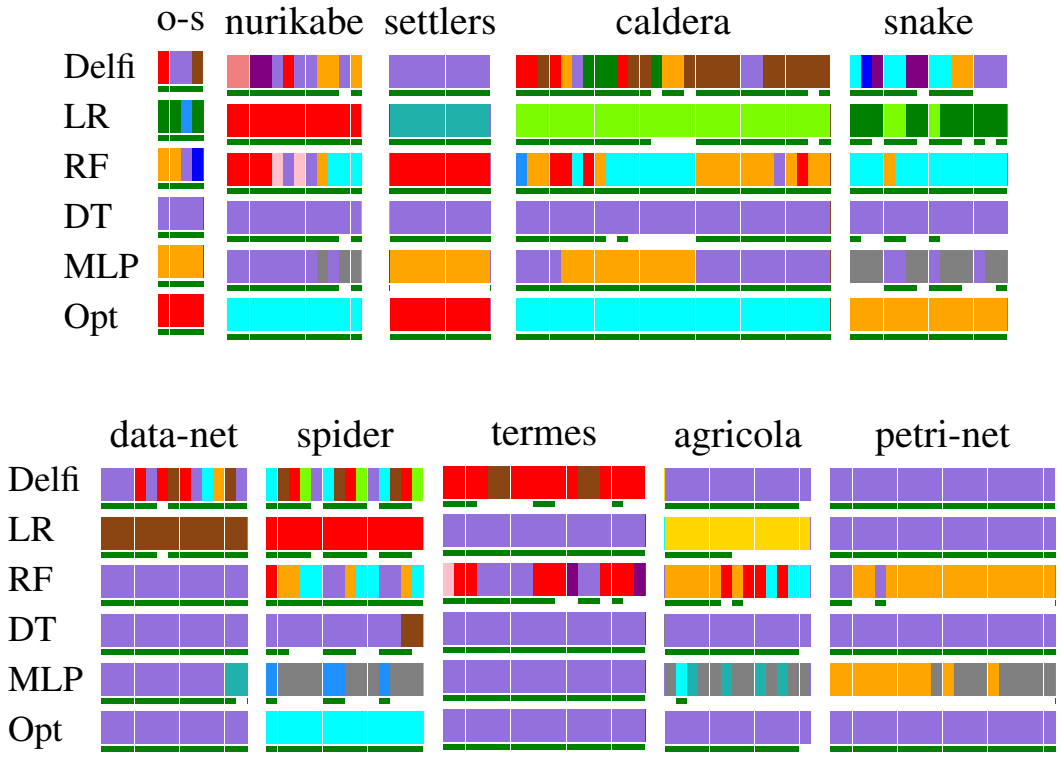


Figure 13.6: Each row indicates for portfolios from Table 13.9 which planners they select on the solvable IPC 2018 tasks. The colors correspond to the colors from Table 13.11. The tasks are grouped by domain and within a domain naturally sorted. The green bar below a task indicates that the selected planner solves the task. Opt is an oracle which picks the single best planner per domain. We abbreviate organic-synthesis with "o-s".

that our single decision tree learned to pick (see Figure 13.5)

Delfi is the best planner which we visualize. We see in its planner choices that it often uses many planners within a domain. Whatever it learned, it did not learn features which discriminate domains. In contrast, our linear regression model selects for almost all domains exactly one planner and that planner indeed solves almost all tasks. Only in two domains it switches between two planners. The fact that it learned features which also separate domains builds trust that those features are meaningful for planner selection. The decision tree is another interesting example. Like the linear regression model, it selects almost always only a single planner, but it selects across all domains the same planner, $SymBA^*$. It could have learned a good set of rules, which just evaluates on this test data to always the same planner. For the test data, this is not a bad choice. But, it does not build additional trust.

14. Summary and Future Work

We presented three new deep learning approaches for training online portfolios. We list and evaluate fundamental parameter choices. The most important question is *how to feed a planning task to a neural network such that it can by itself extract the important features.* We presented a lifted and a grounded graph encoding and observe in all experiments that the lifted encoding is superior. Additionally, the lifted encoding is cheaper to construct. Our first deep learning online portfolio method uses convolutional neural networks (CNNs). Thus, we additionally propose a lossy translation from graph to image. We show that our CNN portfolios learn relevant knowledge, but they are not yet competitive against the top planners from the IPC 2018. Our second online portfolio method uses graph convolution networks (GCNs) or gated graph neural networks (GGNNs) and directly receive the graph encodings as input. Thus, we truly require no handcrafted features. We show that the GCN are always superior to the GGNN and that the lifted GCN portfolio outperforms the state of the art planners on the IPC 2018 tasks. Our third deep learning online portfolio method allows the portfolios to reconsider the chosen planner. The reconsideration strictly improves the portfolios. Now we significantly outperform the state of the art.

In the last part, we showed that we can train machine learning models, which are simpler than CNNs and GNNs as online portfolios. Those online portfolios are competitive with the state of the art planners. They are not competitive with our GNN portfolios. Additionally, we showed how we can understand these online portfolios and how we can build trust in them.

There are two main avenues for future work, improving the online portfolios and exploiting the explainability. The coverage of our online portfolios improves significantly if we allow them to reconsider the chosen planner *once*, after a *fixed point in time*. Relaxing both constraints could lead to further improvements. In the general case, the online portfolio predicts an arbitrary sequence of planner, time limit pairs. This does not work with our current setup. We would need to train one model per point in time at which we want to select a new planner. A solution is to train a single model, which receives as input the task to solve, for each planner the maximum time they were already executed, the currently executed planner, the time for which the current planner is executed, and the remaining time. The model produces two outputs, one for the next planner to choose and one for the time limit for the new planner. Our training data construction would not work anymore either. We cannot construct training data for all passed amounts of time and all combinations of planners executed. A solution is re-

inforcement learning. If the model chooses a planner which solves the training task, the reward is positive. If the model predicts an insufficient time limit, the reward is reduced. If it chooses a planner which does not solve the task, the reward is negative. If the remaining time suffices for another planner to solve the task, the absolute value of the negative reward could be reduced.

In our current explainability setup, we only looked at complete portfolios consisting of arbitrary planners. One could also change the scope to get knowledge of a specific planner. For example, we could just look at the internal model relevant for a single planner, or we could construct a new portfolio where all planners are identical except for one parameter, e.g., A^* search with M&S but only the merging strategy is adapted. The latter allows us to understand when a merging strategy is especially good or bad. If this does not match our intuition, we can inspect the strategy more closely. Either we learn something new, or we improve the code to fix a bottleneck.

Part III.

Learning State Space Topologies

For this part, I collaborated with the following people (listed in alphabetical order):
Liat Cohen, Thomas Keller, Jendrik Seipp

15. State Space Topology

From understanding an algorithm comes the knowledge to improve it. A lot of research tries to understand the properties of A^* (Martelli, 1977; Pearl, 1984; Dechter and Pearl, 1985; Korf, Reid, and Edelkamp, 2001; Helmert and Röger, 2008; Holte, 2010). In recent times, understanding the behavior of GBFS made huge progress (Wilt and Ruml, 2014, 2015, 2016; Heusner, Keller, and Helmert, 2017, 2018). Two important concepts for understanding the behavior of GBFS are the *state space* and the *state space topology*.

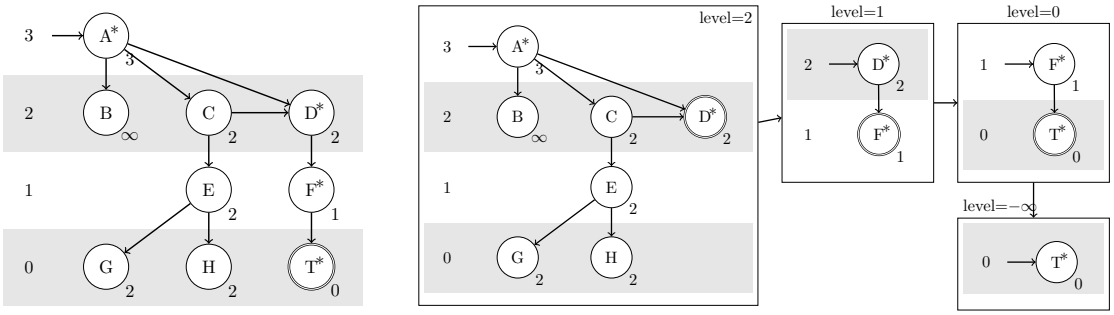
Let Π be a planning task and h be a heuristic. The state space $\mathcal{X}(\Pi)$ of the task Π is a graph structure where the nodes are the states $\mathcal{S}$ and there is an edge between two states $s, s' \in \mathcal{S}$ iff there exists an action which leads from s to s'. A state space with all states annotated by a value is a state space topology.

Definition 15.1. State Space Topology (Hoffmann, 2005)
Let Π be a task with states $\mathcal{S}$ which induces a state space $\mathcal{X}(\Pi)$. Let $h : \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ be a deterministic, path-independent state-value function. $T = \langle \mathcal{X}(\Pi), h \rangle$ is a state space topology.

Figure 15.1a shows an example state space topology. Node A corresponds to the initial state and T to the only goal state. The behavior of GBFS within a state space topology is analogous to water in a physical topology. A water flow starts at a source (initial node) and flows to the deepest point of the first valley. If it cannot drain off the deepest point, then it fills up the valley until it overflows into the next valley. The goal node is a drain, which consumes any amount of water. Once the water reaches the goal node, the system reached its terminal state. Without a reachable goal node, the topology fills completely. In planning, we call a valley without goal node a crater. If there are multiple valleys to flow into, GBFS picks one at random. In our example, GBFS expands first A, then picks the crater of B and expands B. Next it picks the crater of C and expands C, E, G, H. Finally, it expands the valley of D which ends in a goal state.

Based on this intuition, Wilt and Ruml (2014) introduced *high-water marks*. The high-water mark of a location is the height the water has to reach until it finds a goal. In planning terms, the high-water mark of a state s denotes the maximum heuristic value that GBFS has to expand to find an s-plan. Every node in Figure 15.1a is annotated with its high-water mark.

Definition 15.2. High-water mark (Wilt and Ruml, 2014)
Let Π be a planning task with a state $s \in \mathcal{S}$ and a topology $T = \langle \mathcal{X}(\Pi), h \rangle$. Let P be



(a) State space topology **(b)** Bench transition system

Figure 15.1: An example state space topology for a fictive heuristic and its induced bench transition system. States annotated with an asterix are progress states. States with double circles are (Left) goal states or (Right) bench exit states.

the set of all acyclic s-plans. The high-water mark of s is

$$hwm(s) = \min_{\langle a_1,\ldots,a_n\rangle \in P} \max_{i=0,\ldots,n} h(s[\![a_1]\!]\ldots[\![a_i]\!])).$$

Let $S \subseteq \mathcal{S}$ be a set of states. The high-water mark of S is

$$hwm(S) = \min_{s \in S'} hwm(s).$$

Using the high-water mark, we define a progress measure for GBFS. Whenever GBFS expanded a state s such that one successor has a lower high-water mark than the heuristic value of s, then GBFS progressed. In simpler terms, whenever GBFS expanded a progress state s, then all states expanded in the future have a lower heuristic value than $h(s)$.

Definition 15.3. Progress State (Heusner, 2019)
Let Π be a planning task with a state $s \in \mathcal{S}$ and a topology $T = \langle \mathcal{X}(\Pi), h\rangle$. The state s is a progress state iff it is a goal state ($s \in \mathrm{S}_G$) or its heuristic value is higher than the high-water mark of its successors ($h(s) > hwm(succ(s))$).

Figure 15.1a indicates the progress states with an asterisk. In our example, all states along the optimal plan are progress states. In practice, all progress states are along a plan, but not necessarily the optimal plan. Additionally, every plan can also visit non-progress states.

With the notion of progress states, Heusner, Keller, and Helmert (2018) map the states of a state space topology into benches. A bench starts at a progress state s, contains all nodes that GBFS could expand without reaching a progress state, and ends at one of the next progress states GBFS could reach.

Definition 15.4. High-Water Mark Bench (Heusner, Keller, and Helmert, 2018)
Let Π be a planning task with a topology $T = \langle \mathcal{X}(\Pi), h \rangle$ and a non-goal progress state $s \in \mathcal{S}$.

The bench *induced by s is $\mathcal{B}(s) = \langle level(s), s, inner(s), exit(s) \rangle$. The* bench level *of $\mathcal{B}(s)$ is $level(s) = hwm(succ(s))$. The* bench entry state *is s itself. Let P be the set of all paths starting from s which do not contain progress states (except for s itself) and all states s' along the paths satisfy $h(s') \leq level(s)$. The* bench inner states *are the set $inner(s) = \bigcup_{p \in P} p \setminus \{s\}$. The* bench exit states *are the set $exit(s)$ consisting of all successors s' of $inner(s)$ and s which are progress states and $h(s') = level(s)$.*

Let s' be a goal state. By definition s' is a progress state. The bench induced by s' is $\mathcal{B}(s) = \langle -\infty, s, \emptyset, \emptyset \rangle$.

We use $states(s) = \{s\} \cup inner(s) \cup exit(s)$ to denote all states of the bench induced by s. The bench $\mathcal{B}(s)$ induces a state space $\mathcal{X}(\Pi)_{|\mathcal{B}(s)} = \{states(s), s, exit(s), succ' : states(s) \mapsto 2^{states(s)}\}$ where $succ'$ is a restriction from the successor function $succ$ of Π to the states $states(s)$.

Figure 15.1b shows the four benches of our example topology and the connections between them. We call this the *bench transition system* (BTS). The BTS $\mathfrak{B}$ of a task Π is a state space. In this state space, every bench is a node and there is an edge from bench b to bench b' if the entry state of b' is an exit state of b. The initial state of $\mathfrak{B}$ is the bench which contains the initial state of Π as inner state. Any bench which contains a goal state of Π as inner state is a goal node for $\mathfrak{B}$. By construction, the BTS of any task is acyclic.

Both the knowledge of progress states and of bench transition systems are interesting from a theoretical point, but are not yet practically useful. We can compute them only *after* we found a plan. At this point the knowledge is futile. We change this. We show how we can learn formulas, which identify progress states or properties of benches, from simple tasks of a domain.

16. Learning Structures

Ståhlberg, Francès, and Seipp (2021) showed that it is possible to learn a formula, which identifies dead-ends in a domain using a single task of the same domain and description logic features as literals. Inspired by this, we modify the approach to learning formulas, which identify progress states. Our formulas are not only interesting from a theoretical point of view. We successfully exploit them as a tie-breaker during search.

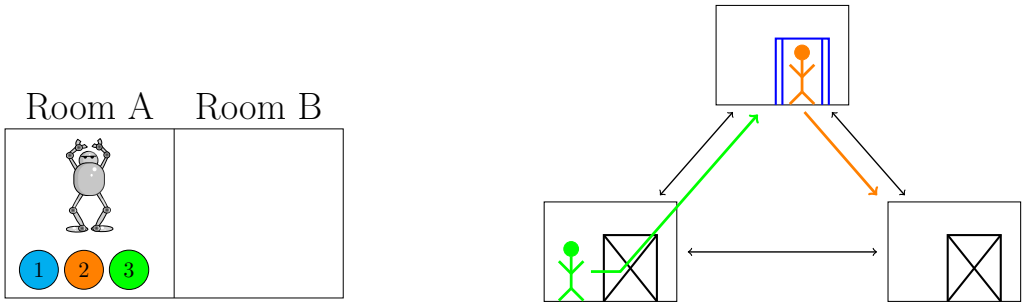
In our ongoing project, we sample the whole bench transition system of a single task and generalize it such that it can be applied to any task of the same domain. To allow this generalization, we describe the properties of a bench using formulas with description logic features as literals.

16.1. Describing Progress States

First, we introduce description logic formulas. Then we use handcrafted formulas to show that these formulas can indeed describe progress states. Finally, we propose a workflow to automatically learn formulas for any domain and any (path-independent) heuristic.

Feature Language. A description logic language (DL, see Chapter 5) describes individuals (constants), concepts (sets of objects), and roles (set of tuples of objects). Furthermore, it defines a set of rules to combine those into new concepts and roles. We follow the instruction from Drexler, Francès, and Seipp (2022) to define and interpret atomic concepts and roles for planning (see Section 5.2) and use the rules from Table 5.1. We use the *size* and *concept distance* function to convert denotations to Boolean values. The PDDL language allows the definition of constant objects for a domain. In theory, we could use these as description logic individuals. In practice, this PDDL feature is rarely used in the IPC domains. For each domain, we manually identify objects which should be constants and use them as description logic individuals.

Our feature language $\mathcal{F}$ is the infinite set of all features which can be constructed using the individuals, the atomic concepts, the atomic roles, the construction rules for complex concepts and roles, as well as, the conversions from concept or role to Boolean. $\mathcal{F}_k$ is a finite subset of $\mathcal{F}$ which contains only features up to complexity k. We call a logical formula where all atomic propositions are from our feature language a description logic formula.



(a) Gripper **(b)** Miconic-STRIPS

Figure 16.1: Visualization of a concrete Gripper and a concrete Miconic-STRIPS task. (Left) The robot is in room A and all balls are on the floor of room A. (Right) The elevator is at the top floor. The orange person boarded the elevator and wants to leave at the right floor. The green person waits at the left floor and wants to leave at the top floor.

Handcrafted Formulas. Now that we defined our feature language, we verify that it is expressive enough to compactly describe progress states for the Gripper and Miconic-STRIPS domain. It depends on a heuristic whether a state is a progress state. For our handcrafted formulas, we use the perfect delete-relaxed heuristic (h^+), because it is intuitive for humans to reason about it. We note that both domains have no local minima (also known as craters) under the h^+ heuristic (Hoffmann, 2011). For all states in Gripper and Miconic-STRIPS, there exists a path which starts in s and ends in a goal state such that the h^+ value decreases monotonically along the path (possibly with plateaus). Thus, for every state, its h^+ value is also its high-water mark under h^+.

We simplify our notation with syntactic sugar. Let s be a state and $\mathcal{I}_s$ be the planning interpretation induced by s. For readability, we write s instead of $\mathcal{I}_s$. Let p be a unary predicate symbol. We define $p(\cdot)^s = C^s_{p,0}$ and $p(o)^s = (C_{p,0} \sqcap \{p\})^s = C^s_{p,0} \cap \{o\}$. Let p be a binary predicate symbol. We define $p(\cdot, o)^s = (\exists R_{p,0,1}.\{o\})^s = \{o' \mid \langle o', o\rangle \in R^s_{p,0,1}\}$. Let X be a role or concept, we define $\bar{X} = \neg X$.

In the Gripper domain, there are two rooms, A and B, as well as some balls. Initially, all balls are in room A. Additionally, there is a robot with two hands, which moves between the rooms, picks up and drops balls. The goal is to move all balls from room A to room B. Figure 16.1a shows an example state.

For every Gripper state, the perfect delete relaxed plan consists of picking up all balls from room A and dropping all balls, which are not yet in room B at room B. Furthermore, if the robot is in room A and there are balls missing in room B, then the plan contains the action to move from room A to room B. If the robot is in room B and there are balls in room A, then the plan contains the action to move to room A.

Thus, there are three kinds of actions in Gripper which make progress. (1) Picking up a ball in room A, (2) dropping a ball in room B, and (3) moving for the last time from

room A to room B. Picking up a ball in room A and dropping a ball in room B removes one action from the perfect delete relaxed plan and does not add any new actions. Thus, the h^+ value of the successor state, as well as the high-water mark, decreases. Thus, the start state is a progress state. If the robot moves from room A to room B and there are still balls in room A, then the successor has the same h^+ value as the current state. This is because the action to move from room A to room B is replaced by the action to move from room B to room A in the perfect delete relaxed plan and both actions have the same cost. Only if there is no ball left in room A, then the robot does not need to return to room A, thus, the action to move from room A to room B is removed without adding a new action. As a consequence, we can describe all progress states with the following conditions:

1. *There are no balls on the floor of room A*: If all balls are in room B, then s is a goal state. Thus, a progress state. If the robot is in room B and carries a ball, it can drop the ball. If the robot is in room A and carries at least one ball, then it moves for the last time to room B.

2. *The robot is in room A, has a free hand and there are balls on the floor of room A*: The robot can pick up a ball.

3. *The robot is in room B and is carries at least one ball*: The robot can drop a ball.

We can describe these conditions as DNF formula in our feature language. A state is a progress state in Gripper if

$$\begin{aligned} s \models\ & (|\mathit{at}(\cdot, roomA)^s| = 0) \vee \\ & ((|\mathit{at\text{-}robby}(roomA)^s| > 0) \wedge (|\mathit{free}(\cdot)^s| > 0)) \vee \\ & ((|\mathit{at\text{-}robby}(roomB)^s| > 0) \wedge (|\mathit{carry}(\cdot)^s| > 0))\,. \end{aligned}$$

In the Miconic-STRIPS domain we control an elevator to transport passengers from their origin floor to their destin floor. The elevator moves directly from one floor to any other floor. We say a floor is required if the elevator still needs to pick up a passenger from this floor or deliver a passenger to this floor. Every perfect delete relaxed plan picks up all passengers who are still at their origin floor, drops all passengers at their destination floor, and moves to all required floors, except for the floor at which the elevator currently is.

Thus, there are three kinds of actions which make progress. (1) Picking up a waiting passenger, (2) dropping a passenger at its destination floor, and (3) moving to a required floor if the current floor is not required anymore. As before, picking up and dropping passengers at the right floor removes actions from the plan. Thus, the successor state has a lower h^+ value. Because there are no local minima, it also has a lower high-water mark. If the elevator is at a required floor f, then moving to another floor f' replaces the action to move to floor f' by the action to move to floor f. Thus, the h^+ value stays

constant and this causes no progress. If the elevator is at a floor which is not required anymore, then moving to a required floor f', removes the action to move to floor f' from the perfect delete relaxed plan. Thus, the h^+ value decreases, thus the high-water mark decreases, and thus, the search progresses. If the elevator moves from a required floor to a not required floor, then the action to return to the previous floor is added to the plan. If it moves from a not required floor to a not required floor, the plan stays unchanged.

Consequently, we can describe the progress states as:

1. *The elevator is at a floor where a person is waiting to board*: The person boards the elevator.
2. *The elevator is at a floor where a boarded person wants to leave*: The person leaves the elevator.
3. *The elevator is at a not required floor*: Either no floor is required, which means all passengers reached their destination or the elevator moves to a required floor where someone can enter or someone can leave. In the former case, the state is a goal state, and thus, a progress state. In the latter case, the relaxed plan shrinks by one action.

Again, we can describe these conditions as DNF. We use f_E as syntactic sugar for the floor at which the elevator is located.

$$\begin{aligned} s \models & (|(\mathit{origin}(\cdot, f_E) \sqcap \overline{\mathit{boarded}(\cdot) \sqcup \mathit{served}(\cdot)})^s| > 0) \vee \\ & (|(\mathit{destin}(\cdot, f_E) \sqcap (\overline{\mathit{served}(\cdot)} \sqcup \mathit{boarded}(\cdot)))^s| > 0) \vee \\ & (|(\mathit{destin}(\cdot, f_E) \sqcap \overline{\mathit{boarded}(\cdot) \sqcup \mathit{served}(\cdot)})^s| = 0) \end{aligned}$$

We empirically verified that our formulas are correct. We computed the progress state labels on all states from multiple tasks of these domains and checked that those labels agree with the results of our formulas. These two examples show that our feature language is expressive enough to identify progress states in some domains. Of course, this does not show that our feature language is expressive enough for all domains.

Training Workflow. We manually designed compact DNFs for Gripper and Miconic-STRIPS. This hides that we tried to construct formulas for the ChildSnack and Hiking domains and failed. Even for the intuitive h^+ heuristic, it is challenging to consider all states and to come up with a good formula. Thus, we present a workflow, which automatically learns them. Our workflow consists of three steps: sampling and labeling states; generating features; learning a formula.

Let Π be a planning task and h be a path-independent heuristic. Let s_I be the initial state of Π. Let $s \in \mathcal{S}$ be a state of Π, $succ(s)$ is the set of all successors of s, $pred(s)$ is the set of all predecessors of s. First, we generate the set $\mathcal{S}_R$ of states reachable from

s_I. Next, we calculate the high-water marks. For every state $s \in \mathcal{S}_R$ we initialize its high-water mark as

$$hwm(s) = \begin{cases} h(s) & \text{if } s \supseteq \delta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We define a priority list pq increasingly sorted by high-water mark and initialize it with the goal states. As long as pq is not empty, we pop the next state s from it. We update the high-water mark of all predecessors $p \in pred(s)$ with undefined high-water mark as

$$hwm(p) = max(h(p), hwm(s))$$

and add it to the priority queue. The high-water mark of a state is the maximum over its heuristic estimate and the minimum high-water mark of its successors. Thus, it suffices to update the high-water mark of a state once we popped its successor with minimal high-water mark. When pq is empty, then all dead-end states have an undefined high-water mark. For every dead-end state d we update their high-water mark to $hwm(d) = \infty$. In the last step, we label all goal states and all states which have a successor with lower high-water mark as progress states and all other states as non-progress states.

Now, we have a set of labeled states. Next, we generate features to describe them. As the size of our feature language is infinite, we cannot generate all features. Furthermore, many concepts and roles evaluate to the same denotations, e.g., let C_1 and C_2 be two concepts, then the complex concepts $C_1 \sqcap C_2$ and $C_2 \sqcap C_1$ evaluate to the same denotation.

We take a set of generated states, possibly from multiple tasks and ignore the labels. We iteratively, construct all features of complexity $K = 2$, then complexity $K = 3$, and so forth until we reach a predefined complexity limit or a time limit. In every iteration, we generate all concepts and roles up to complexity $K-1$ and convert them to features, which have at most complexity K. Converting a concept or role to a Boolean feature increases the complexity by at least 1. If a generated concept or role evaluates to the same denotation on *all* states as a previously generated concept or role, then it is pruned. If a generated Boolean feature evaluates to the same value for all states, then it is pruned. As both pruning steps operate on selected training tasks and optionally a subset of the states of these tasks, it is possible that we incorrectly prune concepts, roles, and features. But without pruning, the number of generated features is infeasible large.

We now have a huge set of labeled states and a finite set of features $\mathcal{F}' \subset \mathcal{F}$. We construct a data set $D = \{\langle \vec{x}_1, y_1 \rangle, \ldots, \langle \vec{x}_n, y_n \rangle\}$. For every labeled state s, its feature vector $\vec{x}$ is the evaluation of all features on s and its label y indicates whether it is a progress state. We can use this data set with any common machine learning technique.

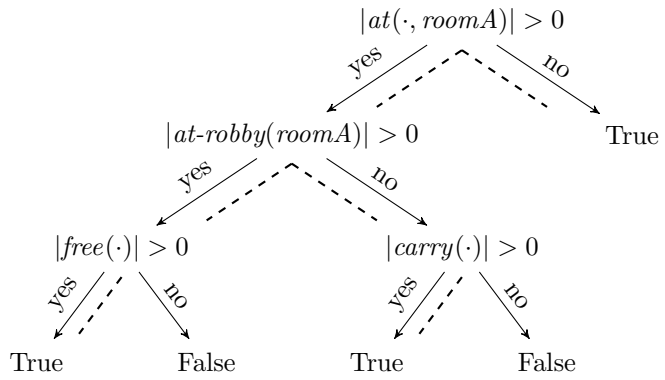


Figure 16.2: Example of a trained decision tree to identify progress states. The branches which lead to positive predictions are annotated with dashed lines).

We expect that we are not always able to perfectly separate progress states from non-progress states. The feature generation could have terminated before a necessary feature was generated, or our feature language is not expressive enough. Furthermore, we want to extract a relative compact DNF formula from our model. Thus, we train a decision tree. We do not limit the tree complexity or tree depth. If the labels can be perfectly separated by the features, the tree will perfectly separate the labels. Figure 16.2 shows a tree for h^+ on the Gripper domain. Furthermore, it highlights all branches which predict that a state is a progress state. To convert the decision tree to a DNF formula, we first convert each highlighted branch to a conjunction of features. For those branches, an internal node i becomes a literal l. If the branch takes the "no" route, at node i, then we add $\neg l$ to the conjunction of this branch. Otherwise, we add l. Next, we join all branches with a logical or.

The resulting formulas can be unnecessarily complicated. We simplify them with an automatic tool. Such a tool works only if the same idea is described with the same feature. Whenever we construct a new internal node and two features separate the training data equally well, we favor features, which are already used by other internal nodes. If there are still multiple candidates, then we favor the feature with the lowest complexity. A decision tree does not require Boolean features. Restricting the features to Boolean also improved the ability to simplify the final formulas.

Use Cases. During search, we exploit the knowledge whether a state is a progress state. Remember that whenever GBFS expands a progress state, it afterwards expands only states with lower heuristic values. The open list of GBFS is sorted by heuristic values. That means whenever it expands a progress state, it will never expand a state which is currently in the open list. Thus, we can simply clear the open list while expanding a progress state. A planner with a fast heuristic quickly runs out of memory due to the

size of its open list. Clearing the open list solves this issue. There is a drawback. If we incorrectly classify a state as a progress state, then the search may become incomplete.

The previous use case is only important once the search approaches the memory limit. We can already use the progress state information earlier. Whenever we expand a progress state, we know that we progress. Whenever we expand another state, we do not necessarily progress. Thus, if two states have the same heuristic values, one of them is classified as a progress state and the other is not, then we expand first the state labeled as progress state. This is less impactful, but GBFS stays complete.

16.2. Constructing Generalized Bench Transition Systems

The knowledge of progress states is already useful, but knowing the bench transition system (BTS) of a task Π, is even more powerful. Instead of finding a solution for the task Π, we can split the search into smaller and simpler searches. Given the BTS of Π, we identify the bench b which has the initial s_1 state as an inner state or entry state. Our subtask Π_b is to find a path from the initial state to an exit state of this bench. The detected goal state of Π_b becomes the initial state of our next subtask. This procedure repeats until we end in a goal state of Π.

The GripperOne domain is equivalent to the Gripper domain, except that the robot has only a single arm. This reduction reduces symmetries and makes the BTS of its tasks easier to visualize. Figure 16.3 shows the first six out of eight bench levels of the BTS from the GripperOne domain for the perfect delete-relaxed heuristic on a task with three balls. Following the definitions, we observe that there exists one non-progress state, which belongs to no bench (the entry state of the single bench with level seven). We construct an exceptional bench for this state and defer the handling of these states to a later point. In every bench, the top most state is the entry state, and the bottom most states are the exit states. We observe that all three benches with a level of five are symmetric, i.e., they are equivalent up to renaming the balls. The same holds for all three benches with a level of four, for all six benches with a level of three, for all three benches with a level of two, for all three benches with a level of one, and for all three benches with a level of 0. Our first generalization of this BTS removes this symmetry. There is a single bench with a level of five. Any state which has two balls in room A, the robot in room B, and the robot is holding the third ball belongs to this bench.

Secondly, we see that the states in the benches with a level of four and six share many similarities. The action applied to progress is the same up to renaming objects. Our second generalization merges benches with the same "goal descriptions". For all those benches, we describe the goals as *the robot is in room B and holds a ball*. Using these observations, we define *generalized high-water mark benches*.

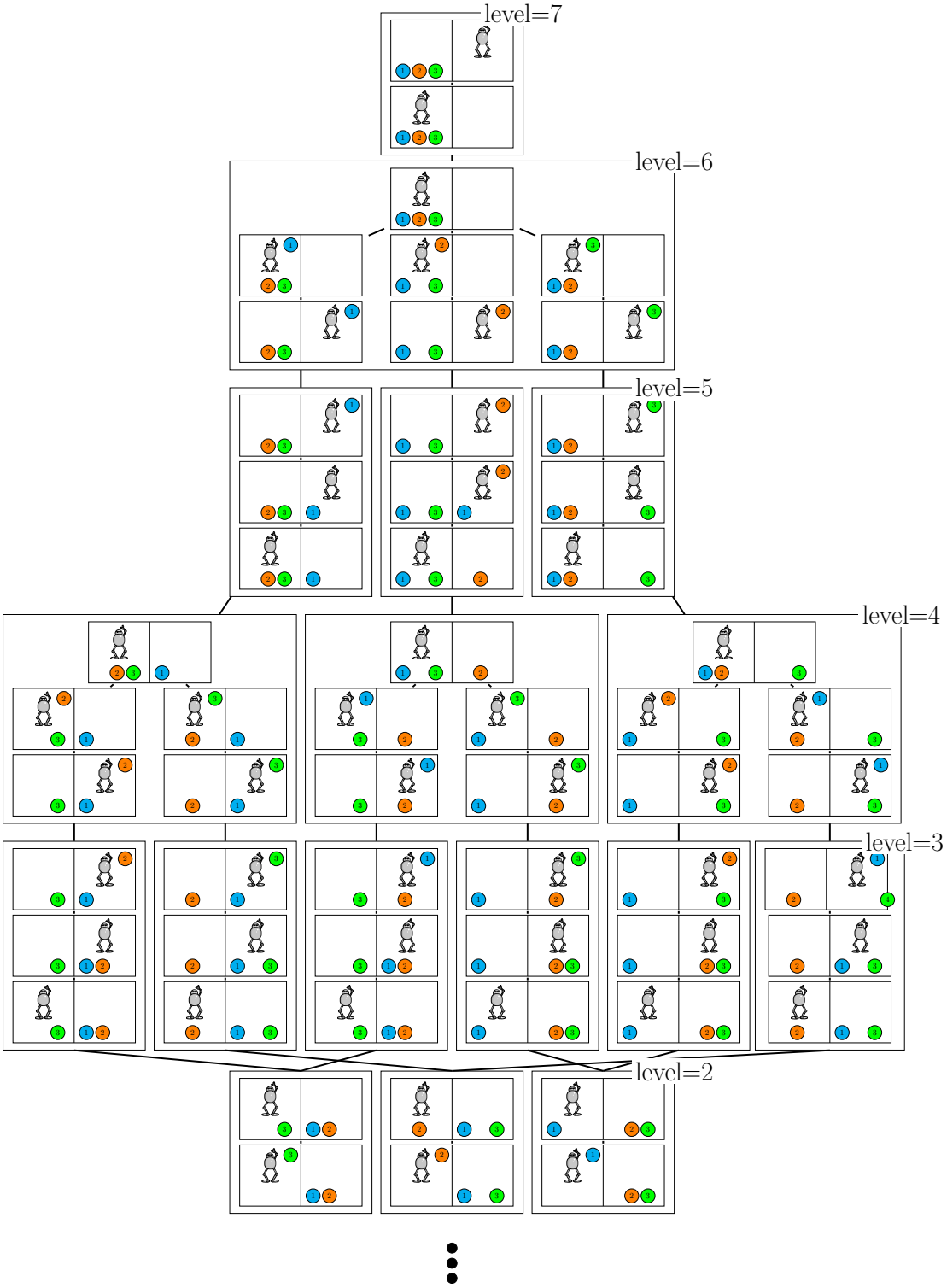


Figure 16.3: Bench transition system for a GripperOne task with three balls using the perfect delete-relaxed heuristic. The last layers are omitted.

Definition 16.1. Generalized High-Water Mark Bench
Let $\mathcal{D}$ be a planning domain. A generalized bench $\mathcal{G}$ for D is a tuple $\mathcal{G} = \langle \phi_{membership}, \phi_{inner\ goal} \rangle$. $\phi_{membership}$ and $\phi_{inner\ goal}$ are two FOL formulas. Let Π be a task from $\mathcal{D}$ with states $\mathcal{S}$. Any state $s \in \mathcal{S}$ for which $\phi_{membership}$ holds ($s \models \phi_{membership}$) is a member *state of the bench. Any state $s \in \mathcal{S}$ which is a member of $\mathcal{G}$ and for which $\phi_{inner\ goal}$ holds ($s \models \phi_{membership} \wedge \phi_{inner\ goal} = \phi_{outer\ goal}$) is a* goal *state of the bench. Any state $s \in \mathcal{S}$ which is a member of the bench, but not a goal state is an* inner *state ($s \models \phi_{membership} \wedge \neg\phi_{inner\ goal}$). For simplicity, we define $\phi_{inner} = \phi_{membership} \wedge \neg\phi_{inner\ goal}$.*

Furthermore, we also generalize the notion of a bench transition system.

Definition 16.2. Generalized Bench Transition System (GBTS)
A generalized bench transition system for a domain $\mathcal{D}$ is a graph $\mathfrak{G} = \langle V, E \rangle$ with a set of generalized benches V and a set of edges between those benches E. For all tasks Π of the domain $\mathcal{D}$ and every state $s \in \mathcal{S}_\Pi$ holds that s is either an inner state of exactly *one bench or s is a goal state ($s \supseteq \delta$). Let $\mathcal{G}_1, \mathcal{G}_2 \in V$ be two generalized benches. There is an edge from $\mathcal{G}_1$ to $\mathcal{G}_2$ iff there exists a task Π with a state $s \in \mathcal{S}_\Pi$ such that s is a goal state of $\mathcal{G}_1$ and an inner state of $\mathcal{G}_2$.*

Figure 16.4 shows a possible GBTS for GripperOne. For simplicity, we constructed the GBTS for the perfect heuristic (h^*). For each bench, the formulas $\phi_{membership}$ and $\phi_{inner\ goal}$ are provided. For convenience, the formulas ϕ_{inner} and $\phi_{outer\ goal}$ are also provided. The first four benches form a cycle. In every iteration of the cycle, the robot moves one ball from room A to room B. The GBTS requires loops! Without loops, the GBTS cannot adapt to tasks with an arbitrary number of balls.

Proposition 16.1. Loopy Generalized Bench Transition Systems
There exists a domain $\mathcal{D}$ and a heuristic h such that any GBTS $\mathfrak{G}$ for them is cyclic.

Proof:
Let $\mathcal{D}$ be the GripperOne domain and h be the perfect heuristic. Let Π_b be a GripperOne task with $b \in \mathbb{N}$ balls and $\mathfrak{B}$ be its bench transition system. Π has a state where all balls are in room A and the robot is in room B. This state has an optimal plan cost of $4b$. The robot has to move b balls from room A to room B. Moving a single ball requires three actions (picking up a ball in room A, moving to room B, dropping the ball in room B). To pick up the first ball, the robot has to move to room A. After dropping a ball, except for the last one, the robot is in room B and has to move back to room A. At every step of along the optimal plan π, the heuristic value drops by one. By the definition of progress states, every state along the optimal plan is a progress state. By the definition of the bench transition system, each of these progress states s' forms a bench where s' is the entry state and its successor along the optimal plan π is an exit state. Thus, the longest acyclic path in the BTS $\mathfrak{B}$ has a length of at least $4b$.

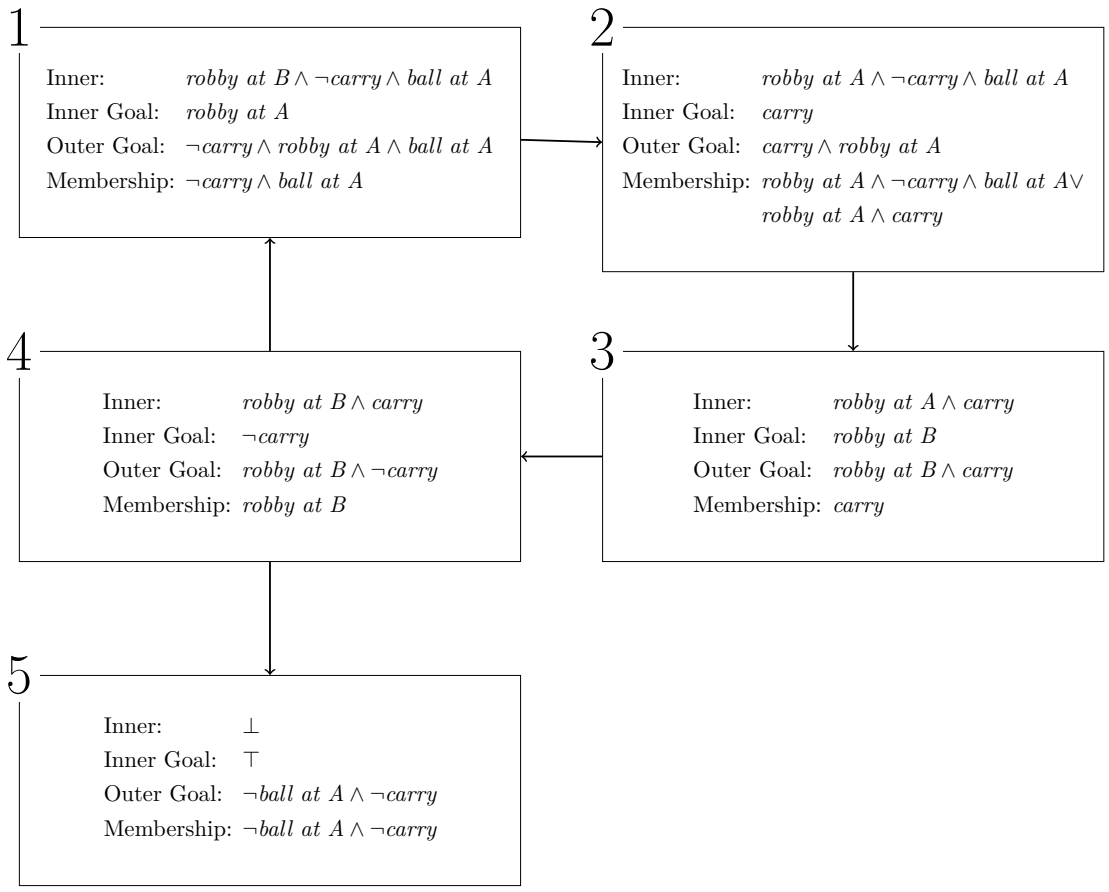


Figure 16.4: Generalized bench transition system for the GripperOne domain using the perfect heuristic.

Assume there exists an acyclic generalized bench transition system $\mathfrak{G}$ for GripperOne using the perfect heuristic. By the definition of GBTS, the number of generalized benches n in $\mathfrak{G}$ is finite.

Let Π_b be a GripperOne task with $b \in \mathbb{N}$ balls and $4b > n$. The longest acyclic path in the $\mathfrak{B}$ of Π_b has a length of at least $4b$. As $\mathfrak{G}$ is acyclic and has fewer than $4b$ generalized benches, there exists no assignment from the benches of $\mathfrak{B}$ to the generalized benches of $\mathfrak{G}$ which preserves the transitions between the benches. This violates the assumption. Thus, all $\mathfrak{G}$ for GripperOne using h^* are cyclic. □

Construct Generalized Bench Transition System. We propose a greedy approach, which simplifies the BTS $\mathfrak{B}$ of a task Π using heuristic h until it becomes a GBTS for the domain. The process has two major assumptions. First, it assumes that the BTS $\mathfrak{B}$ contains all benches necessary to generalize. For example in the GripperOne domain with the perfect heuristic, we cannot learn a GBTS, if the task contains only a single ball. Then the information that we have to return to room A is missing. Secondly, we assume that we have the features to represent all the necessary knowledge. In reality, our feature language might not be expressive enough, or we are unable to generate the necessary features within the computational limits. Our workflow consists of four steps: sampling the BTS, generating description logic features, simplifying the BTS, generating description logic formulas for the generalized benches.

In the first step, we construct the BTS $\mathfrak{B}$ for Π. As previously, we sample all states $\mathcal{S}_R$ reachable from the initial states s_I. We calculate for all of them their high-water mark, and we label them as progress or non-progress states. Then we construct for every progress state s its bench $\mathcal{B}(s)$ as described by Definition 15.4. We observe that there are non-progress states which belong to no bench (see the entry state of the top most bench in Figure 16.3). As long as there is a state s' which belongs to no bench, we construct an additional bench $\mathcal{B}(s')$ relaxing the Definition 15.4. $\mathcal{B}(s')$ has no entry state. Instead s' is added to its inner states.

In our second step, we use the set of reachable states S_R to generate Boolean description logic features. We use the same iterative approach, and the same pruning techniques which we used for learning progress state formulas. This step ends with a finite set of features $\mathcal{F}' \subset \mathcal{F}$.

In the third step, we transform the BTS until we reach a fix point. At the beginning of every iteration, we calculate for every bench b its inner goal formula $\phi_{inner\ goal}$. For this purpose, we adapt the approach from learning progress state formulas. We learn the inner goal formula for bench b using only the member states. For all member states of b, we construct a feature vector by evaluating every feature $f \in \mathcal{F}'$ on it. We label every member state as 1, if it is an exit state of b, and as 0 otherwise. On these data, we learn a decision tree, extract a DNF formula, and simplify it. As for the progress states, if two features separate the training data equally well, we prefer the feature which was already used in this *iteration*! Thus, the feature preference persists when learning the

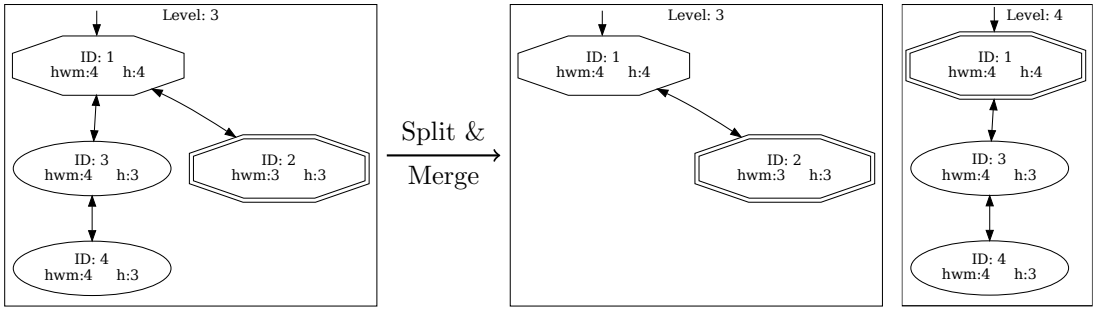


Figure 16.5: Example of the bench splitting in our generalization workflow. Inner states have a round border. Progress states have an octagonal border. Progress states with a double border are bench exit states. States 3 and 4 would be removed from this bench and form a new bench with the exit state 1.

inner goal formulas for all benches of one iteration. This is important as it simplifies merging benches based on the same inner goal formulas.

To merge benches, we relax the definition of a bench. A relaxed bench has an arbitrary number of levels and entry states. A relaxed bench $b = \langle levels_b, entry_b, inner_b, exits_b \rangle$ is a tuple where $levels_b$ is a set of bench levels, $entry_b$ is a set of entry states, $inner_b$ is a set of inner states and $exits_b$ is a set of exit states. Every bench $b \in \mathfrak{B}$ induces a relaxed bench. Let $b_1, b_2 \in \mathfrak{B}$ be two relaxed benches. Merging those constructs a new relaxed bench $b' = \langle levels_{b_1} \cup levels_{b_2}, entry_{b_1} \cup entry_{b_2}, inner_{b_1} \cup inner_{b_2}, exits_{b_1} \cup exits_{b_2} \rangle$. We merge any pair $b_1, b_2 \in \mathfrak{B}$ of relaxed benches, if one of the following three conditions holds:

1. The member states of b_1 are a subset of the member states of b_2 or vice versa.
2. The goal states of b_1 are a subset of the goal states of b_2 or vice versa.
3. b_1 and b_2 have the same goal formula.

We do not only merge benches, but we also split benches. If a relaxed bench b contains a non-progress states s such that all paths from s to any exit state of b contain an entry state of b, then we remove this state s from the bench b and construct a new relaxed bench b'. The relaxed bench b' has no entry states. All non-progress states of b reachable from s without passing through progress states are its inner states. All progress states of b reachable from the inner states of b' are its exit states. If this removes multiple states from the same bench, then the constructed benches are often similar and can be merged. Figure 16.5 shows an example. We repeat the calculation of the goal formulas, the merging, and the splitting until we reach a fix point.

In the last step, we learn for every bench b the membership formula $\phi_{membership}$ using all reachable states $\mathcal{S}_R$ as training data and labeling all member states of b positively and

all other states negatively. We learn the goal formula $\phi_{inner\ goal}$ using all member states of b. We label all exit states of b positively and all other states negatively. Furthermore, we explicitly learn the inner formula ϕ_{inner} using all reachable states $\mathcal{S}_R$ and labeling the inner states of b positively and all other states negatively. Finally, we observe that the inner goal formula loses information, because it is implied in the membership formula. To make this knowledge explicit, we learn an outer goal formula $\phi_{outer\ goal}$ using all reachable states $\mathcal{S}_R$ and labeling the exit states of b positively and all other states negatively. In this last step, we do not need to merge benches with the same inner goal formulas. Thus, we do not track the previously used features across benches, but we track them only within a single tree. Building a decision tree is a greedy approach. It does not always build the most intuitive trees. We evaluate the complexity of a tree as the sum of the complexities of the used features. Furthermore, we extract the maximum complexity k of a used feature. Now, we iteratively train new decision trees restricted to features of complexity less than k, less than $k-1$, and so forth. Whenever we obtain a new tree which also perfectly classifies the states and has a lower sum of complexities, we use this tree as a candidate for extracting the formula.

The final set of relaxed benches annotated with their formulas is the generalized benches of the GBTS for the domain $\mathcal{D}$ of the initially provided task Π and the heuristic h.

Formula to PDDL. We use some formulas during search. Thus, we have to convert them to $PDDL$. We show how to convert a description logic feature to first-order logic (FOL). Rephrasing them from FOL to $PDDL$ is trivial. Let $C_{p,i}$ be an atomic concept for the predicate symbol p with $arity(p) = k$. The feature $|C^s_{p,i}| > 0$ expresses that the denotation of $C_{p,i}$ for state s contains at least one element.

$$\begin{aligned}|C^s_{p,i}| > 0 &= |\{x \mid \exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k) \in s\}| > 0 \\ &\overset{FOL}{\rightsquigarrow} \exists x \exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k)\end{aligned}$$

The condition to have at least one x is translated into the first existential quantifier. The quantifiers introduced by the atomic concept follow afterwards. The actual condition comes last. For an atomic role, the schema is the same. Let $R_{p,i,j}$ be an atomic role with $i < j$.

$$\begin{aligned}|R^s_{p,i,j}| > 0 = |\{\langle x, y\rangle \mid &\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k : \\ &p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k) \in s\}| > 0 \\ \overset{FOL}{\rightsquigarrow} \exists x, y \ &\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k : \\ &p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k)\end{aligned}$$

The adaption for an atomic role with $i > j$ is trivial. For the translation in the general case, we introduce three recursive function. f translates a feature. f_C translates

a concept and receives as an additional input the name of the variable bound on the outside (red part in previous atomic concept example). f_R translates a role and receives as additional input the names of the two variables bound on the outside (red part in the previous atomic role example). For every rule which constructs a complex concept, f_C requires an overload. For every rule which constructs a complex role, f_R requires an overload. Let X be either a concept or a role, D, E be concepts, and S, T be roles. Let $C_{p,i}$ be an atomic concept and $R_{p,i,j}$ be an atomic role. Then we recursively define f, f_C, f_R as follows:

$$
\begin{aligned}
f(\neg|X^s|>0) =& \neg f(|X^s|>0)\\
f(|C^s|>0) =& (\exists x : f_C(C^s, x))\\
f(|R^s|>0) =& (\exists x, y : f_R(R^s, x, y))
\end{aligned}
$$

$$
\begin{aligned}
f_C(C^s_{p,i}, x) =& (\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k))\\
f_C(\neg D, x) =& \neg(f_C(D, x))\\
f_C(D^s \sqcap E^s, x) =& f_C(D, x) \wedge f_C(E, x)\\
f_C(D^s \sqcup E^s, x) =& f_C(D, x) \vee f_C(E, x)\\
f_C((\exists S.D)^s, x) =& (\exists v : f_R(S^s, x, v) \wedge f_C(D^s, v))\\
f_C((\forall S.D)^s, x) =& (\forall v : f_R(S^s, x, v) \implies f_C(D^s, v))
\end{aligned}
$$

$$
\begin{aligned}
f_R(R^s_{p,i,j}, x, y) =& (\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k :\\
& \quad p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k))\\
f_R(\neg S, x, y) =& \neg(f_R(S, x, y))
\end{aligned}
$$

We observed no further rules in our exploratory experiments. Thus, we did not yet define further overloads. Extending the overloads is straight forward.

Bench Walking. We constructed a generalized bench transition system $\mathfrak{G}$ for the domain $\mathcal{D}$ and the heuristic h. Let Π be a task of $\mathcal{D}$. The standard way of solving Π executes a GBFS using the heuristic h. We use our generalized bench transition system $\mathfrak{G}$ to split the original search into simpler subsearches. We iteratively identify to which bench a state belongs and search for a path of the state to an exit state of the bench. At one point, the exit state is a goal state of the original task. We call this procedure *bench walking*. In practice, our GBTS can be imperfect. Thus, a state can be identified as an inner state of multiple or even of no bench. To be more robust, we extend our algorithm with back tracking.

Algorithm 2 shows our bench walking algorithm. We modify the given Π for the subsearches. Thus, we remember the original goal of Π (line 2). Furthermore, we have

Algorithm 2 Bench Walker. For a given planning task Π with an initial state s_{I} and a goal δ, a generalized bench transition system $\mathfrak{G}$, and a heuristic h.

```
1:  procedure BENCH WALKER
2:      δ_global ← Π.δ
3:      π_global ← Stack()
4:      choices ← Stack()
5:      inits ← Stack()
6:      closed ← ∅
7:
8:      inits.push(Π.s_I)
9:      identify ← True
10:     while inits.top() ⊉ δ_global do
11:         s'_I ← inits.top()
12:         // Find bench for next subsearch
13:         if identify then
14:             B ←identify_benches(s'_I, 𝔊)
15:             choices.push(B)
16:         while choices.top().empty() do
17:             choices.pop()
18:             if choices.empty() then
19:                 return unsolvable
20:             π_global.pop()
21:             inits.pop()
22:             s'_I ← inits.top()
23:         b ← choices.top().pop()
24:         if ⟨s'_I, b⟩ ∈ closed then
25:             identify ← False
26:             continue
27:         else
28:             closed ← closed ∪ {⟨s'_I, b⟩}
29:
30:         // Setup and execute subsearch
31:         Π.s_I ← s'_I
32:         Π.δ ← to_pddl(b.outer_goal)
33:         avoid ← to_pddl(¬b.membership)
34:         π ← GBFS(Π, avoid)
35:         if π then
36:             π_global.push(π)
37:             inits.push(s'_I⟦π⟧)
38:             identify ← True
39:         else
40:             identify ← False
41:     return concatenate(π_global)
```

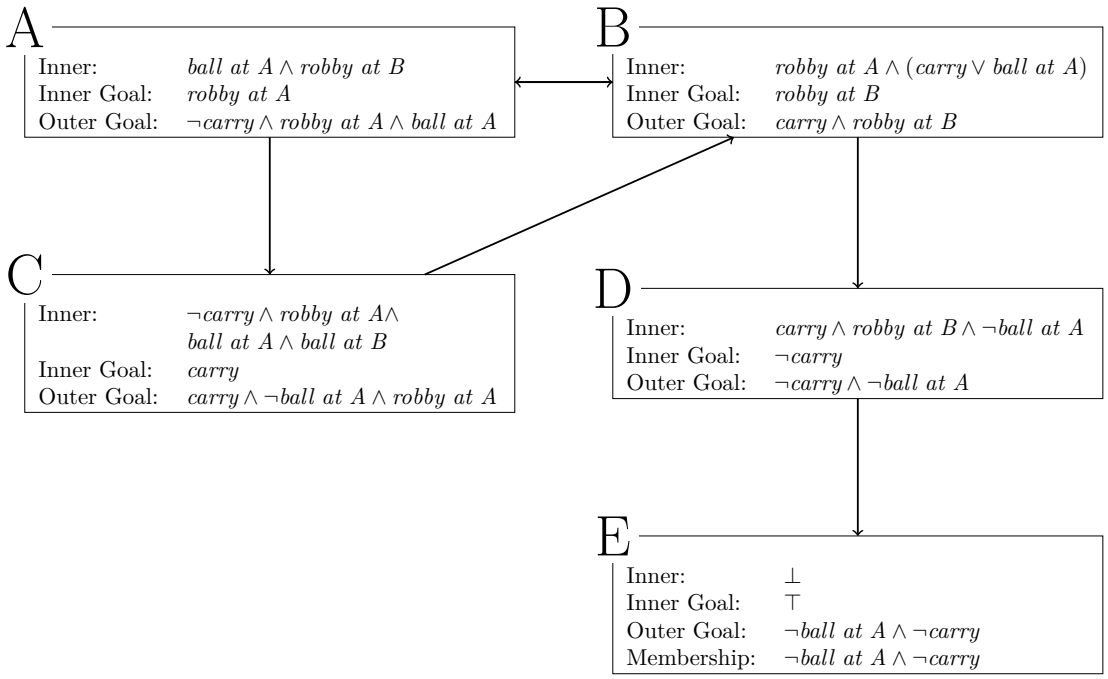


Figure 16.6: Generalized bench transition system for GripperOne using the perfect delete-relaxed heuristic.

a stack, which keeps track of the plans of the previous subsearches; we have a stack, which keeps track of the alternative bench options for each subsearch so far; and we have a stack for the initial state of the subsearches (lines 3–5). To detect whether we walk in a cycle, we remember for all subsearches their initial state - bench of pair (line 6).

The first step of our bench walker is to identify the initial state and bench to use for the next subsearch. In the simplest case, we have the current initial state s'_{I}, identify exactly one bench b which has s'_{I} as inner state (lines 14 & 22) and we did not walk in a cycle (line 27). Then we remember that we will execute a subsearch for the pair $\langle s'_{\mathrm{I}}, b\rangle$ (line 28).

In the second step, we set up and execute the subsearch (lines 30–34). We first update the initial state of the Π to s'_{I} (line 31). Then we construct from the outer goal formula of the bench the next subgoal (line 32). The inner goal formula incorrectly assumes that the subsearch visits only member states of the current bench. Thus, it learns a simpler formula, which falsely identifies non-member states as goal. Furthermore, the additional information in the outer goal formula improves the guidance of the heuristic.

Example 16.1. Superiority of Outer Goal Formula over Inner Goal Formula
Figure 16.6 shows the GBTS for GripperOne under the h^+ heuristic. Let s be a state where there are balls in room A, the robot is in room B, and the robot carries a ball. This state belongs to bench A. The intention of this bench is to drop the ball first in

room B and then to move to room A. The inner goal formula only requires that the robot in room A. A GBFS on this goal, ends in a state where the robot is in room A and still carries a ball. This plan produces the opposite of progress. The outer goal formula says additionally that the robot is not *carrying a ball. As a consequence, the robot drops its ball either in the current room B and moves then to room A or it moves to room A and drops the ball there. Both resulting states are members of bench A. The first state is actual progress. The second state is again a step backwards.*

The example shows that even with the better guidance of the outer goal formula, the subsearch can reverse progress. This is because multiple concrete benches $\hat{b}_1, \ldots, \hat{b}_n$ are merged into the current generalized bench b. As long as the subsearch is allowed to visit *any* state, we might start in a state of the concrete bench $\hat{b}_n$ and end in a goal state of another concrete bench $\hat{b}_1$. To prevent this, we construct an *avoid condition* (Steinmetz et al., 2022b) using the negated membership formula of b (line 33). During search, we prune all states which satisfy the avoid condition. In other words, the search cannot leave the member states of the current bench. *The issue still persist if a GBTS has generalized benches with self-loops.*

Finally, we execute the GBFS subsearch on the modified task Π with the avoid condition. If we find a plan, we remember it and construct the initial state for the next iteration (lines 35–38). If we find no plan and multiple benches where identified for the current initial state, then we start a new subsearch for the next bench. If there are no further benches for the current initial state, then we backtrack.

Previously, we assumed the simplest setting for the first step. The first step becomes more complicated once we identify multiple benches for the current initial state, once we have to backtrack, or once we detect that we walked in a circle. The following paragraph is an explanation of these more complicated cases. Whenever we reach a new initial state s'_I ($identify \leftarrow True$), we first identify all benches B which have s'_I as an inner state and push this bench set as options to our $choices$ stack (line 13-15). Let us assume that B is not empty. Then, the next step is to select and pop the first bench b from B (line 23). If we have already executed a subsearch for s'_I on bench b (line 24), then we select and pop the next bench b from B. This repeats until either B is empty, or we obtain a new pair $\langle s'_\mathrm{I}, b \rangle$. In the former case, we backtrack (lines 16–22), i.e., we remove the empty option set from our stack of $choices$. Thus, the previous set of bench choices is revealed. As we choose a new bench for the previous choice, we remove the plan stored for the previous choice (line 20) and also update the current initial state (lines 21–22). If the set of previous choices is also empty, we continue to backtrack. If it is not empty, we pick the next choice, but also check whether we walk in a cycle. At one point, all choices are either exhausted (line 18) and we terminate without a solution, or we found a new pair of initial state s'_I and bench b.

17. Experiments

We proposed an approach, which learns from simple tasks of a domain a compact formula to identify progress states in any tasks of the same domain. Now we evaluate the workflow and show that the learned formulas make the search more informed. Furthermore, we presented ongoing work, which constructs a generalized bench transition system for a domain. We will present the current results. To generate the description logic features and to evaluate them during search, we build upon the *DLPlan* library by Drexler, Francès, and Seipp (2022).

17.1. Characterize Progress States

We evaluate our approach on the Barman, Blocksworld, ChildSnack, Driverlog, Floor-Tile, Gripper, Miconic-STRIPS, and VisitAll domains. For every domain, we define a parameter space (e.g., for Gripper a range for the number of balls) and generate tasks for all parameter combinations in our parameter space (Seipp, Torralba, and Hoffmann, 2022). From these tasks, we randomly pick 5 as training tasks and all others become our validation tasks. As test tasks, we use all tasks of our domains from the optimal and satisficing Autoscale 21.11 benchmark sets (Torralba, Seipp, and Sievers, 2021). The test tasks have strictly larger parameters than the training and validation tasks. We execute all experiments on a single core of an Intel Xeon Silver 4114 CPU.

State Space Labeling. First, we compute the progression state labels for all states in the state spaces of the training and validation tasks. We use a memory limit of 3.5 GB and a time limit of 5 hours. As heuristics, we use the intuitive perfect delete-relaxed heuristic (h^+ Hoffmann and Nebel, 2001) with the operator counting implementation of Imai and Fukunaga (2015). Due to its complexity, h^+ is not practically useful. We also use the FF heuristic (h^{FF} Hoffmann and Nebel, 2001).

Feature Generation. In our second step, we generate the description logic features. We use the same rules to form complex features as Francès, Bonet, and Geffner (2021), and we use the *size* and the *concept_distance* to convert the denotation of a concept or role to a Boolean. We generate features for 24 hours using 3.5 GB of memory. There is no limit on the feature complexity.

When we use a single state space to generate the features, we observe that the feature generation overfits to the state space. Features, which should be distinct, are not

	Max. Complexity					# Features	
Domain	1.	2.	3.	4.	5.	Min	Max
Barman	7	6	6	6	6	1218	2455
Blocksworld	10	10	10	10	10	15332	16611
ChildSnack	8	8	7	7	7	444	532
Driverlog	9	8	8	7	7	892	1313
FloorTile	8	8	7	7	7	1644	3441
Gripper	12	12	9	9	9	422	1656
Miconic-STRIPS	8	8	7	7	7	332	494
VisitAll	12	12	11	11	11	1692	2118

Table 17.1: The maximum complexity of a generated features and the maximum and minimum number of features generated across the five training sets for each domain and data set when using the FF heuristic.

recognized as distinct and are incorrectly pruned. Thus, we select five training tasks per domain. For each domain, we generate five feature sets. The i-th feature set of a domain is generated from the first i training tasks of that domain, i.e., there is one feature set generated using only the first task, there is one feature set generated using the first two tasks, ..., there is one feature set generated using all training tasks.

We also observe that the feature generation is memory intensive and requires more memory the more states are used. Furthermore, even simple planning tasks can have an enormous number of states. Often, we cannot even execute the feature generation on a single training task. Thus, we sample states from the training tasks. To generate the i-th feature set, we sample up to 10,000 progress states and up to 10,000 non-progress states from each of the first i training tasks.

Table 17.1 shows for each domain and each feature sets, the maximum complexity of a feature generated when using the h^{FF} heuristic to calculate the high-water mark labels. We see that the more training tasks we used for generating the features, the lower is the maximum complexity among the generated features. This is because every newly generated feature is evaluated on every state. The more states we use the higher is the computational burden. Missing features of high complexity can be problematic if the domain requires a certain feature to encode important knowledge. On the other hand, the fewer states we use the more features are incorrectly pruned. With too few states, an important feature could be pruned. As a result, there is a trade-off between generating features with higher complexity and reducing the number of incorrectly pruned features. We also see that the number of generated features varies significantly across domains. We generated around 15,000 features in Blocksworld, but only 300–500 in Miconic-STRIPS. Using the FF heuristic leads to qualitatively similar results (see Table 17.1).

	Mean Validation F1					First DNF			Best DNF		
Domain	1.	2.	3.	4.	5.	$\mathcal{K}(F)$	Clauses	Literals	$\mathcal{K}(F)$	Clauses	Literals
Barman	79	80	81	79	77	7	316	4059	6	1459	25189
Blocksworld	83	–	–	–	–	10	72	1009	10	72	1009
ChildSnack	70	65	73	81	72	8	28	240	7	116	1243
Driverlog	81	81	89	85	85	9	95	829	8	1087	17270
FloorTile	75	85	85	85	88	8	10	91	7	65	926
Gripper	96	96	96	98	98	11	3	3	9	2	4
Miconic-STRIPS	98	98	97	98	99	8	1	1	7	23	274
VisitAll	70	70	73	73	73	12	2413	42496	11	3744	74060

Table 17.2: On the data sets using the FF heuristic. (Left) For each domain and learned formula the F1 score on the validation data. (Middle) For the formula trained on a single state space the maximum complexity of a used feature as well as the number of clauses and literals in the formula. (Right) For the best learned formula with respect to the F1 validation score the maximum complexity of a used feature as well as the number of clauses and literals in the formula.

Learning Formulas. In the last step, we learn for every feature set i a description logic formula, which describes the progress states. We train the formula on the states from the first i training tasks. Some training tasks have significantly more states than other tasks. Thus, the influence of the other tasks on the learning objective would be insignificant. This defeats the purpose of including multiple tasks to prevent overfitting. Thus, we weight the samples such that all tasks have the same impact. If a task Π contributes n states to the training data, then all its states are weighted by $1/n$. Furthermore, we also observe that the classes are highly unbalanced, i.e., there are significantly fewer progress states than non-progress states. We add another weight to each sample. If a task Π contributes p progress states and m non-progress states, then every progress state is additionally weighted with $1/p$ and every non-progress state is weighted with $1/m$. The final weight for the progress states is $1/(n * p)$ and for the non-progress states is $1/(n * m)$. After extracting DNF formulas from the learned decision trees, we simplify them with SymPy (Meurer et al., 2017).

To measure the quality of the trained formulas, we evaluate the F1 score of the formulas on all states from the validation tasks of the same domain (see Table 17.2). We see that training on a single task produces already good formulas. This is important as it validates that our approach classifies most states correctly. If we use more tasks for the training data, the resulting formulas improve only moderately. For Blocksworld we generated enormous feature sets. Using the states from more than a single state space exhausted the memory. The qualitative results are similar for the h^+ heuristic. The major difference is that the quality of the formulas is general better. In one domain, the

Domain	h^+	h^+_{LOpt}	h^+_{L*}
Gripper (17)	137.68	57.03	57.03
Miconic-STRIPS (14)	82.17	51.07	53.15

Table 17.3: Median number of expansions across the commonly solved instance for the Gripper and Miconic-STRIPS domain using GBFS with the perfect delete-relaxed heuristic without tie-breaking (h^+), breaking ties using the handcrafted, perfect progress state formulas (h^+_{LOpt}), and breaking ties using the best trained formula (h^+_{L*}).

formulas are even perfect and in another domain the formulas are almost perfect (see Table C.2).

Secondly, Table 17.2 shows for each domain and the formula trained on a single state space as well for the best formula (with respect to the F1 score) the maximum complexity of a feature in the formula, as well as the number of clauses and literals. All except one formula use at least one feature of the maximum available feature complexity. This indicates that features of higher complexity encode necessary knowledge. Thus, improving the feature generation is a useful venue for future work. Furthermore, we observe that the better formulas are significantly larger than the formulas trained on a single task. As a larger formula also requires more time to evaluate, there is a trade-off between quality of the formula and runtime.

Evaluating Formulas. We learned that our formulas identify progress states very well. Now we show that they improve search. We compare standard GBFS with a heuristic h against GBFS with the same heuristic and break ties in favor of progress states (according to our formulas).

First, we verify that preferring progress state improves the search. We execute GBFS with the h^+ heuristic and use our perfect handcrafted formulas. We use the expansion metric to compare the informedness of two searches on the same task. The fewer states GBFS expands, the better it is informed. Table 17.3 shows the geometric means of the expansions for Gripper and Miconic-STRIPS over all test tasks. Using the perfect progress state information significantly reduces the number of expansions in both domains. The table also shows the expansions when using the best learned formulas for those two domains. The result for Gripper is the same, as we learned the perfect formula. For Miconic-STRIPS the result is almost the same. The learned formula has just minor mistakes. Indeed, breaking ties in favor of progress states is useful.

Next, we evaluate our formulas on all test tasks of all domains. Figure 17.1 compares for every task the expansions of GBFS against GBFS with our tie-breaking. A data point below the diagonal means that our version required fewer expansion on that task than plain GBFS. We immediately observe that our tie-breaking improves the performance in many domains. The GBFS with the FF heuristic expands fewer states than our adapted

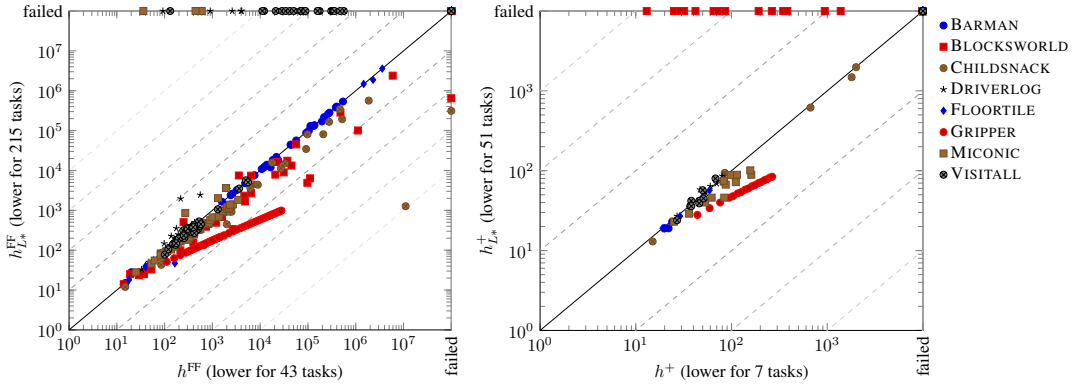


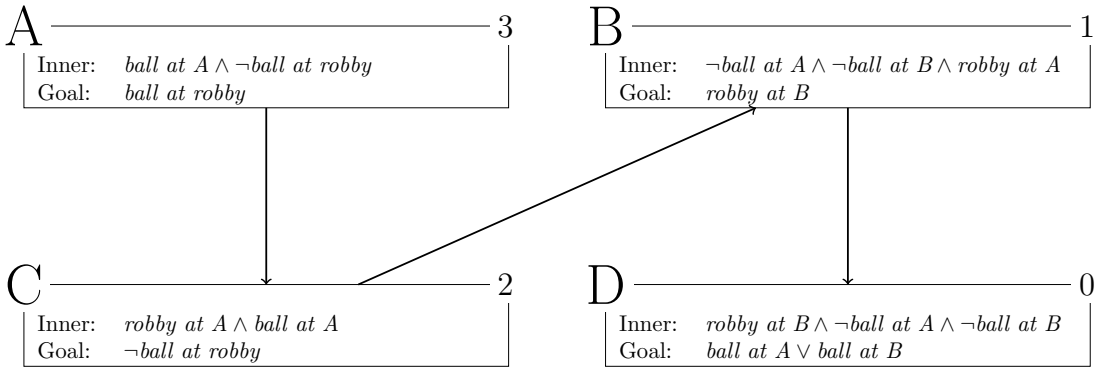
Figure 17.1: Comparison of expansions for GBFS with (left) the FF heuristic and (right) the perfect delete-relaxed heuristic without tie-breaking against GBFS with tie-breaking due to our best trained formula. Every test task is a data point. Data points below the diagonal indicate tasks in which GBFS with our tie-breaking expands fewer states.

version in 43 tasks. On the other hand, our adapted version expands fewer states in 215 tasks. GBFS with h^+ expands fewer states in 7 tasks. Our version expands fewer states in 51 tasks. We conclude that our formulas identify sufficiently well progress states and that using this information as a tie-breaker is useful.

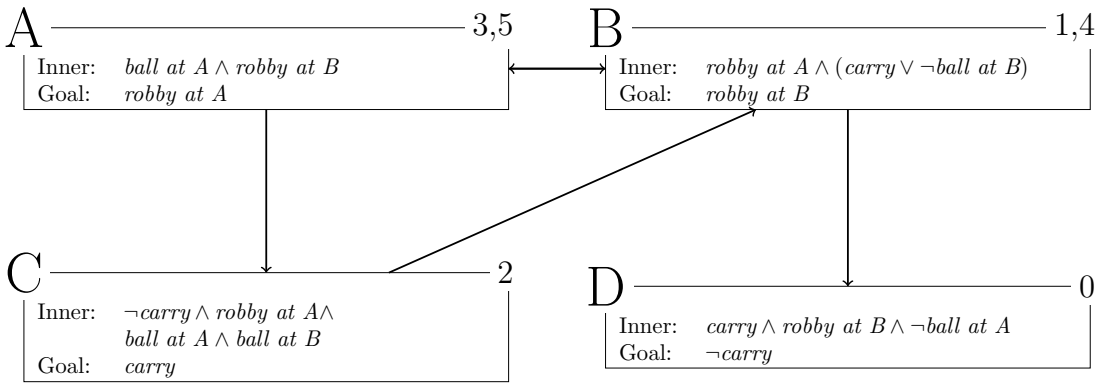
Focusing on the informedness of the searches hides the aspect of time. Although, we like an informed search, we prefer a search which quickly finds a solution. Table 17.4 compares the runtime of the plain GBFSs against their counterparts with our tie-breaking. For a heuristic which is slow to evaluate like h^+, our tie-breaking is beneficial. For a fast heuristic like h^{FF}, our formulas are too slow to evaluate. Evaluating the formulas costs more time than the reduced number of expansions saves. In the end, there are two steps for the future: Making the formulas more compact and speeding up the evaluation. The latter can be achieved by encoding our formulas as additional knowledge in the form of axioms using our conversion from description logic concept or role to *PDDL* like Steinmetz et al. (2022b).

17.2. Constructing Generalized Bench Transition Systems

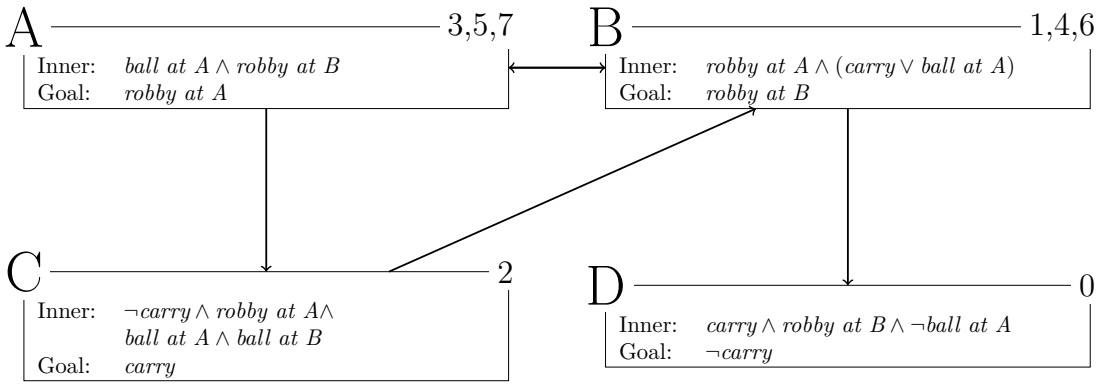
We now present preliminary results about constructing generalized bench transition systems (GBTS). As heuristic we use h^+ and as domain we use GripperOne. To support more domains, we currently improve the expressivity of our feature language. The bench walker guides the search in GripperOne well. Further domains have to be supported for an evaluation.



(a) 1 ball



(b) 2 balls



(c) 3 balls

Figure 17.2: Automatically constructed generalized bench transition systems for GripperOne tasks with 1, 2, and 3 balls. Benches with the same label (top left) share the same meaning. The numbers on the top right indicate the levels of the benches merged into this generalized bench. Following the numbers from maximal to minimal value represents the flow between the generalized benches during search. *Inner* is the inner states formula ϕ_{inner}. *Goal* is the inner goal formula $\phi_{inner\ goal}$.

Domain	h^{FF}	h^{FF}_{L*}	h^{+}	h^{+}_{L*}
Barman	3.4	19.1	161	154
Blocksworld	0.3	0.5	–	–
ChildSnack	3.5	6.8	88	87
Driverlog	0.1	3.2	244	252
FloorTile	0.4	0.9	11	11
Gripper	1.1	0.6	301	266
Miconic-STRIPS	0.2	1.5	103	95
VisitAll	0.0	10.7	305	303

Table 17.4: Geometric mean of the runtimes over the commonly solved test tasks of GBFS with the FF heuristic (resp. perfect delete-relaxed heuristic) without tie-breaking and with tie-breaking due to our best trained formula.

Generalized Bench Transition Systems. We generate three tasks Π_1, Π_2, and Π_3. Π_i has i balls. For Π_i, we use our approach to construct a GBTS $\mathfrak{G}_i$. We limit the generated features to a complexity of at most 10. Figure 17.2 shows those GBTS. For simplicity, the figure shows the formula to identify the inner states instead of the more complicated formula, which identifies all member states.

Our first observation is that $\mathfrak{G}_2$ and $\mathfrak{G}_3$ are isomorphic. $\mathfrak{G}_1$ is similar, but misses two edges. As Π_1 has only a single ball, there is no need for the robot to return to room A and to pick up a second ball. Without this repetition, the missing edges cannot be learned. Thus, $\mathfrak{G}_1$ does not generalize for GripperOne. We conclude that the task from which we construct the GBTS has to exhibit all relevant features of the domain.

Our second observation is that all three GBTS have four benches and all benches represent a similar idea. *Bench A* represents that the robot is in room B and still needs to pick up balls in room A. Thus, its goal is to move to room A. *Bench B* represents that the robot is in room A and still needs to deliver balls to room B. If it already carries a ball, it moves to B. If not it would pick up a ball and then move to B. *Bench C* can be understood as a specialization of bench B. The robot is in room A and has to pick up the last ball. *Bench D* represents the final step of any plan. The robot carries the last ball and is in room B. Now it only needs to drop it. Nevertheless, we also see that $\mathfrak{G}_1$ is expressing some concepts differently than $\mathfrak{G}_2$ and $\mathfrak{G}_3$. This is because it contains only a single ball. If there is only a single ball, then the idea that the robot carries a ball can also be phrased as there is no ball on the floor of room A and no ball on the floor of room B. This idea does not generalize to tasks with multiple balls. Similarly, $\mathfrak{G}_2$ expresses some things differently than $\mathfrak{G}_3$. In the bench B, it requires the robot to carry a ball or that there is no ball on the floor of room B. As this task has only two balls and the robot can carry at most one ball, having no ball on the floor of room B is the same as having at least one ball in room A. But this does not generalize to tasks with more than 2 balls.

The most important part of the $\mathfrak{G}_2$ and $\mathfrak{G}_3$ is the loop between the benches A and B. In every iteration of this loop the robot moves one ball from room A to room B. How is this happening, if the goal of the benches is to move *just* the robot? Keep in mind that the benches are constructed for h^+ using the original goal of Π_i. For the bench A, assume that the robot is in room B and there are still balls in room A. If the robot carries a ball, then dropping the ball reduces the value of h^+. On the other hand, moving to room A while carrying the ball does not reduce the value of h^+. Thus, GBFS will first drop any carried ball. Afterwards, it will move the robot to room A. For the bench B, a similar reasoning applies. If the robot is in room A and carries a ball, then the robot should move to room B. This is fine. If it does not yet carry a ball, then picking a ball up decreases the value of h^+. Moving to another room decreases the heuristic value only if we carry the last ball.

18. Summary and Future Work

Many features of the state space topology are known only a posteriori. We presented first approaches to learn generalizing knowledge over properties of the state space topology. We can classify whether a state is a progress state during search! The formulas we learn for h^+ have a high quality. The formulas for h^{FF} are still good. Our formulas encode useful knowledge, which decreases the number of expansions during search. Furthermore, we presented our ongoing work where we learn the generalized bench transition system for a whole domain from the bench transition system of a single task. We show how the generalization works and how the generalized bench transition system can improve search. Furthermore, we show that our method can successfully construct generalized bench transition systems. But, there is still much to research.

Our progress state formulas are too slow to evaluate. Thus, they are not yet useful in practice. Inspecting more closely the trade-off between quality of the formula and size of the formula could be useful. This knowledge can also be applied to other formulas, which we learn or want to learn. Instead of identifying progress states during search, we could identify crater states. A crater is a set of non-goal states with incorrectly low heuristic values. Once GBFS enters a crater, it has to expand all states within the crater before it exits the crater. If we could prevent GBFS from entering craters, this improves the search performance drastically. Pruning states during search is risky, as an imperfect formula renders the search incomplete.

Bottleneck states are another class of interesting states. A state s is a bottleneck state if it is necessary in all plans. Thus, it is a state landmark. We could learn formulas to identify bottleneck states. Then we could use these formulas (or their conversion to $PDDL$) to construct additional states or action landmarks or subgoals. If our formulas are imperfect, then, we might generate some incorrect landmarks, but this does not render the search incomplete and as long as most landmarks are correct it could improve the search. Furthermore, adding landmarks could help to identify craters and thus, GBFS might avoid them.

The main difficulty of this research is that our feature language has to be expressive enough and that we can generate the necessary features within reasonable computational limits. Currently, our description logic formulas can reason about a current state and about the true goal of the given task. In many domains, we observe that this is insufficient for the inner and outer goal formulas. The goal formulas describe progress. For this purpose, they have to additionally consider the initial state of the subsearch. To reason about the original goal, we include the predicates of the goal in the genera-

tion of the description logic features. For all goal predicates, we suffix their predicate symbols with an annotation for the goal (Section 5.2). We can use the same mechanism to include knowledge about the initial state of subsearch. Every inner state can be the start of a subsearch. Thus, we have to construct an individual set of description logic features per bench. This is more expensive, but feasible on small tasks of a domain.

For the example of the GripperOne domain, we showed that we require a task with at least 3 balls to learn the GBTS for the perfect delete-relaxed heuristic. Another open question is how to pick the right task for constructing the generalized bench transition system? If this is too difficult, we can sample benches from multiple tasks, combine them, and to hope that they cover everything important.

We observed that the generalized bench transition system of the perfect heuristic requires exactly one action to transform an inner state of a generalized bench to an exit state. If we can construct a general description of the action used in every bench, then the GBTS of the perfect heuristic could be used as policy. This could have great potential.

Part IV.

Conclusion

19. Conclusion

We used machine learning to improve the state of the art in planning in three directions.

In Part I, we asked ourselves: *Can we learn powerful and competitive heuristics for reset problems?* We presented one approach based on progression random walks and a teacher planner and three approaches based on regression random walks, which use the current state of the neural network itself as the teacher. All our heuristics, as well as STRIPS-HGN, exhibit highly complementary qualities on the test domains. It is a priori unclear, which method to use for a given domain. LAMA is still the dominant planner. For the second time ever, learning-based planners solve more tasks in a domain than LAMA. These are our three regression random walk planners on Storage.

Not only the learning-based heuristics, but also model-based planning algorithms have complementary strengths. Again, it is not always obvious *when to execute which planner.* Portfolios learn to combine the strengths of multiple planning techniques. In Part II, we construct new online portfolios based on CNNs which receive the task to solve with few modifications and GNNs which receive the task to solve directly as input. Our online portfolios are not biased by human picked input features. Both approaches exceed the state of the art. Enabling the online portfolios to reconsider further boosts their coverages. Finally, we showed that explainable online portfolios are possible and comparable to the previous state of the art and how we can build trust in their predictions.

Even if we select the best planner for a task, it might not be good enough to solve the task within a reasonable time. Thus, we have to improve them. *Can we inspect the behavior of a planner on a simple task, identify mistakes, and learn to avoid them?* In the final Part III, we identified progress states for a heuristic search and learned formulas, which describe them. This knowledge reduces the required expansions during search. In our ongoing work, we generalized the whole bench transition system of a single task to all tasks of the same domain. This can be used to solve a hard task by solving a sequence of simpler tasks.

Our journey started during the hype around neural networks for planning. In each project, we asked ourselves, *do we need the power of neural networks? Can we pay the explanatory price?* The further we progressed, the fewer neural networks we used. For learning progress states, we initially used neural networks, but switched to decision trees, because they provided better opportunities for understanding and debugging the learned models. In the end, the formulas we can extract from decision trees enabled us to generalize the bench transition system of a task. This would be impossible with

neural networks. Neural networks are mighty tools that should be wielded wisely.

There is still a lot of open research for improving classical planning with machine learning. In each of our direction, we either have an ongoing project or planned one. Numerous further directions exist.

Today, most search algorithms work on a grounded representation of the $PDDL$ task, but this grounding can cause an exponential blow up in the size of the task representation. As our algorithms become better and better, we solve harder and harder tasks. Today, we observe for numerous hard tasks that we are unable to ground them. This renders our highly engineered algorithms useless. Solving this issue is a key challenge. A first solution to this problem improves grounding with machine learning. Gnad et al. (2019) present an initial approach, which learns on small tasks to identify important ground actions. On hard tasks, it generates only important ground actions. A second solution is lifted search (Corrêa et al., 2020), i.e., the search algorithm grounds neither the action schemas nor the goal. As many traditional techniques depend on the grounded actions, they do not work anymore. Much research is done on adapting them to lifted search (Corrêa et al., 2021, 2022; Corrêa and Seipp, 2022; Wichlacz, Höller, and Hoffmann, 2022). When we train machine learning techniques for planning, we should consider supporting lifted search. Our neural network heuristics presented in Part I require solely the state as input, thus, we can use them in grounded *and* lifted search. An alternative to grounded search, which requires a grounded action model, are policies, i.e., functions which predict for a state the next action. Much research for learning policies started (Toyer et al., 2018; Drexler, Seipp, and Geffner, 2022; Ståhlberg, Bonet, and Geffner, 2022). As grounded search is still the dominant technique for solving planning problems, we still need to come up with better policies. A final solution splits the hard tasks into a sequence of simpler tasks, as we suggest in our ongoing work of Part III.

A second key challenge is modeling planning tasks. Today, most tasks are modeled by planning experts. To make planning widely applicable, we have to enable non-experts to model their problems by themselves. Asai and Fukunaga (2018) presented a neural network, which learns a planning task from observation. This network does not output the planning task. Thus, we cannot verify the learned model. In succeeding work, they construct a $PDDL$ task from observations (Asai and Muise, 2020). This $PDDL$ task is hard to understand, but an important step to enable the layman to construct and verify their planning task. Over time, the goal of a task can change. To facilitate formalizing goals, Sharma et al. (2022) learn to formalize goals from natural language using observations. These are first steps for automatically constructing tasks.

We recapitulate that there are key challenges in planning, and for each key challenge, researchers came up with machine learning based solutions. But, all solutions are still in its infancy. The future will be exciting.

Appendix A.

Heuristics

A.1. Relation of Canonical Abstractions and Description Logic

Karia and Srivastava (2021) used *canonical abstractions* (Sagiv, Reps, and Wilhelm, 1999) to formalize features for a domain which can be evaluated on *any* task of that domain. They use these features to train a neural network as heuristic and as policy. We show that we can express their fragment of canonical abstractions in description logic.

Let Π be a $PDDL$ task with objects $\mathcal{O}$, predicate symbols $\mathcal{P}$, and states $\mathcal{S}$. Let P_k be the set of all predicate symbols of Π with arity k. Remember from the description logic background that $C_{p,i}$ with $p \in \mathcal{P}$ and $0 < i <= arity(p)$ is an atomic description logic concept, $R_{p,i,j}$ with $p \in \mathcal{P}$ and $0 < i, j <= arity(p)$ is an atomic description logic role, $\top_C = \mathcal{O}$ is the top concept, and $\top_R = \mathcal{O} \times \mathcal{O}$ is the top role.

A *canonical abstraction role* $\mathcal{R}$ is a set of unary predicate symbols, i.e., $\mathcal{R} \subseteq \mathcal{P}_1$. The denotation of a canonical abstraction role $\mathcal{R}$ on a state $s \in \mathcal{S}$ is the set of objects which fulfill this role, i.e., $\mathcal{R}^s = \{o \mid o \in \mathcal{O}, \forall p \in \mathcal{R} : p(o) \in s, \forall p \in \mathcal{P}_1 \setminus \mathcal{R} : p(o) \notin s\}$. And in the other direction, the canonical abstraction role of an object $o \in \mathcal{O}$ in state $s \in \mathcal{S}$ is the set of its unary predicate symbols, i.e., $role(o, s) = \{p \mid p \in \mathcal{P}_1, p(o) \in s\}$.

Let $p \in \mathcal{P}_k$ be a k-ary predicate symbol. Let $\mathfrak{R}_{p,i}$ with $i \in \{1 \dots k\}$ be the set of all canonical abstraction roles which occur in any state at the i-th argument of p, i.e., $\mathfrak{R}_{p,i} = \{role(o_i, s) \mid o_1, \dots, o_k \in \mathcal{O}, s \in \mathcal{S}, p(o_1, \dots, o_k) \in s\}$. The predicate symbol p implies the set of *abstract predicates* $\{\bar{p}(r_1, \dots, r_k) \mid r_1, \dots, r_k \in \mathfrak{R}_{p,1} \times \dots \times \mathfrak{R}_{p,k}\}$. The evaluation of an abstract predicate $\bar{p}_(r_1, \dots, r_k)$ in a state $s \in \mathcal{S}$ is defined as

$$\bar{p}_(r_1, \dots, r_k)(s) = \begin{cases} 1 & \text{if } \forall o_1, \dots, o_k \in r_1(s) \times \dots r_k(s) : p(o_1, \dots, o_k) \in s \\ 0 & \text{if } \forall o_1, \dots, o_k \in r_1(s) \times \dots r_k(s) : p(o_1, \dots, o_k) \notin s \\ 0.5 & \text{otherwise.} \end{cases}$$

In colloquial terms, an abstract predicate describes a set of facts and if all these facts are in a state, it evaluates to one. If none of these facts is in a state, it evaluates to zero. And otherwise it evaluates to a half.

In theory canonical abstraction work with predicate symbols of any arity, but Karia and Srivastava (2021) limit themselves to unary and binary predicate symbols. They compile higher arity predicate symbols to multiple binary predicate symbols. Thus, we restrict ourselves to unary and binary predicates. Let $\mathcal{R} \subseteq \mathcal{P}_1$ be a canonical abstraction role. We can model the $\mathcal{R}$ as description logic concept such that the denotations are the same:

$$\mathcal{R} = \bigsqcap_{p \in \mathcal{R}} C_{p_i,1} \sqcap \bigsqcap_{p \in \mathcal{P}_1 \mathcal{R}} \neg C_{p_i,1}$$

Let $p \in \mathcal{P}_1$ be a unary predicate symbol and $r \subseteq \mathcal{P}_1$ be a role. Let $\bar{p}(r)$ be an abstract predicate. The denotation of the concept $C_{in} = C_{p,1} \sqcap r$ describes all objects o for which $p(o) \in s$ holds and which fit the role r. The denotation of the concept $C_{out} = (\top_C \setminus C_{p,1}) \sqcap r$ describes all objects o for which $p(o) \notin s$ holds and which fit the role r. We can now evaluate $\bar{p}(r)$ using only description logic:

$$\bar{p}(r)(s) = \begin{cases} 1 & \text{if } |C^s_{in}| > 0 \wedge |C^s_{out}| = 0 \\ 0 & \text{if } |C^s_{in}| = 0 \wedge |C^s_{out}| > 0 \\ 0.5 & \text{if } |C^s_{in}| > 0 \wedge |C^s_{out}| > 0 \end{cases}$$

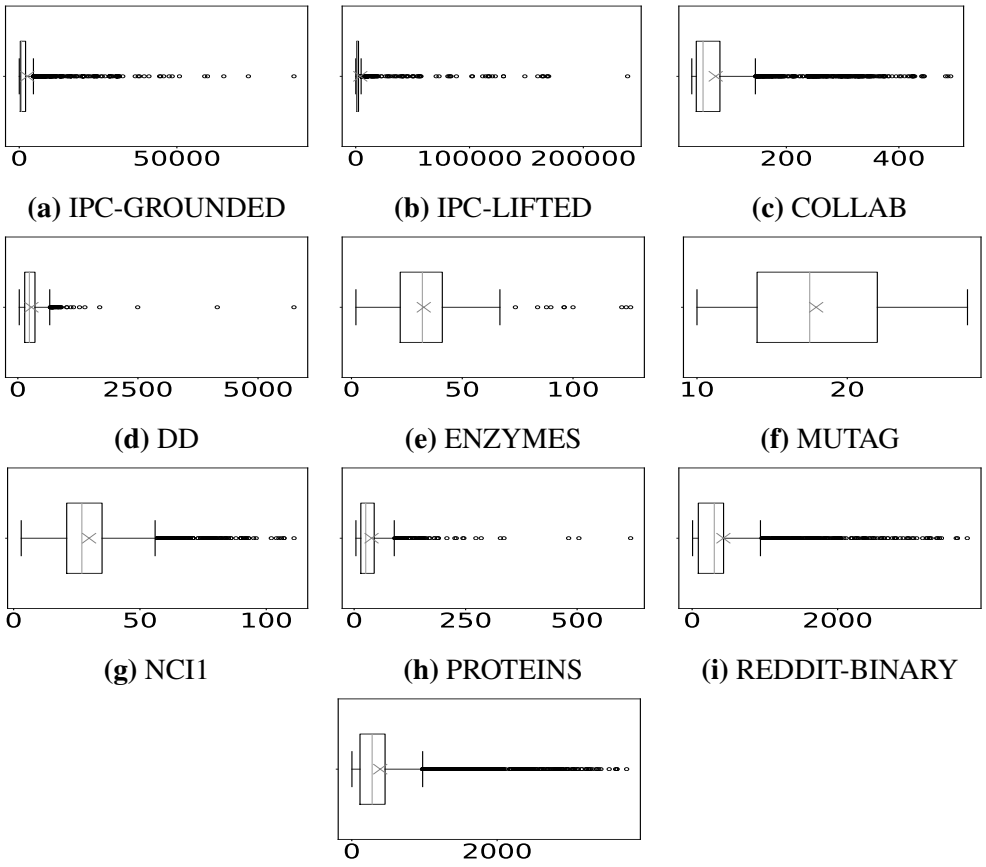
Let $p \in \mathcal{P}_2$ be a binary predicate symbol and $r_1, r_2 \subseteq \mathcal{P}_1$ be two roles. Let $\bar{p}(r_1, r_2)$ be an abstract predicate. The denotation of the role $R_{in} = ((R_{p,1,2|r_2})^{-1})_{|r_1}$ describes all tuples of objects o_1, o_2 for which $p(o_1, o_2) \in s$ holds and which fit the roles r_1, r_2. The denotation of the role $R_{out} = ((\top_R \setminus R_{p,1,2})_{|r_2})^{-1})_{|r_1}$ describes tuples of objects o_1, o_2 for which $p(o_1, o_2) \notin s$ holds and which fit the roles r_1, r_2. We can now evaluate $\bar{p}(r_1, r_2)$ using only description logic:

$$\bar{p}(r)(s) = \begin{cases} 1 & \text{if } |R^s_{in}| > 0 \wedge |R^s_{out}| = 0 \\ 0 & \text{if } |R^s_{in}| = 0 \wedge |R^s_{out}| > 0 \\ 0.5 & \text{if } |R^s_{in}| > 0 \wedge |R^s_{out}| > 0 \end{cases}$$

Appendix B.

Portfolios

B.1. Statistics About the ASG and PDG for Planning



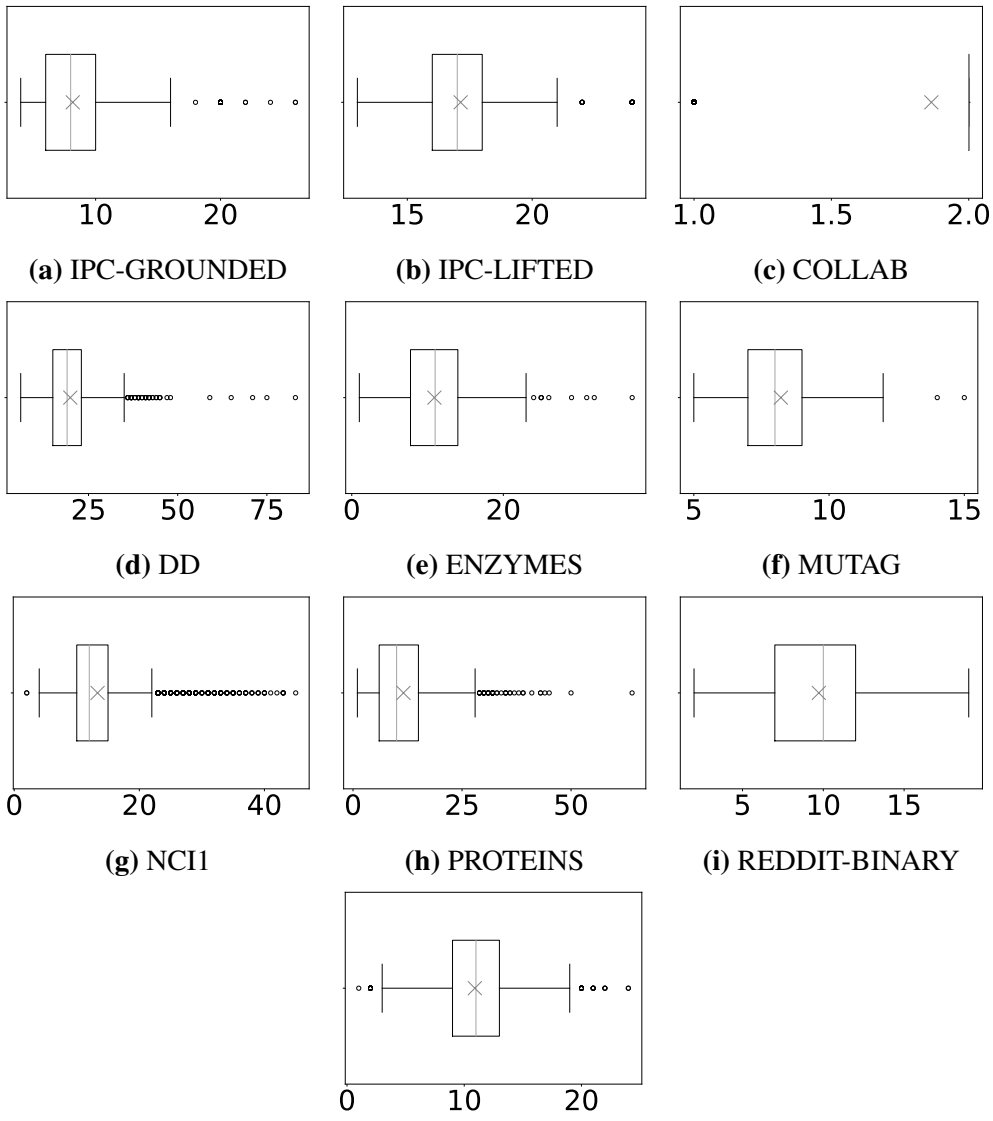
(a) IPC-GROUNDED **(b)** IPC-LIFTED **(c)** COLLAB

(d) DD **(e)** ENZYMES **(f)** MUTAG

(g) NCI1 **(h)** PROTEINS **(i)** REDDIT-BINARY

(j) REDDIT-MULTI-12K

Figure B.1: Comparison of graph sizes between our grounded and lifted graphs and other graph data sets (Kersting et al., 2016). The gray line is the median. The box extends from the 25 to the 75 percentile. The whiskers extend to the 5 and the 95 percentile.



(a) IPC-GROUNDED **(b)** IPC-LIFTED **(c)** COLLAB

(d) DD **(e)** ENZYMES **(f)** MUTAG

(g) NCI1 **(h)** PROTEINS **(i)** REDDIT-BINARY

(j) REDDIT-MULTI-12K

Figure B.2: Comparison of graph diameter between our grounded and lifted graphs and other graph data sets (Kersting et al., 2016). The gray line is the median graph sizes. The box extends from the 25 to the 75 percentile. The whiskers extend to the 5 and the 95 percentile.

B.2. Results of Hyper-Parameter Optimization for Image-Based Portfolios

	time						normalized					
	domain-preserving split			random split			domain-preserving split			random split		
	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$
β_1	0.975	0.99	0.98	0.96	0.96	0.89	0.89	0.6	0.67	0.99	0.77	0.91
β_2	0.99	0.99	0.99	0.99	0.99	0.999	0.999	0.99	0.995	0.99	0.99	0.99
ϵ	9.7e-9	9.9e-9	9.9e-9	4.6e-9	4.6e-9	5.1e-9	7.8e-9	5.85e-9	9.8e-9	9.99e-9	7.7e-9	9.9e-9
learning rate	7e-4	5e-4	6e-4	0.0049	0.0049	0.0041	0.0043	0.0049	0.0036	0.0015	5e-4	5e-4
batch size	124	113	53	70	70	84	85	95	89	63	58	65
conv. filter size	2	2	2	3	3	3	3	6	5	3	5	2
dropout rate	0.49	0.16	0.49	0.34	0.34	0.47	0.48	0.49	0.38	0.16	0.16	0.48
poolfilter size	1	2	1	2	2	4	3	1	2	2	5	2

	discrete						binary					
	domain-preserving split			random split			domain-preserving split			random split		
	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$	$\mathcal{C}_D$	$\mathcal{C}_A$	$\mathcal{C}_C$
β_1	0.99	0.93	0.99	0.99	0.98	0.98						
β_2	0.99	0.998	0.9995	0.99	0.99	0.9998						
ϵ	9.98e-9	4.5e-10	9.85e-9	9.85e-9	9.9e-9	2.5e-10						
decay							0.089	0.089	0.055	4.3e-5	3.2e-5	1.5e-4
momentum							0.73	0.73	0.25	0.95	0.94	0.95
nesterov							1	1	1	1	1	1
learning rate	5e-4	8e-4	5e-4	6e-4	5e-4	5e-4	0.085	0.085	0.07	0.0058	0.0065	0.005
batch size	94	119	114	123	77	101	85	85	89	101	53	56
conv. filter size	6	5	2	2	6	6	3	3	5	6	2	2
dropout rate	0.10	0.43	0.11	0.50	0.49	0.50	0.48	0.48	0.39	0.49	0.49	0.5
poolfilter size	1	4	1	1	4	4	3	3	2	1	1	4

Table B.1: Final set of hyperparameters for the lifted CNN based online portfolios.

Appendix C.

Topology

	Max. Complexity					# Features	
Domain	1	2	3	4	5	Min	Max
Barman	8	7	7	7	7	2726	5896
Blocksworld	10	10	9	9	9	10648	26081
ChildSnack	11	12	10	9	8	621	2611
Driverlog	9	8	8	8	8	1204	1973
FloorTile	8	8	7	7	7	1644	3907
Gripper	14	12	12	9	9	441	5950
Miconic-STRIPS	9	7	7	7	7	218	825
VisitAll	12	12	11	11	10	658	2132

Table C.1: For each domain and data set when using the perfect delete-relaxed heuristic, the maximum complexity of a generated features and the maximum and minimum number of features generated across the five training sets.

	Mean Val. F1					First DNF			Best DNF		
Domain	1	2	3	4	5	$\mathcal{K}(F)$	Clauses	Literals	$\mathcal{K}(F)$	Clauses	Literals
Barman	93	100	100	100	100	8	9	51	7	46	361
ChildSnack	54	55	73	72	71	12	4	8	10	8	34
Driverlog	95	95	90	94	95	9	27	225	9	27	225
FloorTile	84	85	88	88	90	8	5	39	7	49	659
Gripper	100	100	100	100	100	14	2	2	14	2	2
Miconic-STRIPS	99	95	99	99	99	8	1	1	8	1	1
VisitAll	85	84	79	81	81	12	777	12056	12	777	12056

Table C.2: On the perfect delete-relaxed heuristic progress state data. (Left) For each domain and formula the F1 score on the validation data. (Middle) For the formula trained on a single state space and (Right) for the best learned formula with respect to the F1 validation score the maximum complexity of a used feature as well as the number of clauses and literals in the formula.

Bibliography

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

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1: 356–363.

Alcázar, V.; and Torralba, Á. 2015. A Reminder about the Importance of Computing and Exploiting Invariants in Planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2–6. AAAI Press.

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A Stubborn Set Algorithm for Optimal Planning. In De Raedt, L.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P., eds., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, 891–892. IOS Press.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artificial Intelligence*, 175: 2075–2098.

Asai, M.; and Fukunaga, A. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6094–6101. AAAI Press.

Asai, M.; and Muise, C. 2020. Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (to STRIPS). In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020)*, 2676–2682. IJCAI.

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4): 625–655.

Barceló, P.; Kostylev, E.; Monet, M.; Pérez, J.; Reutter, J.; and Silva, J.-P. 2020. The Logical Expressiveness of Graph Neural Networks. In *Proceedings of the Eighth International Conference on Learning Representations (ICLR 2020)*. OpenReview.net.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bhatia, A.; Svegliato, J.; Nashed, S. B.; and Zilberstein, S. 2022. Tuning the Hyperparameters of Anytime Planning: A Metareasoning Approach with Deep Reinforcement Learning. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 556–564. AAAI Press.

Biedenkapp, A.; Speck, D.; Sievers, S.; Hutter, F.; Lindauer, M.; and Seipp, J. 2022. Learning Domain-Independent Policies for Open List Selection. In *ICAPS 2022 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-state Controllers Using Classical Planners. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 34–41. AAAI Press.

Breiman, L. 1996. Technical Note: Some Properties of Splitting Criteria. *Machine Learning*, 24(1): 41–47.

Breiman, L. 2001. Random Forests. *Machine Learning*, 45(1): 5–32.

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees*. Wadsworth.

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*.

Büchner, C.; Ferber, P.; Seipp, J.; and Helmert, M. 2022. A Comparison of Abstraction Heuristics for Rubik's Cube. In *ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2013. Learning predictive models to configure planning portfolios. In *ICAPS 2013 Workshop on Planning and Learning*, 14–22.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2014. IBaCoP and IBaCoPB Planner. In *Eighth International Planning Competition (IPC-8): Planner Abstracts*, 35–38.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2016. The IBaCoP Planning System: Instance-Based Configured Portfolios. *Journal of Artificial Intelligence Research*, 56: 657–691.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2018. IBaCoP-2018 and IBaCoP2-2018. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 9–10.

Cheng, J.; Wang, Z.; and Pollastri, G. 2008. A neural network approach to ordinal regression. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 1279–1284.

Cho, K.; van Merriënboer, B.; Bahdanau, D.; and Bengio, Y. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *EMNLP 2014 Workshop on Semantics and Structure in Statistical Translation*, 103–111.

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

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 94–102. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 80–89. AAAI Press.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2022. The FF Heuristic for Lifted Classical Planning. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9716–9723. AAAI Press.

Corrêa, A. B.; and Seipp, J. 2022. Best-First Width Search for Lifted Classical Planning. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 11–15. AAAI Press.

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4): 303–314.

de Rijke, M. 2000. A Note on Graded Modal Logic. *Studia Logica*, 64(2): 271–283.

Dechter, R.; and Pearl, J. 1985. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, 32(3): 505–536.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In Lee, D.; Sugiyama, M.; Luxburg, U.; Guyon, I.; and Garnett, R., eds., *Proceedings of the Thirtieth Conference on Neural Information Processing Systems (NIPS 2016)*, 3837–3845. Curran Associates, Inc.

Diaz, G. I.; Fokoue-Nkoutche, A.; Nannicini, G.; and Samulowitz, H. 2017. An effective algorithm for hyperparameter optimization of neural networks. *IBM Journal of Research and Development*, 61(4).

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A New Systematic Approach to Partial Delete Relaxation. *Artificial Intelligence*, 221: 73–114.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 343–347. AAAI Press.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2015. Symmetry Breaking in Deterministic Planning as Forward Search: Orbit Space Search Algorithm. Technical Report IS/IE-2015-03, Technion.

Doran, J. E.; and Michie, D. 1966. Experiments with the Graph Traverser program. *Proceedings of the Royal Society A*, 294: 235–259.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed Model Checking with Distance-Preserving Abstractions. In Valmari, A., ed., *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, 19–34. Springer-Verlag.

Drexler, D.; Francès, G.; and Seipp, J. 2022. Description Logics State Features for Planning (DLPlan). `https://doi.org/10.5281/zenodo.5826139`.

Drexler, D.; Seipp, J.; and Geffner, H. 2022. Learning Sketches for Decomposing Planning Problems into Subproblems of Bounded Width. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 62–70. AAAI Press.

Drucker, H.; Burges, C. J. C.; Kaufman, L.; Smola, A.; and Vapnik, V. 1996. Support Vector Regression Machines. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, 155–161.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Edelkamp, S. 2006. Automated Creation of Pattern Database Search Heuristics. In Edelkamp, S.; and Lomuscio, A., eds., *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, Decidability and Undecidability Results for Domain-Independent Planning. *Artificial Intelligence*, 76(1–2): 75–88.

Fan, G.; Müller, M.; and Holte, R. 2014. Non-Linear Merging Strategies for Merge-and-Shrink Based on Variable Interactions. In Edelkamp, S.; and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 53–61. AAAI Press.

Fawcett, C.; Vallati, M.; Hutter, F.; Hoffmann, J.; Hoos, H.; and Leyton-Brown, K. 2014. Improved Features for Runtime Prediction of Domain-Independent Planners. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 355–359. AAAI Press.

Ferber, P. 2020. Simplified Planner Selection. In *ICAPS 2020 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 102–110.

Ferber, P.; Cohen, L.; Seipp, J.; and Keller, T. 2022a. Learning and Exploiting Progress States in Greedy Best-First Search. In De Raedt, L., ed., *Proceedings of the 31th International Joint Conference on Artificial Intelligence (IJCAI 2022)*, 4740–4746. IJCAI.

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022b. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 583–587. AAAI Press.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In De Giacomo, G., ed., *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, 2346–2353. IOS Press.

Ferber, P.; Ma, T.; Huo, S.; Chen, J.; and Katz, M. 2019. IPC: A Benchmark Data Set for Learning with Graph-Structured Data. In *Proceedings of the ICML-2019 Workshop on Learning and Reasoning with Graph-Structured Representations*.

Ferber, P.; and Seipp, J. 2022. Explainable Planner Selection for Classical Planning. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9741–9749. AAAI Press.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2: 189–208.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning General Planning Policies from Small Examples Without Supervision. In Leyton-Brown, K.; and Mausam, eds., *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*, 11801–11808. AAAI Press.

Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The Complementary2 Planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 32–36.

Franco, S.; Lelis, L. H. S.; Barley, M.; Edelkamp, S.; Martines, M.; and Moraru, I. 2018. The Complementary1 Planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 28–31.

Galton, F. 1886. Regression Towards Mediocrity in Hereditary Stature. *Journal of the Anthropological Institute of Great Britain and Ireland*, 15: 246–263.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 631–636. AAAI Press.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural Message Passing for Quantum Chemistry. In Precup, D.; and Teh, Y. W., eds., *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, 1263–1272. JMLR.org.

Gnad, D.; Shleyfman, A.; and Hoffmann, J. 2018. DecStar – STAR-topology DE-Coupled Search at its best. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 42–46.

Gnad, D.; Torralba, Á.; Domínguez, M. A.; Areces, C.; and Bustos, F. 2019. Learning How to Ground a Plan – Partial Grounding in Classical Planning. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7602–7609. AAAI Press.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. MIT Press.

Grädel, E.; Otto, M.; and Rosen, E. 1997. Two-Variable Logic with Counting is Decidable. In *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS 1997)*, 306–317.

Hamilton, W. 2020. *Graph Representation Learning*, volume 14 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool.

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive Representation Learning on Large Graphs. In Guyon, I.; von Luxburg, U.; Bengio, S.; Wallach, H. M.; Fergus, R.; Vishwanathan, S. V. N.; and Garnett, R., eds., *Proceedings of the Thirty-first Conference on Neural Information Processing Systems (NIPS 2017)*, 1024–1034. Curran Associates, Inc.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Haslum, P. 2011. Computing Genome Edit Distances using Domain-Independent Planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, 770–778. IEEE Computer Society.

Heller, D.; Ferber, P.; Bitterwolf, J.; Hein, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions: Taking Confidence into Account. In *Proceedings of the 15th Annual Symposium on Combinatorial Search (SoCS 2022)*, 223–228. AAAI Press.

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

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Helmert, M.; and Röger, G. 2008. How Good is Almost Perfect? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 944–949. AAAI Press.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 Planner Abstracts*, 38–45.

Heusner, M. 2019. *Search Behavior of Greedy Best-First Search*. Ph.D. thesis, University of Basel.

Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the Search Behaviour of Greedy Best-First Search. In Fukunaga, A.; and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 47–55. AAAI Press.

Heusner, M.; Keller, T.; and Helmert, M. 2018. Best-Case and Worst-Case Behavior of Greedy Best-First Search. In Lang, J., ed., *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 1463–1470. IJCAI.

Hoffmann, J. 2005. Where 'Ignoring Delete Lists' Works: Local Search Topology in Planning Benchmarks. *Journal of Artificial Intelligence Research*, 24: 685–758.

Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h^+. *Journal of Artificial Intelligence Research*, 41: 155–229.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Holte, R. C. 2010. Common Misconceptions Concerning Heuristic Search. In Felner, A.; and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 46–51. AAAI Press.

Imai, T.; and Fukunaga, A. 2015. On a Practical, Integer-Linear Programming Model for Delete-Free Tasks and its Use as a Heuristic for Cost-Optimal Planning. *Journal of Artificial Intelligence Research*, 54: 631–677.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 422–430. AAAI Press.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In Leyton-Brown, K.; and Mausam, eds., *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*, 8064–8073. AAAI Press.

Katz, M.; Hoffmann, J.; and Domshlak, C. 2013. Who Said We Need to Relax *All* Variables? In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 126–134. AAAI Press.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 57–64.

Kersting, K.; Kriege, N. M.; Morris, C.; Mutzel, P.; and Neumann, M. 2016. Benchmark Data Sets for Graph Kernels. `http://graphkernels.cs.tu-dortmund.de`.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the Third International Conference on Learning Representations (ICLR 2015)*.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the Fifth International Conference on Learning Representations (ICLR 2017)*. OpenReview.net.

Knuth, D. E.; and Moore, R. W. 1975. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4): 293–326.

Köhler, J. 1999. Handling of Conditional Effects and Negative Goals in IPP. Technical Report 128, University of Freiburg, Department of Computer Science.

Köppel, M.; Segner, A.; Wagener, M.; Pensel, L.; Karwath, A.; and Kramer, S. 2019. Pairwise Learning to Rank by Neural Networks Revisited: Reconstruction, Theoretical Analysis and Practical Performance. In Brefeld, U.; Fromont, É.; Hotho, A.; Knobbe, A. J.; Maathuis, M. H.; and Robardet, C., eds., *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2019)*, volume 11907 of *Lecture Notes in Computer Science*, 237–252. Springer-Verlag.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, 700–705. AAAI Press.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening A*. *Artificial Intelligence*, 129: 199–218.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature*, 521(7553): 436–444.

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11): 2278–2324.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of the Fourth International Conference on Learning Representations (ICLR 2016)*.

Loreggia, A.; Malitsky, Y.; Samulowitz, H.; and Saraswat, V. A. 2016. Deep Learning for Algorithm Portfolios. In Schuurmans, D.; and Wellman, M., eds., *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, 1280–1286. AAAI Press.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 5077–5084. AAAI Press.

Martelli, A. 1977. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence*, 8: 1–13.

McCarthy, J. 1958. Programs with Common Sense. 75–91. Her Majesty's Stationary Office, London.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Meurer, A.; Smith, C.; Paprocki, M.; Čertík, O.; Kirpichev, S.; Rocklin, M.; Kumar, A.; Ivanov, S.; Moore, J.; Singh, S.; Rathnayake, T.; Vig, S.; Granger, B.; Muller, R.; Bonazzi, F.; Gupta, H.; Vats, S.; Johansson, F.; Pedregosa, F.; Curry, M.; Terrel, A.; Štěpán Roučka; Saboo, A.; Fernando, I.; Kulal, S.; Cimrman, R.; and Scopatz, A. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3: e103.

Moraru, I.; Edelkamp, S.; Martinez, M.; and Franco, S. 2018. Planning-PDBs Planner. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 69–73.

Newell, A.; and Simon, H. A. 1963. GPS: A Program that Simulates Human Thought. In Feigenbaum, E. A.; and Feldman, J., eds., *Computers and Thought*, 279–293. Oldenbourg.

Nir, R.; Shleyfman, A.; and Karpas, E. 2021. Learning-based Synthesis of Social Laws in STRIPS. In Ma, H.; and Serina, I., eds., *Proceedings of the 14th Annual Symposium on Combinatorial Search (SoCS 2021)*, 88–96. AAAI Press.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1983–1990. AAAI Press.

Núñez, S.; Borrajo, D.; and Linares López, C. 2014. MIPlan and DPMPlan. In *Eighth International Planning Competition (IPC-8): Planner Abstracts*.

Palacios, H.; and Geffner, H. 2009. Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *Journal of Artificial Intelligence Research*, 35: 623–675.

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

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; ; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, É. 2011. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting Problem Symmetries in State-Based Planners. In Burgard, W.; and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 1004–1009. AAAI Press.

Ramesh, A.; Pavlov, M.; Goh, G.; Gray, S.; Voss, C.; Radford, A.; Chen, M.; and Sutskever, I. 2021. Zero-Shot Text-to-Image Generation. arXiv:2102.12092 [cs.CV].

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. In *ICAPS 2020 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, 16–24.

Roberts, M.; and Howe, A. E. 2006. Directing a Portfolio with Learning. In Ruml, W.; and Hutter, F., eds., *AAAI 2006 Workshop on Learning for Search*, 129–135.

Roberts, M.; and Howe, A. E. 2009. Learning from planner performance. *Artificial Intelligence*, 173: 536–561.

Röger, G.; and Helmert, M. 2010. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 246–249. AAAI Press.

Rubinstein, R. Y. 1997. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1): 89–112.

Rudin, C. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5): 206–215.

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A. C.; and Fei-Fei, L. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3): 211–252.

Sagiv, M.; Reps, T.; and Wilhelm, R. 1999. Parametric shape analysis via 3-valued logic. In *Proceedings of the Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*, 105–118.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Seipp, J. 2017. Better Orders for Saturated Cost Partitioning in Optimal Classical Planning. In Fukunaga, A.; and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 149–153. AAAI Press.

Seipp, J. 2018a. Fast Downward Remix. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 74–76.

Seipp, J. 2018b. Fast Downward Scorpion. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 77–79.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning Portfolios of Automatically Tuned Planners. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 368–372. AAAI Press.

Seipp, J.; and Helmert, M. 2013. Counterexample-guided Cartesian Abstraction Refinement. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 347–351. AAAI Press.

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

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic Configuration of Sequential Planning Portfolios. In Bonet, B.; and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3364–3370. AAAI Press.

Seipp, J.; Sievers, S.; and Hutter, F. 2014a. Fast Downward Cedalion. In *Eighth International Planning Competition (IPC-8): Planner Abstracts*, 17–27.

Seipp, J.; Sievers, S.; and Hutter, F. 2014b. Fast Downward Cedalion. In *Eighth International Planning Competition (IPC-8) Planning and Learning Part: Planner Abstracts*.

Seipp, J.; Sievers, S.; and Hutter, F. 2014c. Fast Downward SMAC. In *Eighth International Planning Competition (IPC-8) Planning and Learning Part: Planner Abstracts*.

Seipp, J.; Torralba, Á.; and Hoffmann, J. 2022. PDDL Generators. `https://doi.org/10.5281/zenodo.6382173`.

Serban, I. V.; Sordoni, A.; Bengio, Y.; Courville, A.; and Pineau, J. 2016. Building End-To-End Dialogue Systems Using Generative Hierarchical Neural Network Models. In Schuurmans, D.; and Wellman, M., eds., *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, 3776–3783. AAAI Press.

Sharma, S.; Gupta, J.; Tuli, S.; and Rohan Paul, M. 2022. GoalNet: Inferring Conjunctive Goal Predicates from Human Plan Demonstrations for Robot Instruction Following. In *ICAPS 2022 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*.

Shen, J.; Pang, R.; Weiss, R. J.; Schuster, M.; Jaitly, N.; Yang, Z.; Chen, Z.; Zhang, Y.; Wang, Y.; Skerrv-Ryan, R.; Saurous, R. A.; Agiomvrgiannakis, Y.; and Wu, Y. 2018. Natural TTS Synthesis by Conditioning Wavenet on MEL Spectrogram Predictions.

In *Proceedings of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2018)*, 4779–4783.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 574–584. AAAI Press.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and Symmetries in Classical Planning. In Bonet, B.; and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3371–3377. AAAI Press.

Sievers, S. 2017. *Merge-and-shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation*. Ph.D. thesis, University of Basel.

Sievers, S. 2018. Fast Downward Merge-and-Shrink. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 85–90.

Sievers, S.; and Katz, M. 2018. Metis 2018. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 83–84.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019a. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7715–7723. AAAI Press.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019b. Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 446–454. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In Brodley, C. E.; and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2358–2366. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An Analysis of Merge Strategies for Merge-and-Shrink Heuristics. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 294–298. AAAI Press.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.;

Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587): 484–489.

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

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676): 354–359.

Speck, D.; Biedenkapp, A.; Hutter, F.; Mattmüller, R.; and Lindauer, M. 2021. Learning Heuristic Selection with Dynamic Algorithm Configuration. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 597–605. AAAI Press.

Speck, D.; Geißer, F.; and Mattmüller, R. 2018. SYMPLE: Symbolic Planning based on EVMDDs. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 91–94.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning Generalized Policies without Supervision Using GNNs. In *ICAPS 2022 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*.

Ståhlberg, S.; Francès, G.; and Seipp, J. 2021. Learning Generalized Unsolvability Heuristics for Classical Planning. In Zhou, Z.-H., ed., *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*, 4175–4181. IJCAI.

Steinmetz, M.; Fišer, D.; Eniser, H. F.; Ferber, P.; Gros, T.; Heim, P.; Höller, D.; Schuler, X.; Wüstholz, V.; Christakis, M.; and Hoffmann, J. 2022a. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 353–361. AAAI Press.

Steinmetz, M.; Hoffmann, J.; Kovtunova, A.; and Borgwardt, S. 2022b. Classical Planning with Avoid Conditions. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9944–9952. AAAI Press.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Sudry, M.; and Karpas, E. 2022. Learning to Estimate Search Progress Using Sequence of States. In Thiébaux, S.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, 362–370. AAAI Press.

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

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient Symbolic Search for Cost-optimal Planning. *Artificial Intelligence*, 242: 52–79.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 376–384. AAAI Press.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6294–6301. AAAI Press.

Vallati, M. 2012. A Guide to Portfolio-Based Planning. In Sombattheera, C.; Loi, N. K.; Wankar, R.; and Quan, T. T., eds., *Proceedings of the 6th International Workshop on Multi-disciplinary Trends in Artificial Intelligence (MIWAI 2012)*, volume 7694, 57–68. Springer.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *Proceedings of the Sixth International Conference on Learning Representations (ICLR 2018)*. OpenReview.net.

Virseda, J.; Borrajo, D.; and Alcázar, V. 2013. Learning heuristic functions for cost-based planning. In *Proceedings of ICAPS workshop on Planning and Learning*.

Wehrle, M.; and Helmert, M. 2014. Efficient Stubborn Sets: Generalized Algorithms and Selection Strategies. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 323–331. AAAI Press.

Wichlacz, J.; Höller, D.; and Hoffmann, J. 2022. Landmark Heuristics for Lifted Classical Planning. In De Raedt, L., ed., *Proceedings of the 31th International Joint Conference on Artificial Intelligence (IJCAI 2022)*, 4665–4671. IJCAI.

Wilt, C.; and Ruml, W. 2014. Speedy versus Greedy Search. In Edelkamp, S.; and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 184–192. AAAI Press.

Wilt, C.; and Ruml, W. 2015. Building a Heuristic for Greedy Search. In Lelis, L.; and Stern, R., eds., *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, 131–139. AAAI Press.

Wilt, C.; and Ruml, W. 2016. Effective Heuristics for Suboptimal Best-First Search. *Journal of Artificial Intelligence Research*, 57: 273–306.

Yu, L.; Kuroiwa, R.; and Fukunaga, A. 2020. Learning Search-Space Specific Heuristics Using Neural Network. In *ICAPS 2020 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 1–8.

Curriculum Vitaei

Due to data protection, the curriculum vitaei is not included in the digital version.